Simulations stochastiques en environnements distribués. Application aux grilles de calcul

Romain Reuillon

Université Blaise Pascal

THESE

pour obtenir le grade de

Docteur de L'Université Blaise Pascal
Ecole Doctorale Sciences Pour L'Ingénieur

Discipline Informatique

présentée par

M. Romain Reuillon

Simulations stochastiques en environnements distribués
Application aux grilles de calcul

---

sous la direction du Professeur

David R.C. Hill

Soutenue publiquement le 28 novembre 2008

Rapporteurs :	M. Jean Pierre Briot, Directeur de Recherche CNRS, Université de Paris 6
	M. Hans VanGheluwe, Professeur, Université McGill, Canada
Examinateurs :	M. Vincent Breton, Directeur de Recherche CNRS, IN2P3 Clermont-Fd
	M. Eric Innocenti, Maître de Conférences, Université de Corse.
Directeur :	M. David R.C. Hill, Professeur, Université Blaise Pascal
Co-direction :	M. Mamadou K. Traore, Maître de Conférences, Université Blaise Pascal
Invité :	M. Guillaume Deffuant, Directeur de Recherche, CEMAGREF, Clermont-Fd

DILBERT *By* Scott Adams

Remerciements

Je tiens à remercier toutes les personnes sans qui ses travaux n'aurai pas vu le jour :

Le **LIMOS** (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes) pour m'avoir accueilli pendant la durée de mes travaux de Doctorat.

Christophe Gouinaud, chercheur au LIMOS et grand marabout des systèmes pour son aide et ses précieux conseils.

Françoise Toledo et **Béatrice Bourdieu** pour leur précieuse assistance pour les tâches administratives.

Vincent Bara pour m'avoir aimablement fourni ses résultats de simulations stochastiques.

Les étudiants de **l'ISIMA** avec qui j'ai eu un grand plaisir à travailler au cours de projets encadrés de 1ère, 2ème et 3ème année.

Yannick Legré pour son aide et les accès à la grille « biomed ».

L'association **HealthGrid** pour son partenariat dans la mise en place d'ateliers de travail autour de l'installation de services de grille.

La **région Auvergne** et le grand projet Instruire (**Auvergrid**) pour le financement de ses travaux.

Laurence Girolami, Pierrick Noblet et **Jean-Luc Reuillon** pour la relecture de ce manuscrit.

Les menbres du jury : **Jean-Pierre Briot**, Hans **Vangheluwe, Vincent Breton, Eric Innocenti, Guillaume Deffuant.**

Mamadou Traoré pour son suivi de mes travaux, ses conseils précieux et son investissement lors de la rédaction de mon manuscrit.

Et pour le dernier mais pas le moindre, je tiens à remercier mon directeur de thèse **David (Benny) Hill** pour son écoute, son soutien, son encadrement, sa disponibilité, sa direction et ses judicieux conseils tout au long de mon travail de thèse.

Table des matières

Table des figures

Table des tableaux

Table des codes

Introduction générale

I Contexte

La révolution informatique initiée lors de la deuxième partie du 20ème siècle place la simulation informatique en tant que paradigme très populaire pour l'investigation scientifique. Elle est devenue l'outil privilégié dans l'étude des systèmes stochastiques dynamiques complexes (Pawlikowski, Towards Credible and Fast Quantitative Stochastic Simulation 2003b). Pawlikowski nous donne différentes conditions pour obtenir des résultats crédibles en simulation :

- un modèle valide, avec un niveau de détail approprié,
- une implémentation rigoureuse et absente d'erreurs logiques,
- un contexte expérimental valide.

De nombreux auteurs se sont intéressés à la validation et au crédit que l'on peut accorder aux modèles, le lecteur intéressé par une synthèse sur ce sujet peut consulter (Balci 1998).

Un processus stochastique, par opposition à un processus déterministe, utilise des sources de hasard pour en tirer un avantage. C'est-à-dire qu'il permet généralement d'estimer un résultat le plus souvent incalculable de manière déterministe. Les

simulations stochastiques sont donc conçues en utilisant des sources de hasard. De la qualité des ces sources va dépendre la qualité des résultats de la simulation.

La génération de hasard sur un ordinateur parfaitement déterministe soulève des problèmes liés à la manière dont est généré l'aspect aléatoire. Les séquences de nombres aléatoires peuvent, soit être prélevées dans l'environnement, soit être générées par des algorithmes déterministes. Dans ce cas, le plus répandu, il n'est plus question de hasard, mais de pseudo-hasard pour lequel il est impossible d'éviter certaines régularités ou structures dans les séquences générées. Ces dernières peuvent biaiser les résultats de simulations.

L'utilisation de simulations stochastiques permet l'obtention de résultats approximés pour des problèmes impossibles à résoudre avec des algorithmes déterministes. L'obtention d'une approximation intéressante pour de nouvelles applications scientifiques demande, dans certains cas, un temps de calcul de plusieurs dizaines d'années. Ce type d'application est irréalisable en pratique avec un seul ordinateur séquentiel. Une des pistes afin de réduire le temps global de simulation est de paralléliser le code du simulateur sur des fermes de calcul en utilisant des bibliothèques d'envoi de messages comme par exemple dans (Hill, A Design Pattern for Object-Oriented Distributed Simulation of Large Scale Ecosystems 1996). Cependant la parallélisation des réplications des simulations stochastiques permet d'exploiter plus efficacement la puissance de nouveaux environnements de calcul massivement distribués arrivant à maturité, comme les grilles de calcul et de ce fait d'obtenir des gains en terme temps de calcul très importants.

II Objectifs

Les simulations stochastiques utilisant de nombreuses réplications indépendantes présentent en effet un aspect intrinsèquement parallèle qui, sous certaines conditions, permet d'exploiter la puissance d'environnements de calculs massivement distribués. L'exécution d'une simulation stochastique requiert l'exécution de nombreuses expériences indépendantes afin de réduire l'erreur statistique sur les résultats. L'indépendance représente cependant une contrainte forte qui doit être assurée. Ainsi l'exécution en parallèle de ces expériences pose de nouveaux problèmes liés à

l'indépendance des flux de nombres aléatoires utilisés par chacune d'elle dont certains aspects sont exposés dans (Traoré et Hill 2001).

L'utilisation de flux de nombres aléatoires corrélés lors de l'exécution distribuée d'une simulation stochastique peut mener à la production de résultats biaisés et inutilisables dans un contexte scientifique. Nous, nous sommes rendu compte lors de nos premiers travaux de distribution des réplications d'une simulation stochastique sur grille de calcul (Maigne, et al. 2004), la tâche de parallélisation d'une simulation stochastique est complexe. Elle nécessite une connaissance spécifique afin d'éviter d'introduire, par cette opération, des biais dans les résultats de simulation. Il est important pour l'expérimentateur non spécialiste dans la distribution de simulations stochastiques de disposer d'outils qui lui permettront de s'assurer d'un maximum de rigueur dans le processus de distribution.

Quand bien même la distribution d'une simulation stochastique se base sur une parallélisation rigoureuse du générateur de nombres pseudo-aléatoires, aucune garantie ne peut être apportée quand à la validité des résultats de cette simulation. Comme pour le cas des simulations stochastiques séquentielles (L'Ecuyer, 1998), il semble que la méthode la plus fiable afin de détecter un biais du à l'utilisation d'un générateur de nombres pseudo-aléatoires parallèle dans les résultats d'une simulation stochastique distribuée est d'exécuter cette simulation plusieurs fois avec divers générateurs de nombres pseudo-aléatoires parallèles et de comparer les résultats obtenus.

Ces précautions prises, les simulations stochastiques peuvent être considérées comme des applications de choix de type « killer app » pour les environnements distribués. Elles permettent par conception d'utiliser la puissance de nombreux processeurs distribués sans communication interprocessus. Il est ainsi possible d'obtenir des gains en temps de calcul décroissant linéairement avec le nombre de processeurs utilisés.

Bien que l'appellation grille de calcul soit une analogie à la simplicité d'utilisation de la grille électrique américaine, l'utilisation d'environnements distribués comme les grilles de calcul demande un temps de prise en main significatif et un surplus de travail. Avant la mise à disposition des outils présentés dans ce manuscrit, aucun environnement de distribution spécifique à la distribution des réplications des simulations stochastiques sur grille de calcul n'était disponible. Mes travaux de thèse

visent ainsi à rendre disponible pour les modélisateurs des outils de distributions des réplications des simulations stochastiques sur des environnements de calcul distribué en accord avec l'état de l'art.

L'objectif principal de ce travail est de proposer des outils et des méthodes permettant à l'expérimentateur non spécialiste une distribution rigoureuse d'une simulation stochastique. Les outils proposés devront ainsi masquer la double complexité inhérente d'une part à la parallélisation des simulations stochastiques par distribution des réplications et d'autre part à l'exploitation des environnements de calcul distribué. Il est nécessaire de fournir plusieurs outils permettant :

- de paralléliser les générateurs de nombres pseudo-aléatoires existants avec diverses méthodes issues de l'état de l'art,
- de faciliter la production de résultats de simulations stochastiques basés sur différents générateurs de nombres pseudo-aléatoires parallèles afin de les comparer,
- de permettre la génération automatique de dizaines de milliers de « jobs » de simulation prêts pour une exécution en environnement distribué et correspondants à la distribution des réplications d'une simulation stochastique,
- de permettre l'utilisation de simulateurs existants (sans modification du code) et le support des générateurs de nombres pseudo-aléatoires déjà implémentés dans ces simulateurs,
- de supporter l'utilisation d'environnements d'exécution existants comme les gestionnaires d'exécution par lot pour les fermes de calcul et les intergiciels de grille.

III Plan

Ce manuscrit aborde dans le premier chapitre la problématique de la distribution des réplications des simulations stochastiques et met en lumière la difficulté de cette tâche. Il dépeint un contexte actuel, scientifique et technique, risqué et montre les conséquences d'erreurs, ou de négligences, dans la distribution des simulations stochastiques, sur la qualité des résultats de simulation.

Le deuxième chapitre présente les méthodes et outils existants pour résoudre ce problème et montre leur nette insuffisance.

Le troisième chapitre propose l'implémentation de deux outils visant à combler certaines lacunes par rapport aux solutions disponibles.

Le quatrième chapitre expose des réalisations utilisant ces outils et montre leurs utilisations dans un contexte pratique et sur des cas d'utilisation réels.

Enfin un bilan et des perspectives sont proposés dans une conclusion générale.

Chapitre 1 : Problématique

I Introduction

La méthode de Monte-Carlo apporte des résultats approximés à une variété de problèmes mathématiques par échantillonnage statistique sur un ordinateur. Cette méthode s'applique aussi bien à des problèmes sans contenu probabiliste qu'à ceux présentant un contenu probabiliste inhérent. Parmi toutes les méthodes basées sur l'évaluation de *n* points dans un espace à *m* dimensions, l'erreur absolue de la méthode de Monte-Carlo décroit selon $n^{-1/2}$, en l'absence d'une structure spéciale exploitable (comme par exemple avec l'utilisation de séquences à discrépance faible (Sobol 1976)). L'erreur de toutes les autres méthodes décroit au mieux avec $n^{-1/m}$. Alors que la complexité de calcul de la solution exacte à un problème combinatoire augmente de façon exponentielle avec la taille de ce problème, la méthode de Monte-Carlo offre souvent une estimation de la solution exacte avec une erreur tolérable pour une complexité polynomiale (Fishman, Monte Carlo: Concepts, Algorithms, and Applications 1995).

Si la simulation stochastique permet de résoudre des problèmes complexes elle présente de nombreux écueils dus à l'utilisation de sources d'aléa non appropriées ou

biaisées ainsi qu'à de mauvaises pratiques dans leur utilisation. De plus, elle n'en est pas moins consommatrice de temps de calcul. Pour exemple, certaines simulations en reconstruction tomographique ne convergent pas en moins de trois ans de calcul sur un ordinateur récent et pour des cas d'études simples (Lazaro, El Bitar, et al. 2005) (El Bitar, et al. 2006). Pour ces cas de simulation, il est nécessaire d'utiliser des environnements de calcul distribué afin d'obtenir des résultats de simulation utilisables. Même si, comme il fut noté dés les premières heures de l'informatique (Metropolis et Ulam 1949), les simulations de type Monte-Carlo qui utilisent de nombreuses réplications présentent un aspect naturellement parallèle, de nouveaux problèmes difficiles à résoudre liés à la génération de flux de nombres aléatoires parallèles naissent avec l'exécution distribuée de simulations stochastiques à grande échelle.

Ce chapitre expose dans une première partie les écueils liés à l'exécution de simulations stochastiques. Dans une deuxième partie, il présente plus spécifiquement les problèmes posés par l'exécution distribuée de telles simulations.

II Les écueils de la simulation stochastique

Bien que couramment utilisée dans beaucoup de domaine de recherche, la simulation stochastique sur ordinateur présente plusieurs difficultés importantes. La simulation des processus aléatoires nécessite l'utilisation de séquences de nombres aléatoires. La génération de telles séquences représente une difficulté majeure. C'est le thème de la première partie. Ainsi il est courant que de mauvais générateurs de nombres pseudo-aléatoires soient implémentés et distribués, même dans des logiciels réputés. Ceci fait l'objet de la deuxième partie. Enfin la troisième partie expose les mauvaises pratiques liées à l'utilisation de simulations stochastiques dans certains domaines, et pose le problème de la crédibilité des résultats publiés.

II.1 La difficulté de générer des séquences de nombres pseudo-aléatoires

II.1.1 La notion de hasard

Les simulations stochastiques utilisent des variables aléatoires, ou plus précisément, des réalisations de ces variables aléatoires, sous forme de séquences de nombres générés simulant le « hasard ».

Un bref historique de la notion de « hasard », présenté dans (Wolfram 2002) p. 967, pose le problème de l'appréhension du hasard par l'être humain : « *In the antiquity, it was often assumed that all events must be governed by deterministic fate – with any apparent randomness being the results of arbitrariness on the part of the gods. Around 330 BC Aristotle mentioned that instead randomness might just be associated with coincidences outside whatever system one is looking at, while around 300 BC Epicurus suggested that there might be randomness continually injected into the motion of all atoms*[1]*. The rise of emphasis on human free-will eroded belief in determinism, but did not especially address issues of randomness* ».

De nos jours La définition du dictionnaire en ligne de l'académie française[2] du mot « hasard », n'est pas plus précise que celle d'Aristote : « *n. m. XIIe siècle, hasart, jeu de dés. Emprunté, par l'intermédiaire de l'espagnol azar, de l'arabe az-zahr, jeu de dés. Au singulier. Rencontre imprévisible de séries d'évènements indépendants et ne résultant d'aucune intention* ».

Afin d'appréhender plus précisément cette notion, tournons nous vers les mathématiques et la physique. D'après (Stewart 1998), le hasard est introduit en mathématique par Blaise Pascal au milieu du XVII^{ème} siècle pour répondre à des questions dans le domaine du jeu, il fonde ainsi la théorie de la probabilité. Celle-ci prend la dimension de champ d'étude mathématique avec la publication de Laplace en 1812 : « Théorie analytique des probabilités ».

Durant la première moitié du XIX^{ème} siècle, Adolphe Quételet constate que nombres de paramètres prélevés dans la société tels que la taille des hommes d'une population, les taux de crimes et de suicides, sont distribués selon des lois normales. L'utilisation des statistiques fait alors naître les sciences sociales.

Lors de la deuxième moitie du XIX^{ème} siècle, Francis Galton étudie les caractéristiques d'une population cherchant à comprendre l'hérédité. Il formule ainsi l'analyse de régression qui permet d'inférer des tendances sous-jacentes à partir de données aléatoires.

[1] La première référence au concept d'atome remonte au 6eme siècle av. J.C. en Inde Ancienne. Les références à la notion d'atomes en occident émergent un siècle plus tard avec Leucippe, philosophe présocratique, dont le disciple Démocrite décrivit les atomes comme étant des corpuscules solides et indivisibles, séparés par des intervalles vides, et dont la taille fait qu'ils échappent à nos sens.

[2] http://www.academie-francaise.fr

Enfin, en 1873, James Clerk Maxwell propose d'utiliser des méthodes statistiques pour l'avancement de la science et l'étude des systèmes complexes. Il cite alors l'impossibilité d'étudier chaque individu d'une population de molécules, leur nombre rend le système trop complexe pour être formulé par des équations déterministes. Ainsi la physique déterministe et les mathématiques servent à étudier les systèmes simples avec des comportements simples et les statistiques à étudier des systèmes complexes avec des comportements complexes. « *Un terme nouveau est créé pour exprimer que même le hasard obéit à ses propre lois : la stochastique. (Le mot grec stochastikos veut dire « habile à viser » et transmet l'idée de l'utilisation des lois du hasard en vue d'un avantage personnel.)* » (Stewart 1998).

Une question simple vient alors chambouler cette dichotomie : « *Un système déterministe peut-il se comporter comme un système aléatoire ? […] Tout le progrès de la science était fondé sur la croyance que la manière de chercher la simplicité dans la nature était de trouver des équations simples pour la décrire. Quelle question stupide !* » (Stewart 1998). Cette question « stupide » fait cependant émerger la caractérisation de système chaotique. Dans « La théorie du chaos » (Gleick 1989), on trouve une illustration de l'imprévisibilité d'un comportement chaotique du à l'effet dit « papillon » : « *Pour de petits phénomènes météo – et à l'échelle d'une prévision globale, petit peut signifier des orages ou des blizzards -, toute prédiction perd très vite de sa fiabilité. Les erreurs et les incertitudes se multiplient, s'amplifient en cascade et génèrent des processus turbulents, des tornades et des bourrasques, jusqu'aux tourbillons de la dimension d'un continent que seuls les satellites peuvent détecter* ».

Au-delà de la physique newtonienne qui présentait le monde comme déterministe, où la place du hasard ne se trouvait que dans les incertitudes liées aux mesures, au delà des comportements chaotiques des systèmes où même des systèmes simples peuvent présenter des comportements tellement complexes que l'observateur les perçoit comme aléatoires, l'avènement de la physique quantique place aujourd'hui le hasard comme une propriété fondamentale du monde physique. Il est donc théoriquement possible d'observer le monde physique pour y prélever des séquences de nombres réellement aléatoires.

II.1.2 Les sources de hasard

Le hasard issu de systèmes chaotiques et le hasard physique peuvent être utilisés pour générer des séquences de nombres aléatoires en simulation stochastique. Les méthodes de génération de séquences de nombres aléatoires utilisables en simulation peuvent se distinguer selon les mécanismes qui induisent le comportement aléatoire :

- Le hasard physique : certains phénomènes aléatoires sont directement issus de phénomènes physiques, comme les mouvements browniens (Levy 1965) ou certaines mesures en physique quantique. A l'heure actuelle des sociétés proposent des générateurs quantiques de nombres aléatoires sous forme de carte PCI ou de périphérique USB 2.0[3] et des sites internet proposent le téléchargement de nombres aléatoires générés en observant des phénomènes quantiques[4].

- Le hasard issu de l'observation de phénomènes chaotiques : contrairement aux systèmes aléatoires aux sens physiques, le comportement des systèmes chaotiques n'est pas réellement aléatoire mais fortement sensible aux petites variations des conditions initiales. Il est impossible de prévoir leur comportement autrement que par simulation. C'est le cas, des attracteurs de Lorentz (Lorenz 1963) et du système double pendule traité par exemple dans (Zhou et Whiteman 1996). Les générateurs de séquences de nombres aléatoires basés sur l'observation de systèmes chaotique (Stojanovski et Kocarev 2001) (Drutarovsky et Galajda 2007) sont le plus souvent utilisés en cryptographie.

- Le pseudo-hasard : le pseudo-hasard est un comportement aléatoire directement induit par un système. Le système ne peut être considéré comme aléatoire mais il est très difficile, pour un observateur extérieur de voir des régularités dans son comportement, on parle alors de systèmes pseudo-aléatoires. Ce système est généralement basé sur l'utilisation de suites mathématiques conçues pour obtenir un comportement le plus chaotique possible et pour générer des suites de nombres contenant le moins de régularités observables possibles.

[3] http://www.idquantique.com
[4] http://www.randomnumbers.info et http://random.irb.hr

Ces différentes sources de hasard sont utilisables en simulation. Cependant chacune d'entre elle présente des inconvénients.

II.1.3 Quel hasard pour la simulation stochastique ?

Les premières tentatives pour générer des nombres réellement aléatoires se firent à l'aide de systèmes mécaniques, comme le tirage de boules de loterie. Ces méthodes donnaient des résultats satisfaisants mais elles étaient intrinsèquement lentes.

Il est admis que le statisticien L.H.C. Tippett fut le premier à publier une table de nombres aléatoires en 1927 (Tippett 1927). Celle-ci contenait 41 600 valeurs générées en utilisant selon les sources : les surfaces des paroisses anglaises ou les registres de recensement de la population ne retenant que certains chiffres parmi les valeurs utilisées.

En 1939, une table de 100 000 nombres fut publiée (Kendall et Smith 1939). En 1956, la RAND corporation publia, après huit années de travail, une table de 1 million de nombres. Celle-ci a été générée en simulant une roulette sur les premiers ordinateurs à cartes perforées (RAND Corporation 1955). Les résultats furent filtrés et testés afin de produire une série de nombres de qualité. Une table si importante et si méticuleusement préparée n'avait encore jamais été disponible, cette publication représente une rupture dans l'utilisation des nombres aléatoires. Elle fit référence encore des années plus tard. Dans la publication (Landis et Feinstein 1973) parue en 1973, les auteurs se demandent si les générateurs pseudo-aléatoires des ordinateurs modernes sont aussi adaptés pour les simulations de Monté Carlo que la table de nombre « Rand Table » couramment utilisée.

Depuis, de nouvelles pistes ont étés étudiées pour générer de l'aléa dans un programme informatique : le bruit blanc de systèmes électroniques, la détection de l'émission radioactive de particules ou l'observation du fonctionnement de l'ordonnanceur d'un système d'exploitation (Castro 1996). Ces phénomènes, dont les propriétés stochastiques ont été démontrées, produisent des séries imprédictibles et considérées comme réellement aléatoires dans les années 80 (Leroudier 1980).

La génération de nombres aléatoires basée sur l'utilisation de phénomènes physiques aléatoires, ou chaotiques, est aujourd'hui couramment utilisée en cryptographie afin d'initialiser des générateurs déterministes, mais délaissée pour les

simulations informatiques. En effet ce type de génération présente trois défauts majeurs (Matsumoto, Saito, et al. 2006) :

- La nécessité de relier physiquement à l'unité de calcul une unité spécialisée produisant ou observant le phénomène utilisé pour la génération de nombres aléatoires. La maintenance de telles unités est coûteuse et la seule manière de détecter un disfonctionnement possible du dispositif matériel est de tester les nombres en permanence.
- Les séquences de nombres produites ne sont pas reproductibles, il est donc très difficile de développer et de déboguer des simulations avec de tels générateurs.
- Ces générateurs basés sur des phénomènes physiques peuvent être sujets à des interférences avec leur environnement (comme un changement de température ou les ondes électromagnétiques produites par un téléphone mobile) et de ce fait générer des séries biaisées.

Les inconvénients de telles méthodes de génération ont mené à l'utilisation et au développement d'algorithmes déterministes pour générer des nombres utilisables pour la simulation stochastique : les séquences de nombres pseudo-aléatoires.

II.1.4 Une utilisation hasardeuse des générateurs de nombres pseudo-aléatoires

En simulation, les séquences de nombres utilisées sont le plus souvent générées par des algorithmes parfaitement déterministes, c'est-à-dire des séquences de nombres pseudo-aléatoires. Le problème majeur est qu'il n'y a pas de définition raisonnable du terme pseudo-aléatoire (Matsumoto, Saito, et al. 2006). La définition la plus courante, qui est d'ailleurs aussi courante qu'approximative, est qu'un processus pseudo-aléatoire doit présenter un comportement apparent aléatoire au sens statistique. On retrouve cette définition dans (Leroudier 1980) : « *Il peut sembler paradoxal de vouloir générer une suite de nombres aléatoires au moyen d'un algorithme s'exécutant sur un ordinateur, puisque chaque nombre va dépendre d'une manière déterministe (par l'intermédiaire d'un algorithme) du ou des nombres qui le précèdent. Aussi devons nous nous entendre sur le hasard que nous désirons avoir lors de la génération des nombres. En fait, que les nombres soient générés par un algorithme déterministe ou non, nous importe peu. Le point de vue que nous adoptons est beaucoup plus pragmatique : une suite de nombres nous satisfera s'il est difficile d'y constater des régularités évidentes.* »

Si on souhaite être très rigoureux, on peut trouver des définitions précises, mais elles ne sont pas applicables à la génération de nombres pseudo-aléatoires pour la simulation stochastique. L'une d'entre elles a été donnée par Kolmogorov (Kolmogorov 1965) et Chaitin (Chaitin 1966) : une suite finie de nombres est aléatoire s'il n'existe pas de description de celle-ci plus courte qu'elle-même. Cette définition trouve des applications en théorie de l'information.

La deuxième définition rigoureuse est instiguée par (Yao 1982) (Shamir 1983) et (Blum et Micali 1984) pour des applications cryptographiques : une séquence de nombres pseudo-aléatoires est utilisée en cryptographie si elle peut être générée en un temps polynomial quand les termes de la récursion ainsi que les valeurs initiales sont connus, mais qu'il n'existe pas de moyen pour deviner les nombres suivants de la suite en un temps polynomial. Des générateurs correspondants à ces critères sont utilisés couramment en cryptographie, mais ils sont trop lents et certaines de leurs propriétés sont trop difficiles à étudier de manière théorique pour des calculs de type Monte-Carlo à grande échelle.

Si le terme pseudo-aléatoire est mal défini d'un point de vu théorique en simulation stochastique, les caractéristiques attendues pour un bon générateur de nombres pseudo-aléatoires sont définies d'un point de vue pratique. Les séquences produites par un générateur de nombres pseudo-aléatoires idéal pour la simulation (P. D. Coddington, Random number generator for parallel computers 1996) :

1. sont uniformément distribuées,
2. sont non corrélées,
3. ne se répètent jamais,
4. satisfont les tests statistiques connus pour vérifier l'aléa,
5. sont reproductibles, pour des considérations de débogage,
6. sont portables (obtention des mêmes séquences de nombres sur des ordinateurs différents),
7. peuvent être changées facilement en utilisant un germe,
8. peuvent se subdiviser facilement en beaucoup de sous-séquences indépendantes,

9. peuvent être générées rapidement en utilisant une taille mémoire limitée.

Un générateur qui posséderait les caractéristiques décrites par Coddington serait idéal. Dans la réalité un générateur pseudo-aléatoire peut être décrit comme un automate à état fini. L'état d'un générateur de nombres pseudo-aléatoires est caractérisé par la valeur de l'ensemble des variables qu'il manipule. Pour que l'automate ne retrouve jamais le même état (recommandation 3), il faut qu'il comporte une infinité d'états possibles qu'on peut numéroter de 0 à 2^{∞}. Il faudrait ainsi une infinité d'espace pour stocker le numéro de l'état courant d'un générateur acyclique dans la mémoire d'un ordinateur (ce qui va à l'encontre de la recommandation 9). Le générateur idéal n'existe pas ! Peu importe, mais d'un point de vue pratique avons-nous seulement les outils pour tester si un générateur est utilisable de manière satisfaisante pour des simulations stochastiques ?

Les nombres produits par des générateurs pseudo-aléatoires sont les résultats d'un ensemble de fonctions mathématiques, une structure apparaît donc systématiquement dans ces séquences de nombres. On dit que des corrélations sont présentes dans la séquence de nombres. En probabilité et en statistique, la corrélation ou coefficient de corrélation, est une mesure numérique de l'importance de la relation linéaire entre deux variables aléatoires. Le coefficient de corrélation donne une idée de la corrélation linéaire existant entre deux ensembles de données de même nombre de mesures, ou entre les réalisations de deux variables aléatoire x et y :

$$r = \frac{\sum(x_i - \bar{x})\,(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 (y_i - \bar{y})^2}}$$

Si le coefficient de corrélation est égal à zéro les variables aléatoires x et y sont dite non-corrélées linéairement. Si deux variables aléatoires sont indépendantes, alors leur coefficient de corrélation sera nul, cependant l'inverse n'est en général pas vrai. En effet il peut exister d'autres formes de corrélations non linéaires. D'où l'intérêt d'utiliser conjointement aux calculs de corrélations classiques des tests statistiques empiriques.

Les tests empiriques sont des simulations dont on connaît les résultats. Les simulations utilisées comme test sont sensibles à certaines structures induites par la génération déterministe des nombres pseudo-aléatoires. Même si une séquence de nombres passe avec succès tous les tests actuellement publiés, il n'est pas impossible,

que la structure présente dans la séquence de nombres biaise malgré tout les résultats d'une autre simulation :

« *Monte Carlo results are misleading when correlations hidden in the random numbers and in the system interfere constructively* » (Compagner 1995)

Les générateurs de nombres pseudo-aléatoires sont comme les médicaments actuels. Chaque type de générateur a ses effets secondaires. Il n'y a pas de générateur sûr. Tous les générateurs ont des régularités, qui occasionnellement peuvent devenir des défauts.

Les bons générateurs de nombres pseudo-aléatoires se caractérisent par des bases théoriques solides, consolidées par des preuves empiriques et un bon aspect pratique. Ils produiront des résultats corrects dans beaucoup de simulations, mais pas dans toutes (Hellekalek, Good random number generators are (not so) easy to find 1998b). Si la structure des nombres pseudo-aléatoires interfère avec une simulation stochastique donnée, alors les résultats de la simulation peuvent être inutilisables ou pire biaisés. Ce fait a été montré dans de nombreuses publications scientifiques, parmi elles : (Ferrenberg, Landau et Wong, Monte Carlo Simulation: Hidden errors from "good" random number generator 1992) (Grassberger 1993) (Sleke, Talapov et Schur 1993) (Schmid et Wilding 1995).

En simulation stochastique, il est important que les générateurs de nombres pseudo-aléatoires soient testés afin de réduire le risque de biais statistique sur les résultats de sortie. Ce risque est induit par des défaillances, corrélations ou régularités, présentes dans des séquences de nombres considérées, à tort, comme aléatoires. Les tests statistiques présentent alors un indicateur de la qualité des séquences de nombres pseudo-aléatoires, mais n'apportent en aucun cas une garantie absolue quand à l'utilisation de la séquence de nombres dans l'intégralité des cas de simulations.

II.1.5 Des défauts significatifs dans les générateurs de nombres pseudo-aléatoires

En 1946, John Von Neumann propose un générateur de nombres pseudo-aléatoires connu sous le nom de la méthode du « middle-square » (carré médian)[5]. Elle consiste à prendre un nombre, à l'élever au carré et à prendre les chiffres au milieu comme sortie. La sortie est utilisée comme germe pour l'itération suivante. Von Neumann utilisa des nombres comportant 10 chiffres. Toutefois, la périodicité du middle-square est faible. La

[5] http://fr.wikipedia.org

qualité des sorties dépend du germe initial. De plus « 0000 » constitue un état absorbant dans lequel l'algorithme tombe souvent. Von Neumann en était conscient mais il craignait que des retouches a priori nécessaires n'apportent d'autres vices cachés.

Depuis, différentes fonctions mathématiques ont étés reconnues comme génératrices de séquences de nombres pseudo-aléatoires. Les générateurs linéaires congruenciels ou LCG (Linear Congruential Generator) sont les générateurs les plus largement utilisés. Ce type de générateur pseudo-aléatoire a été proposé par Lehmer (Lehmer 1949) en 1949. Un générateur de type LCG(N, a, c, x_0) est caractérisé par une équation de la forme :

$$x_{j+1} = ax_j + c \; mod \; N$$

La période de ces générateurs est souvent égale à N si a et c sont choisis de façon convenable. N est généralement soit un nombre premier soit une puissance de 2 (2^{32} pour profiter de toute la plage des entiers non signés sur les ordinateurs 32 bits). Cependant si un LCG utilise un modulo de la forme 2^b sa période maximale est $2^{b-2} = \frac{N}{4}$. Cette période maximale est atteinte si et seulement si $a \; mod \; 8 = 3 \; ou \; 5$ et que le germe initial est impair. A ces conditions le générateur produit une permutation périodique de la moitié des nombres impairs entre 1 et 2^b-1 (Park et Miller 1988).

Les LCG ont plusieurs défauts majeurs. Tout d'abord, les bits de poids faibles produits par ce type de générateurs sont hautement corrélés (P. D. Coddington, Analysis of Random Number Generators Using Monte Carlo Simulation 1994) et d'autant plus lorsque N est une puissance de 2 (Mascagni, Parallel Pseudorandom Number Generation 1999). De plus, si on considère les nombres générés comme des tuples (x_n, x_{n+1}, …, x_{n+d}) et qu'on réalise un test spectral en les plaçant dans un espace à d dimensions, les séquences de nombres issues de LCG forment, quelques soient les paramètres du LCG, des structures en treillis réguliers (G. A. Marsaglia 1968) (Afflerbach et Grothe 1988). Par exemple, dans (Hellekalek, Good random number generators are (not so) easy to find 1998b), Hellekalek montre que le générateur LCG (2^{31}, 65539, 0, 1) a une couverture spectrale correcte en dimension 2, il occupe de l'espace de manière uniforme, mais se répartit sur seulement 15 plans en dimension 3 (Figure 1).

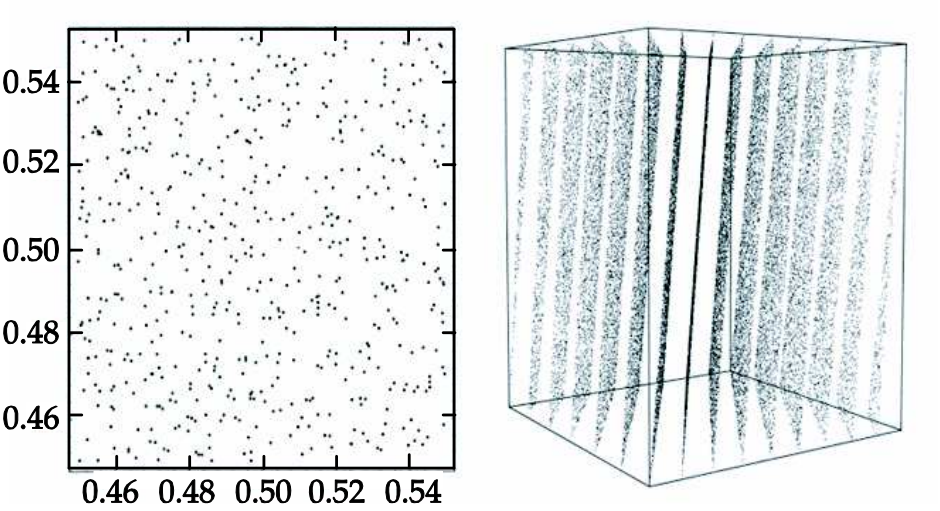


Figure 1 Extrait de (Hellekalek, Good random number generators are (not so) easy to find 1998b) : Analyse spectrale du LCG (2^{31}, 65539, 0, 1) en dimensions 2 et 3

Ce problème est d'autant plus fréquent que le nombre de dimensions est élevé. Il peut largement affecter les simulations stochastiques utilisant des espaces comprenant de nombreuses dimensions (comme par exemple certains calculs d'intégrales).

Enfin, les LCG sont connus pour avoir des corrélations à longue portée (Filk et Fredenhagen 1985) ou pseudo-périodes. C'est-à-dire que deux sous séquences d'une séquence plus longue présentent un fort degré de corrélation.

Il faut aujourd'hui considérer les générateur de type LCG qui utilisent des nombres sur 32 bits comme obsolètes. Dans le cas où M est codé sur 32 bits, la période maximale du générateur est de 2^{32}, soit environ quelques milliards de nombres. Sur les machines actuelles, capable de calculer plusieurs milliards d'opérations en virgule flottante par seconde, la période peut être épuisée en quelques minutes seulement. Il est ainsi possible d'augmenter la période maximum des générateurs LCG en utilisant des nombres sur 64 bits (G. Marsaglia 1985), cependant ceci ne résout pas les autres problèmes.

D'autre générateurs, appelés générateurs à récurrence multiple, ou MRG, généralisent le concept de générateur linéaire congruentiel. Ils exploitent une fonction de la forme suivante :

$$x_{j+1} = a_1 x_j + a_2 x_{j-1} + \ \ldots + a_k x_{j-k+1} + \ c \ mod \ N$$

Choisir un k supérieur à 1 augmente le temps de calcul mais améliore grandement la période et les propriétés statistiques du générateur. Quelques implémentations ont étés

proposés par P. L'Ecuyer dans (L'Ecuyer, Blouin et Couture, A search for good multiple recursive generators 1993) pour k=2.

Les LCG existent depuis les années 50 et présentent des défauts notables. De ce fait, d'autres algorithmes de génération de nombres pseudo-aléatoires ont étés conçus. Cependant, même l'un des meilleurs générateurs actuels, le Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997), qui présente une période de 2^{19937} tirages, qui est equidistribué dans 623 dimensions et qui est très recommandé pour son utilisation en simulation stochastique par son concepteur dans (Matsumoto, Saito, et al. 2006), a été pris en défaut (Panneton, L'Ecuyer et Matsumoto 2006). Comme le montre la Figure 2, lorsque le générateur est initialisé avec une large majorité de 0 et un seul bit à 1, il lui faut plus de 700 000 tirages pour retrouver un état avec des proportions de 0 et de 1 équilibrées, et donc dans ces conditions, la séquence de sortie présente un biais.

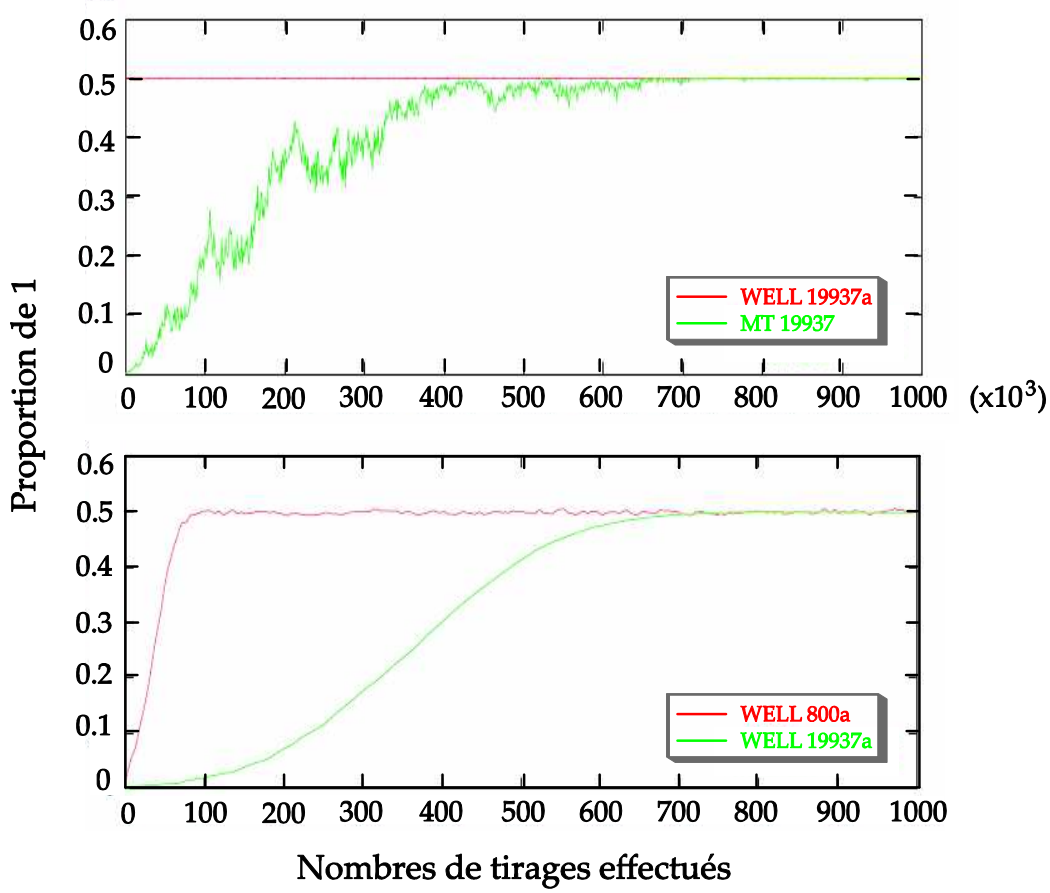


Figure 2 Extrait de (Panneton, L'Ecuyer et Matsumoto 2006) : Nombre de tirages pour retrouver des proportions de 1 et 0 équilibrées après une initialisation avec une large majorité de 0 des générateurs WELL et de Mersenne Twister 19937 (en ordonnée ou trouve la proportion de 1 et en abscisse le nombre de tirages effectués)

Un calcul de π par la méthode de Monté Carlo montre l'impact d'un tel biais sur des résultats de simulation. La Figure 3 présente les résultats d'un calcul de π en utilisant un générateur de type Mersenne Twister convenablement initialisé et un autre initialisé dans un état comprenant un seul bit à 1. Chaque réplication utilise le tirage de 1000 points dans un espace à 2 dimensions et consomme donc 2000 nombres. Les courbes présentent pour chaque simulation l'intervalle de confiance à 95%.

Cette figure met en évidence que le biais dans la génération des nombres ne permet pas d'avoir une approximation convenable de π même après 25 réplications (courbe foncée) alors qu'il suffit de 10 réplications pour avoir une estimation correcte de la valeur de π en utilisant une autre sous-séquence moins biaisée du cycle de génération du Mersenne Twister 19937 (courbe claire).

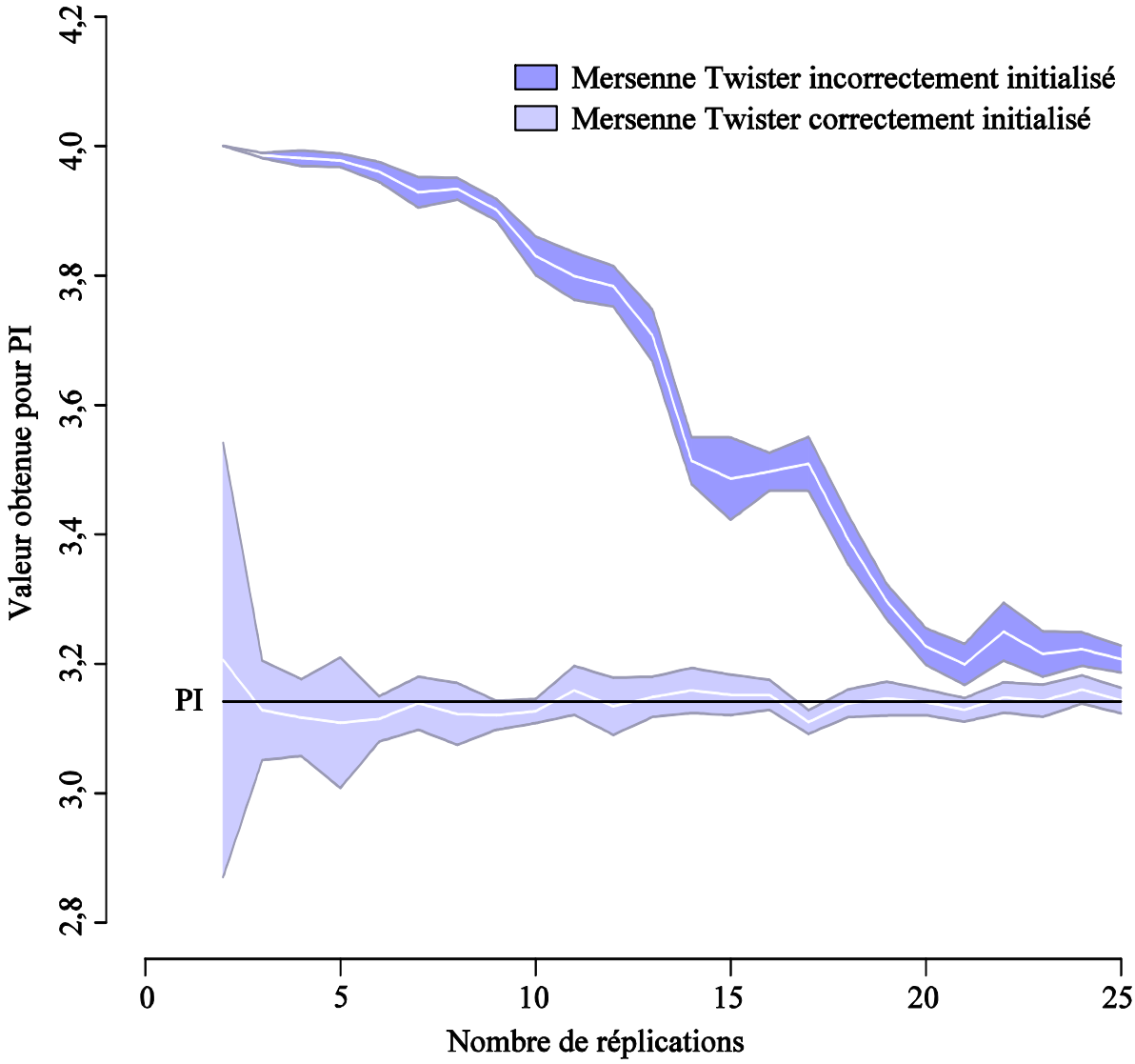


Figure 3 Influence d'une mauvaise initialisation du générateur Mersenne Twister 19937 sur un calcul de PI. Pour la courbe en clair, le générateur a été correctement initialisé et pour la courbe foncée l'état de départ du générateur ne comprenait qu'un seul bit à 1.

Dans cette partie nous avons vu que la génération de nombres aléatoires ou pseudo-aléatoires inhérente à la simulation stochastique n'est pas une mince affaire. Pour ne rien arranger, certains logiciels réputés et couramment utilisés proposent des générateurs de nombres pseudo-aléatoires présentant des défauts majeurs liés à leur faible qualité intrinsèque ou à la présence d'erreur d'implémentation. Certains de ces logiciels sont présentés dans la partie suivante.

II.2 De mauvais générateurs dans de bons logiciels

Les générateurs de nombres pseudo-aléatoires ont une obsolescence rapide au regard des applications nouvelles qui font une grande consommation de ces nombres. Un bon générateur actuel sera très certainement inutilisable dans 10 ans.

Dans (Wichmann et Hill 1982), les auteurs vantent les mérites d'un générateur d'une période de $2{,}78\ 10^{13}$ qui permet de lancer des simulations qui consomment 1000 nombres par secondes pendant 880 années. Depuis peu, nous avons des simulations dans des domaines tels que la médecine nucléaire qui consomment jusqu'à 15 milliards de nombres pour seulement 12 heures d'exécution (Maigne, et al. 2004) et le temps total de simulation pour obtenir des résultats statistiques corrects est d'au moins trois ans. Pour une telle simulation le générateur de Wichman et Hill, générerait 1 182 464 fois la même séquence de nombres.

On peut imaginer qu'avec des générateurs présentant une période inépuisable en pratique comme par exemple les générateurs de type Mersenne Twister (2^{19937} tirages pour certains d'entre eux), nous sommes aujourd'hui à l'abri de ce processus d'obsolescence. Il n'en est rien ! Sur le site internet de Makoto Matsumoto[6], le chercheur présente le générateur qu'il a conçu comme étant un générateur pour le 21ème siècle : « *Mersenne Twister: Random number generator for (the beginning of) 21st Century* ». Il ajoute cependant par prudence entre parenthèse : « pour le début du ». En effet, même si le générateur Mersenne Twister 19937 est conçu sur des bases théoriques solides, il n'est pas exclu que des défauts significatifs soient découverts dans les années à venir, comme de fortes corrélations dans les séquences de nombres pseudo-aléatoires générées. On trouve de ce fait couramment des générateurs inadaptés aux problématiques actuelles même dans de bons logiciels.

II.2.1 ROOT

Le logiciel ROOT est un cadriciel orienté objet développé par le CERN (Brun et Rademakers 1997) pour l'analyse efficace de grandes quantités de données. Dans ce logiciel, un générateur de nombres pseudo-aléatoires est implémenté dans la classe `TRandom`. Cette classe permet la génération de lois uniformes et non-uniformes. Toutes les générations de lois de probabilité sont basées sur le générateur uniforme. Ce générateur est, d'une part mal implémenté, et d'autre part présente de larges défauts

[6] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/eindex.html

(Heinrich, TRandom PitFalls 2006). Les nombres en double précision produits par cette classe sont des nombres compris entre 0 et 1. Pour générer de tels nombres on fixe la partie exposant à 0 et on initialise les bits de la mantisse aléatoirement. La mantisse d'un nombre double précision (Figure 4) comprend 52 bits. Seulement, l'implémentation du générateur dans ROOT n'initialise que 31 bits sur les 52 de manière pseudo-aléatoire, les autres sont fixés à zéro.

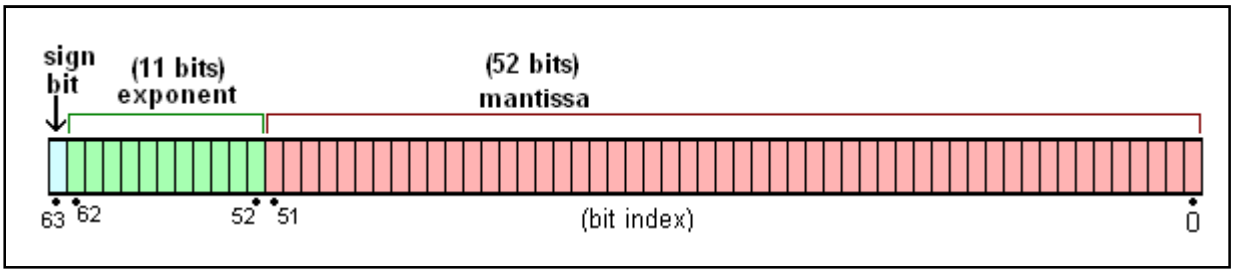


Figure 4 Extrait de Wikipedia : Structure d'un nombre double précision standard IEEE 754

Indépendamment de cette erreur d'implémentation, le générateur présente intrinsèquement des défauts. La génération d'une série de nombres après une initialisation du générateur `TRandom` avec le germe 2^{28} est présentée dans le Code 1. La période de répétition est alors de 2.

```
CINT/ROOT C/C++ Interpreter version 5.15.94, June 30 2003
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] TRandom r(1<<28);for(int i=0;i<10;++i) cout << r.Rndm() << '\n';
0.625
0.125
0.625
0.125
0.625
```

Code 1 Extrait de (Heinrich, TRandom PitFalls 2006) Génération d'une séquence de nombres pseudo-aléatoires avec la classe TRandom de ROOT

Enfin, la classe `TRandom` de ROOT permet de générer des nombres pseudo-aléatoires selon une approximation de la loi de poisson. L'article (Heinrich, TRandom PitFalls 2006) montre que l'approximation de la loi de poisson est si mauvaise qu'elle est inutilisable.

II.2.2 CLHEP

La bibliothèque CLHEP (Lönnblad 1994) (Class Library for High Energy Physic) également développée par le CERN depuis 1992, est utilisée dans des environnements

de simulations en physique des hautes énergies comme Geant4 (Allison et al. 2006) ou GATE (Jan et al. 2004).

Cinq des douze générateurs implémentés dans cette bibliothèque ne doivent pas être utilisés dans un contexte de simulation (Heinrich, Detecting a Bad Random Number Generator 2004).

```
double MTwistEngine::flat() {
  …
  y = mt[count624];
  y ^= ( y >> 11);
  y ^= ((y << 7 ) & 0x9d2c5680);
  y ^= ((y << 15) & 0xefc60000);
  y ^= ( y >> 18);

  return y * twoToMinus_32  +                    // Scale to range
       (mt[count624++] >> 11) * twoToMinus_53 +   // fill remaining bits
                      nearlyTwoToMinus_54;        // make sure non-zero
}
```

Code 2 Extrait de la classe MTwistEngine de CLHEP

De plus, à cause d'une erreur de programmation dans le générateur uniforme `RandomEngine`, les suites de nombres générées avaient pour moyenne théorique 5/8 sur les plateformes linux. Ce bug n'a été détecté et corrigé qu'en 2004 (Heinrich, Detecting a Bad Random Number Generator 2004).

Enfin, en observant le code source de CLHEP, nous avons trouvé un problème dans la génération des nombres doubles précision par l'algorithme Mersenne Twister 19937. Le Code 2 présente un extrait de la classe qui implémente l'algorithme Mersenne Twister 19937, de la version la plus récente de la bibliothèque CLHEP (2.0.3.1). Le nombre entier sur 32 bits `mt[count624]` est utilisé deux fois pour générer la mantisse du nombre double précision. Les bits de poids fort (51 à 19) et faible (18 à 0) sont donc identiques.

Nous avons vu dans cette partie que des générateurs de nombres pseudo-aléatoires de faible qualité ou présentant des erreurs d'implémentation sont couramment présent dans des logiciels réputés et utilisés à des fins scientifiques. Même si la qualité du générateur de nombres pseudo-aléatoires est importante, elle n'est pas suffisante.

Obtenir des résultats crédibles en simulation stochastique nécessite une démarche rigoureuse qui n'est pas toujours suivie comme le montre la partie suivante.

II.3 De mauvaises pratiques

La méthode de Monte-Carlo est basée sur une approche théorique solide, cependant en pratique, l'intégrité statistique de tout échantillonnage de type Monte-Carlo nécessite certaines précautions élémentaires.

Pour présenter des résultats issus d'une simulation stochastique, il est recommandé de mentionner :

- Le générateur de nombres pseudo-aléatoires utilisé pour la simulation.
- Le type de simulation.
- La méthode de traitement des résultats de sortie.
- L'erreur statistique associée aux résultats.

Ces recommandations ne sont pas d'usage systématique. Dans le domaine des télécommunications, la Figure 5, extraite de l'article (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks 2002), montre que de 1992 à 1998, plus de la moitié (51%) des articles de trois revues de références se rapportent à des résultats de simulations stochastiques. Cependant, la majorité des articles étudiés par l'article (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks 2002) ne commente pas la partie génération des nombres pseudo-aléatoires de leur simulation.

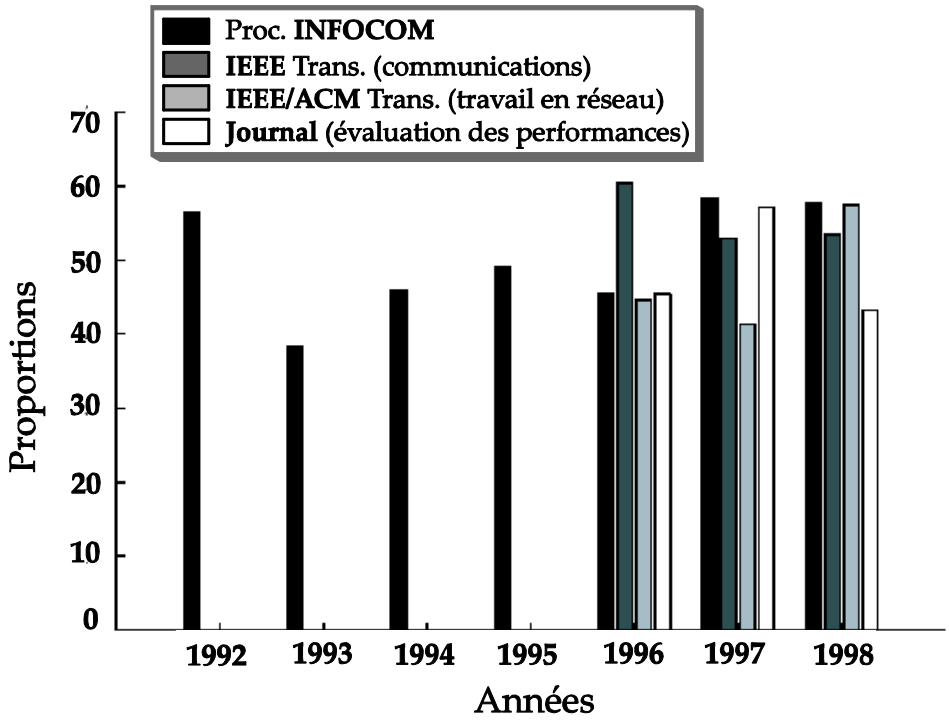


Figure 5 Extrait de (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks 2002) : Proportion des publications se rapportant à des résultats de simulations stochastiques dans trois revues de référence dans le domaine des télécommunications

De plus, comme le montre la Figure 6, en 1998, 55% des articles parus dans les « Proceedings of the INFOCOM » et utilisant des résultats de simulations stochastiques ne mentionnent pas le type de simulation stochastique utilisée. Une simulation de système terminant comporte des conditions de départ et sa fin est conditionnée par un événement donné. Pour un système de simulation non-terminant, la durée de simulation n'est pas finie. D'après (Banks 1998) le type de système étudié conditionne le traitement statistique des données de sortie. On peut donc aisément en déduire qu'aucun calcul de l'erreur statistique n'est mené sur les résultats de ces simulations stochastiques.

Enfin la Figure 6 montre que les articles qui font mention du type de simulation ne traitent pas, la plupart du temps, de l'erreur statistique associée aux résultats présentés.

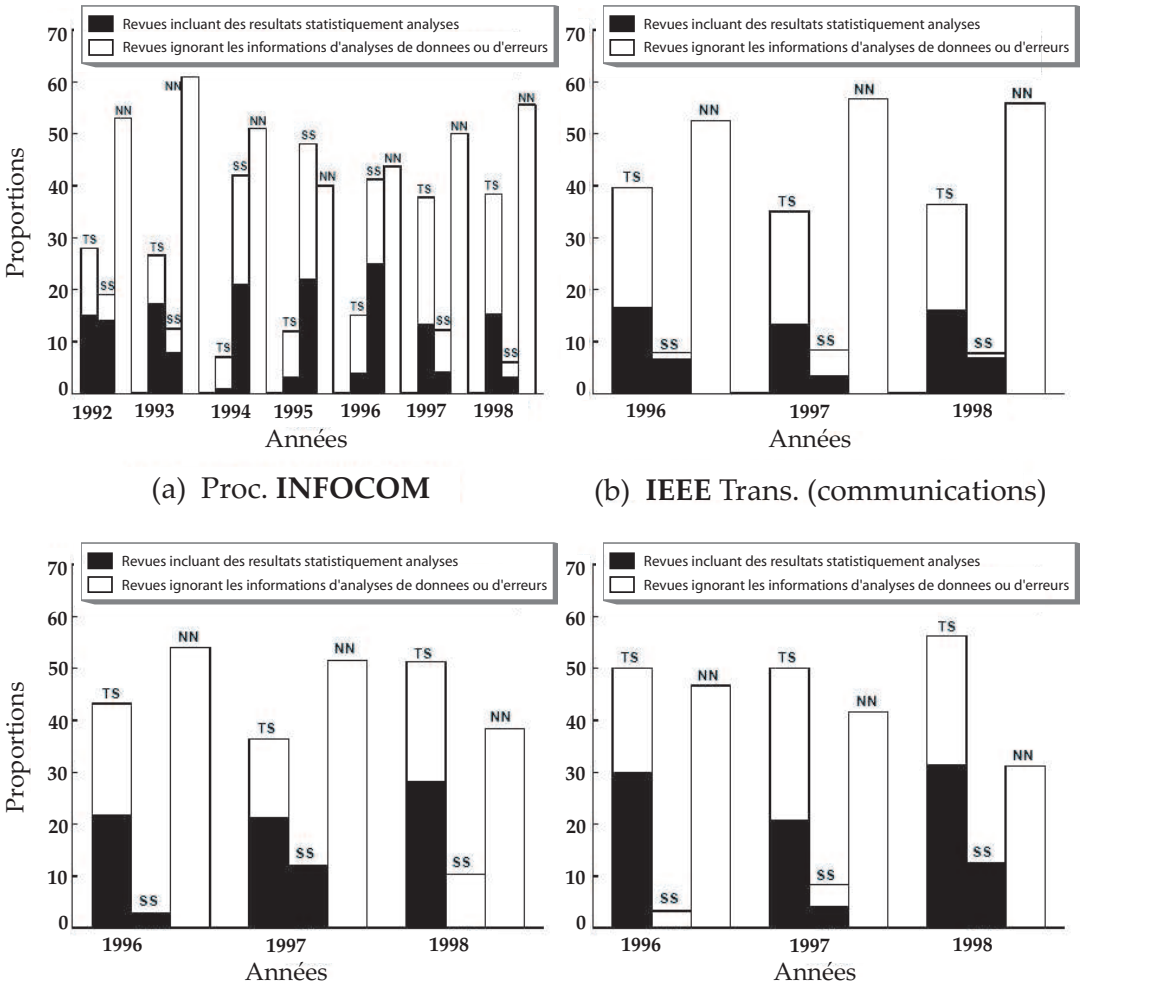


Figure 6 Extrait de (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks 2002) : Histogrammes des articles reportant des résultats de simulation publiés dans quatre revues différentes. TS : Papiers reportant des résultats de simulations terminant ; SS : Papiers reportant des résultats de simulations à état stationnaire ; NN : Papier sans information à propos du type de simulation exécuté

Ajoutons à cela :

- Qu'il n'existe pas de générateur de nombres pseudo-aléatoires sans défaut mais seulement des recettes de cuisine pour générer des nombres aléatoires. John Von Neuman affirme en 1951 : *« Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number – there are only methods to produce random numbers, and strict arithmetic procedures of course is not such a method … We are here dealing with mere "cooking recipes" for making digits… »*. Cette affirmation n'a pas été infirmée depuis et est même utilisée dans des ouvrage de référence comme « The art of computer programming volume II, Seminumerical

Algorithms » (D. E. Knuth, The art of computer programming, Vol. 2 Seminumerical Algorithms 3rd Edition 1997). On la trouve reprise dans des cours de 2007 dispensés à l'université de l'Illinois sur la génération de variables aléatoires pour la méthode de Monté Carlo (Koenker 2007).

- Que de nombreux générateurs de nombres pseudo-aléatoires furent développés dans les années 70 et présentent des défaillances significatives pour les simulations à grande échelle. Ainsi les générateurs `ran1` et `ran2` présentés dans la première édition de « Numerical Recipies in C » (Press, et al. 1988) sont actuellement considérés comme défectueux pour un usage en simulation (Antoch, Deshouillérs et Purnaba 1998). De même dans un ouvrage plus récent : la troisième édition du livre de référence « The art of computer programming volume II, Seminumerical Algorithms » (D. E. Knuth, The art of computer programming, Vol. 2 Seminumerical Algorithms 3rd Edition 1997), le générateur proposé nommé `ran_array` présente un biais significatif et facilement observable avec les machines actuelles (Matsumoto et Nishimura, Sum discrepancy test on pseudorandom number generators 2003).

Ce manque de rigueur a inévitablement mené à la publication de résultats erronés. Ainsi les risques d'obtenir des résultats de simulations biaisés ou inutilisables liés à l'utilisation sans précaution d'un générateur de nombres pseudo-aléatoires ne sont pas que théoriques, ils sont bien réels. Au LIMOS (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes) un chercheur en informatique calcule de manière probabiliste la solution d'une EDP (Equations aux Dérivées Partielles) à l'aide de processus aléatoires, et en particulier d'un mouvement brownien selon la méthode présenté dans (Oksendal 2005). La méthode qu'il utilise nécessite la génération de deux variables aléatoires indépendantes et identiquement distribuées suivant une loi uniforme sur [0,1].

Il a simulé dans un premier temps le mouvement brownien avec le générateur de nombres pseudo-aléatoires implémenté dans la fonction rand de Linux qui est un générateur LCG de faible qualité statistique. Il a alors obtenu les résultats exposés à gauche de la Figure 7. Le résultat escompté converge vers la forme d'une demi-sphère. Ici, les corrélations internes et la trop faible période de la séquence de nombres générée

par la fonction `rand` biaisent fortement les résultats. Les réplications de la simulation du mouvement brownien ne sont pas indépendantes. On lui a alors conseillé de changer le générateur `rand` pour un générateur réputé pour ses bonnes propriétés statistiques : le générateur Mersenne Twister 19937 version 32 bits (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997). Les résultats qu'il a obtenus alors ne sont plus biaisés par les corrélations dans la séquence de nombres pseudo-aléatoires consommée par la simulation. Ils sont présentés à droite de la Figure 7.

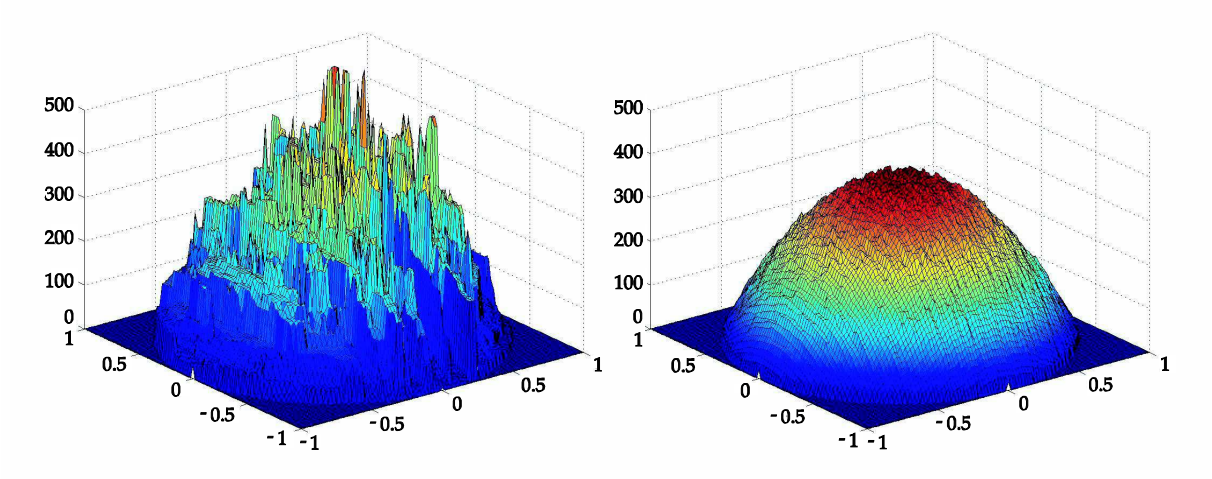


Figure 7 Résultats d'un calcul d'EDP probabiliste après 10000 réplications, à gauche en utilisant la méthode Rand de Linux et l'autre en utilisant l'algorithme Mersenne Twister 19937.

La difficulté de générer des séquences de nombres pseudo-aléatoires et la présence de générateurs présentant de larges défaillances dans des logiciels réputés peuvent mener à l'utilisation de mauvais générateurs pour des simulations stochastiques. De plus les mauvaises pratiques dans certains domaines mettent en doute la crédibilité des résultats obtenus par simulation stochastique. Ces défaillances sont problématiques pour des simulations stochastiques séquentielles. Elles sont cependant d'autant plus fréquentes, et ont un impact d'autant plus important sur des simulations stochastiques distribuées, comme le montre la partie III de ce chapitre.

III Les dangers liés aux simulations stochastiques distribuées

Dans le cadre, de simulations de type Monte-Carlo, un contexte expérimental valide comprend d'une part : des sources de hasard appropriées, et d'autre part : des méthodes correctes pour analyser les données. Nous avons déjà parlé de l'étude (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks

2002), elle porte sur 2400 publications dans le domaine des réseaux de télécommunication. Parmi les publications se rapportant à des résultats obtenus par l'intermédiaire de simulations stochastiques 77% négligent deux des aspects nécessaires à un contexte expérimental valide (Pawlikowski, Do Not Trust All Simulation Studies of Telecommunication Networks 2003). De telles négligences sont certainement déjà graves si on parle de simulations séquentielles, mais elles deviennent plus significatives si on fait référence à des simulations distribuées. Il est ainsi indispensable de se pencher sur la question de l'indépendance des flux de nombres pseudo-aléatoires.

Dans cette partie, on verra tout d'abord la technique de parallélisation utilisant l'aspect naturellement parallèle des simulations stochastiques : l'approche de distribution selon les réplications ou MRIP (Multiple Replication In Parallel). Cette technique est très efficace, et permet dans certains cas une réduction du temps global de simulation linéaire avec le nombre d'unités de calcul utilisées. Cependant MRIP nécessite la génération de flux de nombres pseudo-aléatoires non corrélés en parallèle. Le danger, si cet aspect est ignoré, est que la corrélation interprocessus entraine un biais dans les estimateurs statistiques. Ainsi, une attention toute particulière doit être portée sur la parallélisation des flux de nombres aléatoires, et des méthodes appropriées doivent être utilisées. La deuxième partie présente ce qu'on peut attendre d'un bon générateur parallèle de nombres pseudo-aléatoires. La troisième partie présente les techniques pour la génération en parallèle de nombres pseudo-aléatoires, et leurs faiblesses. Enfin, la quatrième partie présente un cas pratique ou les corrélations inter-séquences dues à une mauvaise parallélisation d'un générateur de nombres pseudo-aléatoires a mené à des résultats inutilisables.

III.1 L'indépendance des réplications

Comme exposé dans (Law et Kelton 1991), considérons Y_1, Y_2, … comme un processus stochastique de sortie issu d'une et une seule exécution d'une simulation. Considérons y_{11}, y_{12}, …, y_{1m} comme la réalisation des variables aléatoires Y_1, Y_2, …, Y_m résultant de l'exécution d'une simulation de m observations en utilisant les nombres aléatoires u_{11}, u_{12}, …. Si on exécute la simulation avec un ensemble différent de nombres aléatoires, u_{21}, u_{22}, … on obtient alors une réalisation différente y_{21}, y_{22}, …, y_{2m} des variables aléatoires Y_1, Y_2, …, Y_m. Supposons que l'on fasse n exécutions ou

« réplications » indépendantes de la simulation de longueur m (des nombres aléatoires différents sont utilisés et les compteurs statistiques sont remis à zéro au début de chaque réplication ; par contre les conditions initiales sont les mêmes), on obtient alors les observations suivantes :

$$\begin{array}{ccccc} y_{11} & \cdots & y_{1i} & \cdots & y_{1m} \\ y_{21} & \cdots & y_{2i} & \cdots & y_{2m} \\ \vdots & & \vdots & & \vdots \\ y_{n1} & \cdots & y_{ni} & \cdots & y_{nm} \end{array}$$

Les observations d'une même colonne sont des réalisations indépendantes et identiquement distribuées des variables aléatoires Y_1, Y_2, …, Y_m. Il donc possible d'effectuer une analyse simple et d'en déduire des inférences à propos des variables aléatoires Y_1, Y_2, …, Y_m. Par exemple, $\bar{y}_i = \sum_{j=1}^{n} \frac{y_{ij}}{n}$ est un estimateur non biaisé de $E(Y_i)$ (Law et Kelton 1991).

Les réplications sont des exécutions indépendantes d'une même simulation, de ce fait les simulations de Monte Carlo présentent un aspect naturellement parallèle ou « embarrassingly parallel » (Mascagni, Ceperley et Srinivasan, SPRNG: A Scalable Library for Pseudorandom Number Generation, 2000). L'approche de distribution par les réplications ou MRIP (Multiple Replications In Parallel) (Pawlikowski, Towards Credible and Fast Quantitative Stochastic Simulation 2003b) permet de paralléliser les simulations stochastiques selon cet aspect. Elle est basée sur le principe que le temps d'exécution d'une simulation stochastique est le temps nécessaire à la récolte de suffisamment d'échantillons résultat, ou observations, afin d'obtenir une erreur statistique convenable. Une simulation peut ainsi être parallélisée en calculant les différentes observations sur différents processeurs de manière indépendante. Un analyseur global calcule les propriétés statistiques d'un critère de performance de la simulation. La simulation s'arrête lorsqu'on obtient des résultats statistiques corrects pour l'ensemble des critères de performances, donc une erreur statistique suffisamment basse pour chacun des critères.

L'indépendance des séquences de nombres aléatoires utilisées par chaque réplication est nécessaire pour assurer la convergence de la simulation globale. Cette tâche n'est pas triviale et fait l'objet de la prochaine partie.

III.2 Qu'est-ce qu'un bon générateur parallèle de nombres pseudo-aléatoires ?

Certains seraient tentés de dire qu'avec un bon générateur parallèle de nombres pseudo-aléatoires « tu tires un nombre, il est aléatoire, et c'est un bon générateur », alors qu'avec un mauvais « tu tires un nombre, il est aléatoire, mais c'est un mauvais générateur… » Et ils ne seraient pas loin de la réalité !

Nous avons vu dans (P. D. Coddington, Random number generator for parallel computers 1996) les propriétés pratiques attendues pour un générateur de nombres pseudo-aléatoires séquentiel. Paul Coddington propose également cinq caractéristiques nécessaires à un bon générateur de nombre pseudo-aléatoires parallèle ou PPRNG (Parallel Pseudo Random Number Generator) :

1. Le générateur doit fonctionner pour un nombre quelconque de processeurs.
2. La séquence des nombres aléatoires générée sur chaque processeur doit satisfaire à toutes les caractéristiques d'un bon générateur séquentiel (par opposition à parallèle).
3. Il ne doit pas y avoir de corrélations entre les séquences utilisées par les différents processeurs.
4. La même séquence de nombres doit pouvoir être générée pour différents nombres de processeurs et pour le cas particulier d'un seul processeur.
5. L'algorithme doit être efficace, ce qui signifie qu'il ne doit pas y avoir de transmission de données entre les processeurs. Après que le générateur ait été initialisé, chaque processeur doit générer sa séquence indépendamment des autres.

Afin de satisfaire tous ces critères, la majorité des techniques de génération parallèle de nombres pseudo-aléatoires utilisent à l'heure actuelle des sous-séquences d'une séquence de nombres pseudo-aléatoires séquentielle. Cependant les corrélations longues-portées dans les générateurs pseudo-aléatoires séquentiels, qui ont la plupart du temps peu d'influence sur les résultats de simulation stochastiques, peuvent engendrer de fortes corrélations inter-séquences lors de la parallélisation de celui-ci. Les corrélations entre les séquences utilisées par les différentes réplications peuvent alors induire un biais dans les résultats de la simulation globale.

Le concept de corrélation longue-portée dans les générateurs de nombres pseudo-aléatoires a été introduit par DeMatteis, Pagnutti, Eichenauer-Herrmann et Grothe (Eichenauer-Herrmann et Grothe, A remark on long-range correlations in multiplicative congruential pseudo random number generators 1989) (De'Matteis et Pagnutti, Parallelisation of random number generators and long-range correlations 1990b) (DeMatteis et Pagnutti, A class of parallel random number generators 1990c) (DeMatteis, Eichenauer-Herrmann et Grothe, Computation of critical distances withing multiplicative congruential pseudorandom number sequences 1992). Les corrélations longues-portées sont définies comme des corrélations entre des blocs consécutifs de nombres dans une séquence de nombres pseudo-aléatoires. L'article (Entacher, Uhl et Wegenkitt, Parallel Random Number Generation: Long-Range Correlations Among Multiple Processors 1999) présente une étude des corrélations longues-portées avec différents générateurs de nombres pseudo-aléatoires en utilisant des points de la forme :

$$X_i = \left(x_i, x_{i+t}, x_{i+2t,} \dots , x_{i+(s-1)t}\right) \text{où } i \geq 0 \; s \geq 2$$

Les points ainsi générés, et un test de couverture spectrale, ont permis de tester quatre générateurs pour la présence de corrélations longues-portées :

1. Le générateur ranf ou LCG(2^{48},44485709377909,0,1), un générateur utilisé sur les systèmes CRAY.
2. Le générateur dran48 ou LCG(2^{48},25214903917,11,0) de la librairie standard du C.
3. Le générateur LCG(2^{48}-49,49235258628958,0,1) avec un modulo premier présenté dans (L'Ecuyer, A table of linear congruential generators of different sizes and good lattice structure 1999).
4. Un générateur inversif congruentiel explicite (pour la définition de ce type de générateur voir (Eichenauer-Herrmann, Statistical Independence of a New Class of Inversive Congruential Pseudorandom Numbers 1993)).

Alors que le test n'a pas permis de mettre en évidence des corrélations longues-portées dans les séquences générés par les générateurs 3 et 4 avec des tailles de blocs allant jusqu'à 2^{17}. Les deux premiers générateurs présentent de très fortes corrélations longues-portées avec des tailles de blocs supérieures à 2^{11}.

Cette étude montre qu'il n'est pas trivial de paralléliser un générateur de nombres pseudo-aléatoires. Dans le cas où les différentes séquences parallèles sont générées à partir d'une même séquence de nombres pseudo-aléatoires séquentielle une technique de parallélisation mal choisie peut mener à de fortes corrélations inter-séquences.

III.3 Génération de flux aléatoires parallèles

III.3.1 L'approche « serveur central »

L'approche la plus naturelle, est l'utilisation d'un serveur central qui exécute un algorithme de génération de nombres. Les différents processus de la simulation demandent leurs nombres pseudo-aléatoires à ce dernier. Il y a deux désavantages majeurs à utiliser cette méthode. Tout d'abord, le serveur central peut rapidement devenir un goulot d'étranglement pour la simulation. Le gain obtenu par la parallélisation est alors fortement réduit. D'autre part, la reproductibilité n'est pas garantie. Les nombres obtenus par chaque processus vont dépendre de l'ordre des demandes des nombres. Ce type de générateur parallèle viole ainsi deux des cinq recommandations pour un bon générateur de nombres pseudo-aléatoires parallèle établies par Coddington.

III.3.2 Le schéma de découpe par blocs ou « sequence splitting » ou « blocking » ou « boosting » ou « regular spacing »

Un générateur de nombres pseudo-aléatoires peut être représenté mentalement comme un cycle périodique de nombres (Mascagni et Srinivasan, Parameterizing parallel multiplicative lagged-Fibonacci generators 2004). Ce cycle peut être réparti en blocs de manière déterministe entre les différents processeurs. L'utilisateur choisit une taille des séquences (ou blocs), qui correspond à la taille d'une sous séquence dans le cycle global, et alloue chaque sous-séquence à un processeur différent. Le danger réside dans le risque que la simulation utilise plus de nombres que prévu et que les séquences se chevauchent. Dans ce cas la simulation peut ne pas converger ou produire des résultats statistiques présentant un biais.

Par exemple, le simulateur de réseau à événements discrets OMNeT++[7] propose un mécanisme de parallélisation des simulations à l'utilisateur. Pour ce faire, le générateur de nombres pseudo-aléatoires est parallélisé en utilisant un tableau de germes initiaux. Cependant les germes de ce tableau correspondent à des états du générateur de

[7] http://www.omnetpp.org

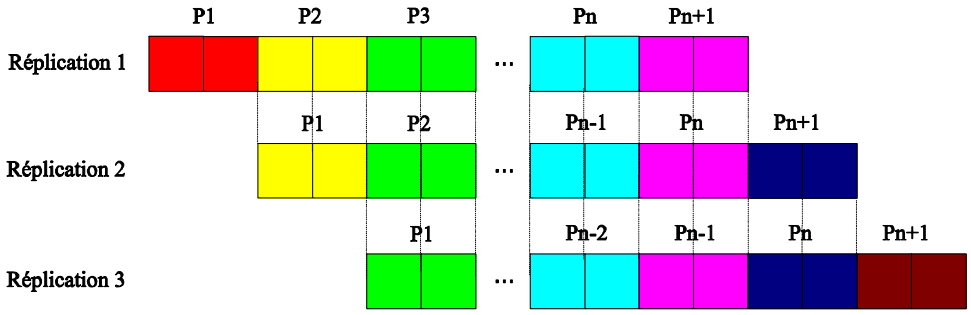
nombres pseudo-aléatoires espacés de seulement 1 millions de tirages. Ce faible espacement peut mener au chevauchement des séquences et à des résultats biaisés ou inexploitables (Hechenleitner, Defects in Random Number Routines of Well-Known Network Simulators and Appropriate Improvements 2004).

La Figure 8 et la Figure 9 présentent les résultats de deux calculs de π selon la méthode de Monte-Carlo. Afin de montrer l'erreur due à des corrélations entre les séquences de nombres lors d'une parallélisation par découpage en séquence, nous avons tracé pour chacun l'évolution de l'intervalle de confiance à 95% en fonction du nombre de réplications exécutés. Dans un cas, représenté par la courbe claire, les réplications utilisent des séquences de nombres faiblement corrélées. L'intervalle de confiance diminue et la moyenne des résultats des réplications converge vers π quand le nombre de réplications pris en compte augmente.

Sur la Figure 8, la courbe foncée représente les résultats obtenus alors que les séquences de nombres utilisées pour les 25 réplications sont fortement corrélées suivant le schéma présenté en haut de la figure. 999 points sur 1000 dans l'espace d'intégration à deux dimensions sont identiques d'une réplication à l'autre. Le calcul d'intervalle de confiance est biaisé. La moyenne des résultats des réplications oscille et ne tend pas vers π quand le nombre de réplications pris en compte dans les calculs statistiques augmente.

Selon les cas, il est possible que la simulation semble converger même avec des séquences de nombres hautement corrélées. Ainsi, la corrélation entre les réplications peut ne biaiser que certains indicateurs statistiques. Sur la Figure 9, la courbe en foncé représente les résultats d'une simulation qui utilise des séquences de nombres générés de manière à présenter des corrélations inter-séquences significatives selon le schéma en haut de la figure. Dans ce cas, la simulation utilise deux ensembles de points : une pour les réplications de numéros paires et une autre pour les réplications de numéro impaires. Ainsi la réplication numéro 25 partage 976 points avec la réplication numéro 1, et 977 avec la réplication 3. Elle n'a cependant aucun point en commun avec les réplications numéro 2 et 4. Ces deux cas (réplications paires | réplications impaires) peuvent être considérés comme deux réplications indépendantes. Elles produisent toutes deux un résultat dont la moyenne est proche de π. Le résultat de la simulation globale est très proche de la moyenne des réplications 1 et 2, donc de π. Même si la

moyenne des résultats semble correcte, les autres indicateurs statistiques (ici le calcul d'intervalle de confiance) sont biaisés. Il est par conséquent impossible de tirer des conclusions de ces simulation sans connaître à l'avance la valeur de π.



P = Paire : Une paire de nombres représente les coordonnées d'un point dans l'espace d'integration à 2 dimensions.
Les paires de la même couleur sont constituées de nombres de même valeurs.

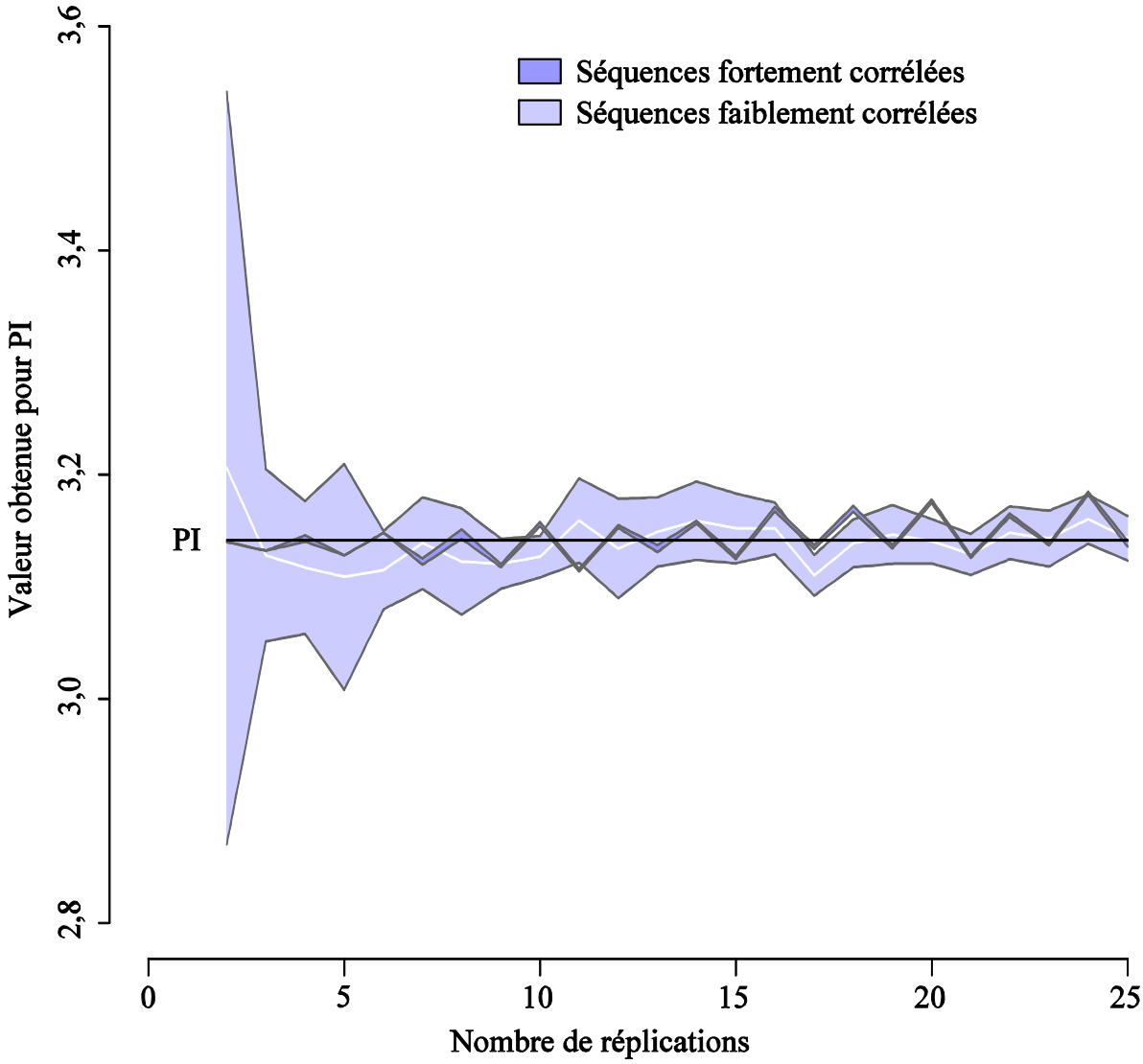


Figure 8 Convergence d'un calcul de PI par la méthode de Monté Carlo avec des réplications utilisant des séries de nombres pseudo-aléatoires indépendantes d'une part et fortement corrélées d'autre part

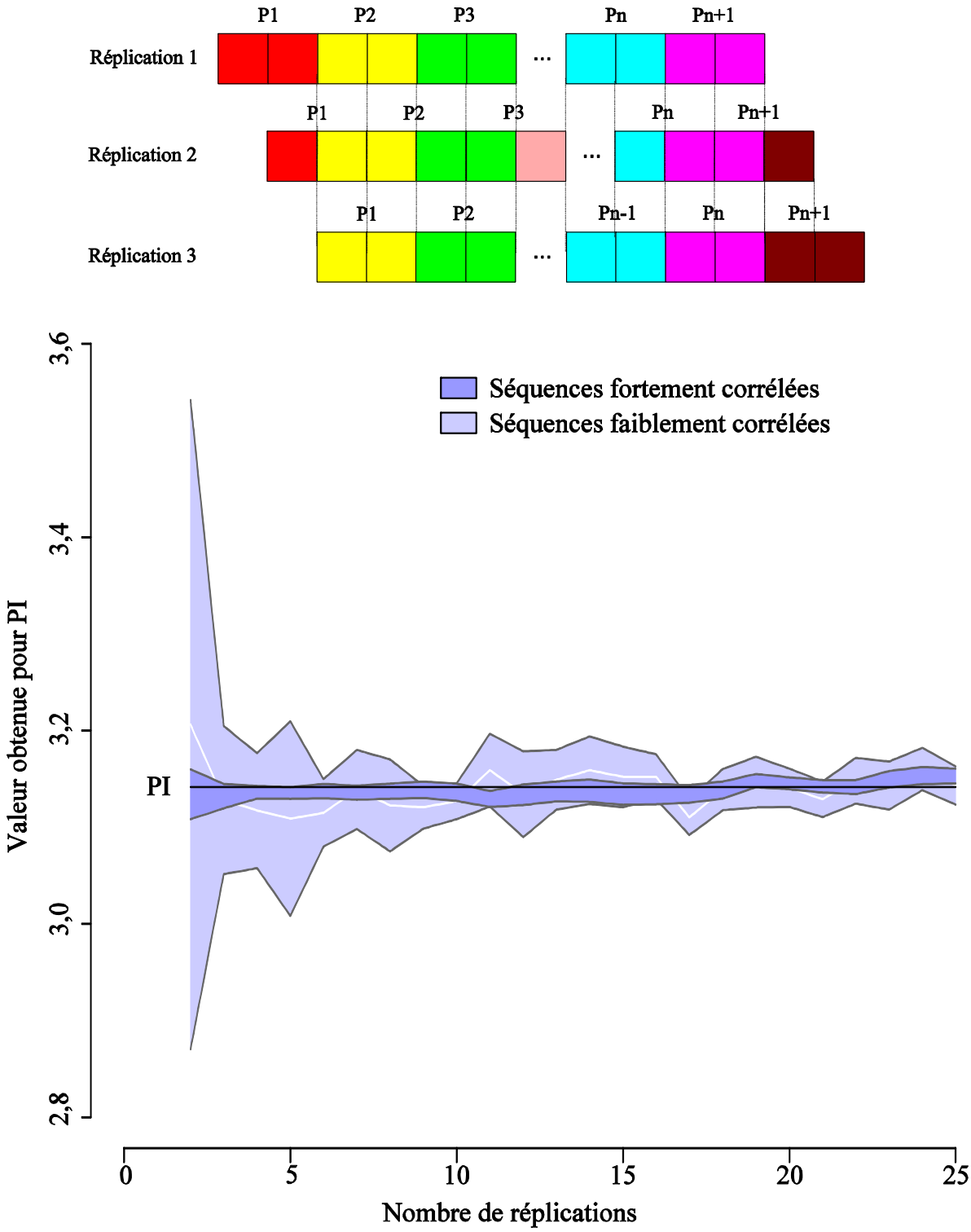


Figure 9 Convergence d'un calcul de PI par la méthode de Monté Carlo avec des réplications utilisant des séries de nombres pseudo-aléatoires indépendantes d'une part et fortement corrélées d'autre part

Un soin particulier doit être pris afin d'éviter les problèmes de chevauchements lors de la parallélisation d'un générateur de nombres pseudo-aléatoires par découpage en séquence. S'il est possible d'éviter les chevauchements, un autre problème est lui plus

difficile à résoudre : les corrélations longues-portées dans le générateur de nombres pseudo-aléatoires séquentiel (De'Matteis et Pagnutti, Long-range correlations in linear and non-linear random number generators 1990) (De'Matteis et Pagnutti, Parallelisation of random number generators and long-range correlations 1990b) se traduisent par des corrélations à faible portée entre les flux pour les générateurs aléatoires parallélisés par « sequence splitting ». C'est-à-dire que les séquences de nombres considérées comme indépendantes présenteront de fortes corrélations.

Les corrélations faibles-portées sont considérées comme plus dangereuses pour les applications classiques. L'article (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003) montre que le générateur LCG sur 48 bits, appelé sous Unix drand48, présente des corrélations de longue-portée qui deviennent des corrélations inter-séquences et de ce fait le rende impropre à être parallélisé par découpage en séquence ou « sequence splitting ».

Malgré les problèmes inhérents à cette méthode de parallélisation elle peut être utilisée sainement avec certains générateurs présentant de longues périodes. Dans ce cas, il est intéressant de coupler la parallélisation d'un générateur par découpage en séquence avec des techniques appelées « division de cycle ». Celles-ci permettent de calculer analytiquement l'état du générateur après plusieurs cycles de génération. Il est alors possible d'avancer le générateur de plusieurs itérations en une seule opération, sans générer les états intermédiaires. Pour que la technique soit efficace, le calcul de l'état après n pas de génération doit être beaucoup plus rapide que le calcul de chacun des n pas intermédiaires. Il est alors possible d'accélérer la création de blocs très « éloignés les uns des autres » dans le cycle de génération.

L'utilisation conjointe d'une technique de division de cycle et d'un découpage en séquence permet d'éradiquer facilement les problèmes de chevauchement. Un exemple est donné dans (Mascagni et Srinivasan, Parameterizing parallel multiplicative lagged-Fibonacci generators 2004) en utilisant l'algorithme de Miller and Brown. Cependant ces techniques de division de cycles sont impraticables pour des générateurs linéaires avec des périodes excessivement longues comme le Mersenne Twister (L'Ecuyer et Panneton, Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparison 2005). Une autre stratégie consiste alors à générer aléatoirement des états

pour le générateur de nombres pseudo-aléatoires cible à l'aide d'un autre générateur. C'est la technique dite d'« indexation de séquences ».

III.3.3 La technique d'indexation de séquences ou « Index sequence »

La technique dite des séquences indexées qu'on trouve aussi sous l'appellation « random spacing » consiste à générer des états pour un générateur de nombres pseudo-aléatoires avec un autre générateur de nombres pseudo-aléatoires. On considère alors que les états ainsi produits vont permettre de générer des sous-séquences indépendantes, et éloignées dans le cycle de génération du générateur parallélisé.

Cette technique est facilement exploitable avec certains générateurs comme les générateurs de Fibonacci avec décalage (Coddington et Newell, JAPARA – A Java Parallel Random Number Library for High-Performance Computing 2004) pour lesquels il suffit de remplir la table d'historique des nombres avec un autre générateur de nombres pseudo-aléatoires. L'article (Wu et Huang 2006) montre que la distance minimale, entre *n* statuts générés de la sorte, est de $1 / n^2$ fois la période du générateur cible en moyenne. Cette technique est donc applicable, sans risque de chevauchement lors de l'utilisation des séquences, pour des générateurs avec de très grandes périodes (par exemple le Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) avec une période de 2^{19937}-1).

III.3.4 La technique du saut de grenouille ou « Leap frog »

Cette technique consiste à distribuer les nombres comme on distribue des cartes à des joueurs. Chaque processeur utilise alors un nombre sur *n* dans la série originale. En utilisant cette technique les corrélations longues-portées d'un générateur de nombres pseudo-aléatoires peuvent cependant devenir des corrélations faibles-portées dans les séquences générées quand le nombre de processeurs est grand. De plus, des cas de forte corrélation entre les séquences (corrélations croisées) sont observés quand l'intervalle utilisé entre deux nombres de la série originelle est mal choisi (Wu et Huang 2006). Enfin, le cas d'un générateur dont la qualité est largement réduite par la parallélisation par « leap frog » est présenté dans (Hellekalek, Don't trust parallel Monte Carlo 1998).

Pour une question de performance, cette technique est généralement couplée à des techniques de division de cycles (Srinivasan, Ceperley et Mascagni, Random Number Generators for Parallel Applications 1999). En effet, sans le couplage avec une technique

de division de cycle le générateur de nombres pseudo-aléatoires parallélisé par saut de grenouille est *n* fois moins performant que sa version séquentielle, où *n* est le nombre de sous-séquences générées. Elle n'est donc pas exploitable avec la majorité des générateurs de nombres pseudo-aléatoires.

III.3.5 Paramétrisation du générateur

La paramétrisation des cycles pour un générateur, permet d'obtenir des séries de nombres différents, qui ne se recouvrent jamais. Ainsi chaque processeur dispose de son propre cycle de nombres pseudo-aléatoires.

Pour la mise en œuvre de cette technique, on doit disposer d'un générateur sous forme d'une fonction itérative paramétrable, qui, étant donné un jeu de paramètres, générera des séries différentes et indépendantes. Cette technique de parallélisation n'est pas applicable à tous les générateurs de nombres pseudo-aléatoires séquentiels, du fait qu'une étude théorique est nécessaire afin de pouvoir générer des paramètres qui mènent à la génération de séquences non corrélées.

Par exemple, les générateurs de type Mersenne Twister (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) ont fait l'objet d'une telle étude (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000). On dispose maintenant d'un algorithme qui permet de générer des paramètres pour le Mersenne Twister. On peut donc obtenir de nombreux Mersenne Twisters différents qui génèrent tous en théorie des séquences hautement indépendantes les unes des autres. L'algorithme actuellement disponible permet de générer jusqu'à 65535 générateurs indépendants.

Une étude similaire a été menée pour les générateurs linéaires congruentiels (Mascagni et Chi, Parallel linear congruential generators with Sophie-Germain moduli 2004). Elle montre comment il est possible de générer des LCG indépendants où les modulos sont : ou des nombres premiers de Mersenne, ou des nombres premiers de Sophie-Germain. Cette étude montre de plus que le nombre de générateurs pseudo-aléatoires indépendants qu'il est possible de générer avec des nombres de Sophie-Germain est bien plus important que celui qu'il est possible de générer avec l'utilisation des nombres premiers de Mersenne pour un intervalle donné. Le temps de génération des paramètres est ainsi grandement réduit. Cette technique de parallélisation par

paramétrisation est implémentée dans la librairie SPRNG (Mascagni, Ceperley et Srinivasan, SPRNG: A Scalable Library for Pseudorandom Number Generation, 2000).

Même si les bases théoriques de cette technique de parallélisation sont solides dans l'état actuel de nos connaissances, il est impossible de vérifier l'indépendance des séquences dans la pratique. Ce problème est transversal à toutes les techniques de parallélisation de générateur de nombres pseudo-aléatoires. Par exemple, pour les Mersenne Twister, les séquences les plus courtes comprennent 2^{521} nombres pseudo-aléatoires. Tester empiriquement leur indépendance deux à deux est impossible aux vues des périodes mises en jeu. En testant une infime partie du cycle, comme pour les générateurs séquentiels, la somme de calcul nécessaire pour de tels tests reste trop importante pour les ordinateurs séquentiels actuels du fait de l'explosion combinatoire du nombre de tests avec l'augmentation du nombre de séquences à tester.

III.3.6 Automates cellulaires

Les automates cellulaires sont des systèmes dynamiques dans lesquelles le temps et l'espaces sont discrets. Une solution pour la génération de séquences de nombres pseudo-aléatoires parallèles utilisant des automates cellulaires booléens à n dimensions (ou chaque cellule contient 0 ou 1) est présentée dans (Sipper 1996) et dans (Tomassini, et al. 1999).

Dans ces articles, seuls des automates à 1 dimension sont considérés. Chaque cellule contient 1 bit utilisable indépendamment des autres ; l'automate génère ainsi n flux de bits aléatoires. Les auteurs testent les propriétés statistiques et mesurent les corrélations croisées des séquences générées pour différentes règles de transitions et différentes règles de prélèvement des nombres (utilisation d'espacements temporels en prélevant un bit tout les m cycles de l'automate). Ils en concluent que les générateurs aléatoires basés sur des automates cellulaires peuvent être utilisés pour des simulations en environnement distribué. Cette méthode de génération utilise des algorithmes assez lents. Son intérêt principal est donc de pouvoir être implémentée facilement et à moindre frais de façon matérielle, par exemple en utilisant des puces programmables (Ackermann 2001).

III.4 Les corrélations inter-séquences en pratique

III.4.1 Le simulateur NS2

L'absence d'étude préalable, l'utilisation de mauvaises pratiques ou de techniques inappropriées sont autant de pièges qui peuvent mener à la génération de séquences de nombres fortement corrélées qui biaisent les résultats d'une simulation stochastique distribuée.

Le simulateur open source à élément discret pour les réseaux NS2[8] est développé depuis 1989. Ce projet populaire a été financé par la DARPA (Defense Advanced Research Projects Agency). Jusqu'en 2001, les simulations NS2 étaient basées sur l'utilisation du générateur de nombres pseudo-aléatoires ran0 aussi appelé « minimal standard » conçu en 1969 pour le système IBM System/360 (Lewis, Goodman et Miller 1969).

Comme établis dans (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002), ce générateur est obsolète et présente plusieurs défauts majeurs. Tout d'abord, il présente une période de 2^{31}-2, ce qui est très insuffisant au regard des applications actuelles.

Malgré sa faible période, le plus gros défaut du générateur de nombres pseudo-aléatoires de NS2 est lié à une exécution distribuée des simulations NS2 en utilisant le mécanisme d'initialisation proposé dans l'application. Toujours dans l'article (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002), les auteurs citent un extrait des commentaires du code source du logiciel reproduit dans le Code 3.

[8] http://www.isi.edu/nsnam/ns/

```
// NEEDSWORK: should be a way to set seed to PRDEF_SEED_SOURCE
    .
    .

// NEEDSWORK: should we throw out known bad seeds?
// (are there any?)
    .
    .
// Toss away the first few values of heuristic seed.
// In practice this makes sequential heuristic seeds
// generate different first values.
// How many values to throw away should be the subject
// of careful analysis.  Until then, I just throw away
// ``a bunch''.  --johnh
```

Code 3 Extrait de (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002): extrait des commentaires du code source de NS2 sur l'initialisation du générateur de nombres pseudo-aléatoires.

Ces commentaires montrent clairement qu'aucune analyse n'a été mise en œuvre avant de proposer cette fonctionnalité aux utilisateurs du logiciel. La question « *Are there any? (bad seeds)* » est tranchée dans la suite de l'article ; et la réponse est très clairement positive !

Pour exemple, les trois séquences de nombres générées par le générateur de nombres pseudo-aléatoires de NS2 en utilisant les germes : 1,2 et 3, sont fortement corrélées. L'utilisation de ces séquences de nombres comme coordonnées pour des points dans des espaces à 2 dimensions et à 3 dimensions, révèle de fortes corrélations linéaires, comme le montre la Figure 10.

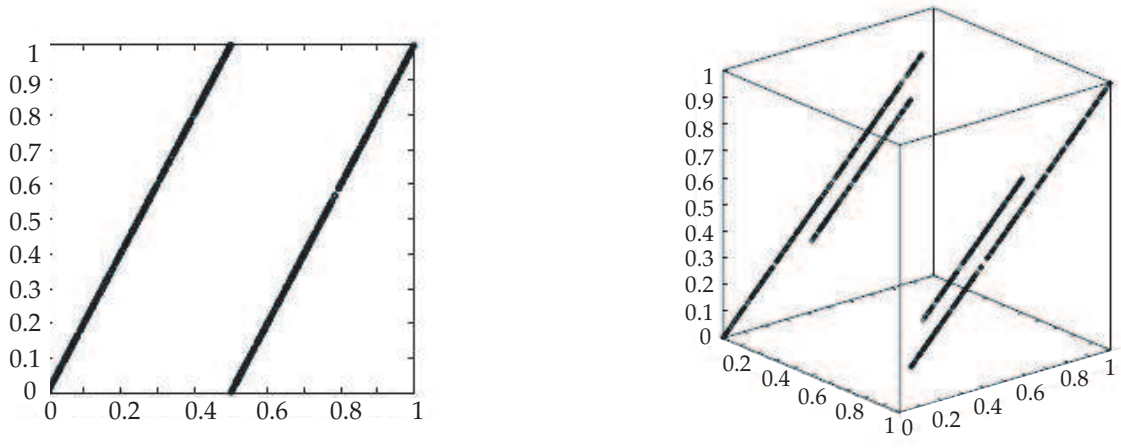


Figure 10 Extrait de (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002) : Illustration de la corrélation entre les séquences générées en utilisant le générateur de nombres pseudo-aléatoires de NS2 initialisé avec les germes 1,2 et3.

La suite de (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002) montre que ces germes ne sont pas les seuls à générer des séquences fortement corrélées, et que par conséquent ce générateur et la méthode de parallélisation qui lui est associée dans NS2, sont impropres à être utilisés pour exécuter des simulations distribuées. Comme le montre le Tableau 1, l'utilisation de NS2 pour une exécution distribuée mène à la production de résultats de simulation biaisés.

Tableau 1 Extrait de (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002) : Comparaison des résultats de simulations distribuées avec NS2 en utilisant le générateur d'origine initialisé correctement ou pas et en utilisant le générateur Mersenne Twister 19937

Simulation	RNG seed set 1 ('good' seeds)	RNG seed set 2 ('bad' seeds)	RNG seed set 3 ('good' seeds)	RNG seed set 4 ('bad' seeds)
seed 1	1973272912	1	934100682	1
seed 2	1822174485	2	558746720	634005912
seed 3	1998078925	3	2081634991	634005911
seed 4	678622600	4	144443207	2147483646
seed 5	999157082	5	513791680	151347773
$\overline{q}$	19.451	24.288	19.374	17.573
Simulation	MT RNG seed set 1	MT RNG seed set 2	MT RNG seed set 3	MT RNG seed set 4
seed 1	1973272912	1	934100682	1
seed 2	1822174485	2	558746720	634005912
seed 3	1998078925	3	2081634991	634005911
seed 4	678622600	4	144443207	2147483646
seed 5	999157082	5	513791680	151347773
$\overline{q}$	20.259	21.227	20.120	19.803

Sans rentrer dans le détail de la simulation étudiée, les simulations NS2 distribuées avec 5 processus parallèles utilisant les germes 1, 2, 3, 4, 5 ou 1, 634005912, 634005911, 2147483646, 151347773 produisent des résultats éloignés des résultats obtenus avec l'utilisation de « bons » germes ou avec l'utilisation du Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) et du résultat théorique $\bar{q} = 20{,}488$.

III.4.2 OMNeT++

OMNeT++ (Varga 2001) est un simulateur à événements discrets pour la simulation de réseaux d'ordinateurs et de systèmes distribués. Son implémentation a commencé en 1992 à l'université de Budapest. Ce logiciel est disponible ainsi que son code source[9]. Il est utilisé par de nombreuses universités et industriels ; parmi eux : Siemens et Lucent.

ONMeT++ utilise le même algorithme de génération de nombres pseudo-aléatoires que NS2 et par conséquent souffre des mêmes faiblesses quand à son fonctionnement

[9] http://www.omnetpp.org/

pour des simulations stochastiques distribuées. Dans (Entacher et Hechenleitner, Pitfalls when using parallel streams in OMNET++ simulations 2003) les auteurs étudient l'impact de des corrélations inter-séquences de ce générateur de nombres pseudo-aléatoires sur des résultats d'une simulation stochastique distribuée.

Tableau 2 Extrait de (Entacher et Hechenleitner, Pitfalls when using parallel streams in OMNET++ simulations 2003) : Résultats d'une simulation distribuée avec OMNeT++ en utilisant le générateur de nombres pseudo-aléatoires d'origine et de mauvaises initialisations

RNG Object	Seed Set 1	Seed Set 2
Expo 1	1	1
Expo 2	2	634005912
Expo 3	3	634005911
Expo 4	4	2147483646
Expo 5	5	1513477735
FIFO	6	1513477736
$\overline{N}$	43.3752186	38.9098305

Les résultats exposés dans le Tableau 2 sont éloignés du résultat théorique : $\bar{q} = 40{,}0004$. Les auteurs remplacent ensuite le générateur d'origine, ran0, par les générateurs : RandU01 implémenté par L'Ecuyer et al. (L'Ecuyer, Simard et Chen, et al. 2002) et Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997). Les résultats obtenus alors sont significativement plus proches de la moyenne théorique. Il semble donc que l'utilisation dans OMNET++ d'un générateur de nombres pseudo-aléatoires inadapté pour une exécution distribuée biaise les résultats de certaines simulations.

III.4.3 Impact des corrélations inter-séquences sur des résultats de simulation

L'utilisation de séquence de nombres pseudo-aléatoires inter-corrélés dans une simulation distribuée peut avoir des effets très négatifs sur la qualité des résultats de la simulation. Dans cette partie nous exposons une partie des résultats d'une étude, présentée plus loin, dans laquelle nous avons mis en évidence l'impact des corrélations inter-séquences dans le contexte de la reconstruction d'images tomographiques avec le logiciel GATE (Jan et al. 2004)

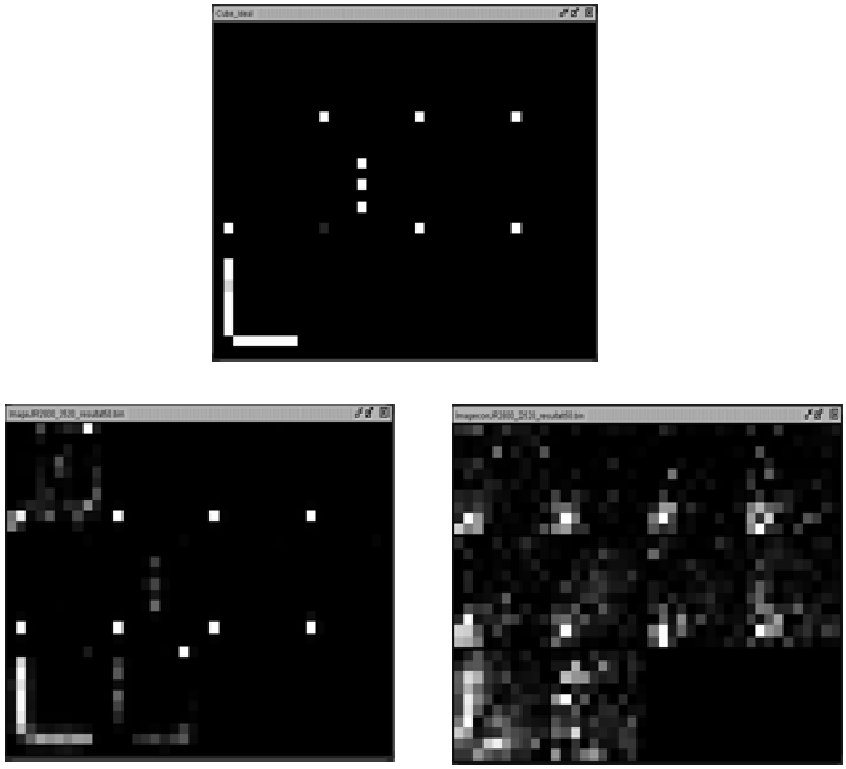


Figure 11 Effet de la corrélation inter-séquences dans un générateur parallèle de nombres pseudo-aléatoires sur les résultats d'une simulation de Monté Carlo distribuée. L'image en haut présente le résultat idéal, celle en bas à gauche est un résultat de simulation distribuée utilisant des séquences non-corrélées et celle en bas à droite utilisant des séquences de nombres aléatoires présentant des corrélations inter-séquences significatifs.

Parmi les trois images de la Figure 11, l'image du haut est le résultat escompté ou idéal. L'image en bas à gauche est obtenue après une reconstruction en trois dimensions basée sur les résultats d'une simulation de Monte-Carlo distribuée. Dans ce cas, les processus de la simulation distribuée ont utilisé des séquences de nombres faiblement corrélées en elles. L'image en bas à droite a été obtenue avec le même algorithme de reconstruction et le même générateur de nombres pseudo-aléatoires avec peu de corrélations intra-séquences. Cependant, lors de la parallélisation par découpage en séquence du générateur, de forte corrélations inter-séquences ont été volontairement introduites par chevauchement de séquences. Ces corrélations rendent cette image très bruitée et inutilisable dans un cadre scientifique.

La distribution rigoureuse des flux de nombres pseudo-aléatoires en vue de la parallélisation d'une simulation de Monté Carlo est une tâche difficile, qui nécessite un effort cérébral non négligeable. Il nous est arrivé de faire des erreurs au cours du processus de distribution d'une simulation de Monte-Carlo, menant à l'utilisation de séquences corrélées entre elles par les différentes réplications.

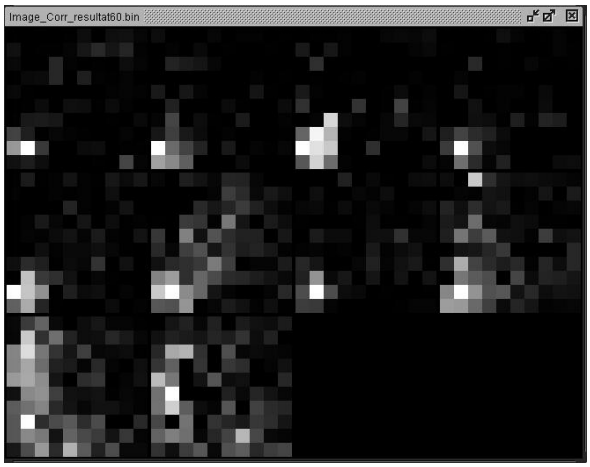


Figure 12 Résultat d'une simulation de Monté Carlo distribuée basée sur l'utilisation involontaire de séquences de nombres pseudo-aléatoires corrélées entre elles.

L'image de la Figure 12 présente ainsi un autre résultat issu d'un processus de simulation de reconstruction d'images tomographiques basée sur l'utilisation de séquences involontairement corrélées entre elles à la suite d'une erreur humaine. Lors de la génération de cette image par simulation, environ un tiers des séquences consommées par la simulation distribuée présentaient des corrélations, conduisant à la génération d'environ 1000 résultats non indépendants parmi des 3000 réplications exécutées en parallèle. En définitive, le résultat est inutilisable pour un usage scientifique.

IV Conclusion

La simulation stochastique séquentielle présente différents écueils liés à l'utilisation de processus aléatoires. La génération de nombres pseudo-aléatoires est une tâche difficile. La notion de hasard est difficile à appréhender, et les outils mathématiques actuels ne nous permettent pas d'avoir de garantie sur le caractère aléatoire d'une séquence de nombres. Ainsi la méthode la plus efficace à l'heure actuelle pour détecter des régularités dans les suites de nombres générées par un générateur de nombres pseudo-aléatoires est de soumettre une partie de son cycle de génération (généralement infime) à des cas de simulations dont on sait qu'ils sont sensibles à certaine structures dans les suites générées. Même si ces tests empiriques ne détectent pas de défauts dans les séquences de nombres générées, ils ne sont en aucun cas la garantie que la séquence est utilisable pour un autre cas de simulation. Le meilleur test à l'heure actuelle pour voir l'impact d'un générateur de nombres pseudo-aléatoires sur les résultats d'une

simulation, est d'exécuter cette simulation avec différents générateurs et de comparer les résultats.

Comme nous l'avons vu dans cette partie, de très mauvais générateurs de nombres pseudo-aléatoires et des générateurs obsolètes sont couramment distribués avec des logiciels récents et de renom. De plus même les bons générateurs de nombres pseudo-aléatoires présentent des défauts. Les mauvaises pratiques de non-spécialistes peuvent alors mener à la production de résultats de simulation inutilisables ou à la publication de résultats biaisés.

Pour les simulations stochastiques distribuées la situation est encore plus critique. En plus des problèmes de régularités dans les séquences de nombres issues de la simulation stochastique séquentielle, les corrélations inter-séquences représentent un problème majeur lié à la génération parallèle de nombres pseudo-aléatoires. Tout comme pour les générateurs de nombres pseudo-aléatoires séquentiels, il n'est pas possible de montrer en pratique l'absence de corrélation inter-séquence. La somme de calcul que représentent ces tests les met hors de portée d'une réalisation pratique à grande échelle à l'heure actuelle. Les techniques de parallélisation des générateurs pseudo-aléatoires séquentiels peuvent mener dans certains cas à de fortes corrélations entre les séquences utilisées par les différentes réplications d'une simulation stochastique distribuée selon la méthode MRIP et mener à des résultats erronés. Les corrélations longues-portées dans la séquence de nombres pseudo-aléatoires parallélisés deviennent alors des corrélations inter-séquence courte-portée entre les flux qui sont généralement dommageables pour les résultats de simulation. Enfin la gestion de milliers de flux de nombres pseudo-aléatoires est une tâche délicate en pratique et peut mener à des erreurs humaines.

Des outils et des méthodes existent afin d'éviter les écueils liés à l'exécution de simulations stochastiques en environnements de calcul distribué. Ces outils sont présentés dans le chapitre 2.

Chapitre 2 : L'existant

I Introduction

Le chapitre 1 met en avant les difficultés que peut rencontrer un modélisateur lors de la distribution de ses calculs stochastiques. Même si des précautions sont à prendre pour la distribution des simulations de type Monte-Carlo suivant l'approche MRIP, la plupart des environnements de calcul distribué sont très adaptés pour l'accélération des calculs stochastiques. Des techniques d'optimisation spécifiques ont même été développées comme les techniques N out of M (Li et Mascagni 2003) ou d'ordonnancement dynamique avec historique (Mazumdar, Mathew et Leathrum 2004). Dans le meilleur des cas, on peut espérer un gain en temps de calcul croissant linéairement avec le nombre de processeurs utilisés. L'utilisation d'environnements de calcul extensifs, type « grille de calcul », ou de type « internet computing », permet alors d'exécuter des simulations impraticables en utilisant des calculateurs séquentiels. Ce chapitre décrit ainsi les solutions existantes pour l'exploitation de la puissance de calcul de processeurs distribués pour la simulation.

Il présente tout d'abord les architectures extensives de calcul distribué. Celles-ci visent à fédérer la puissance de calcul de nombreux processeurs reliés par des réseaux

étendus au delà même d'un domaine administré, et contrôlé par un seul individu ou une seule institution.

Les temps d'exécution et la consommation en mémoire vive de certaines simulations peuvent excéder les capacités d'un seul ordinateur séquentiel. Afin d'exploiter la puissance des architectures de calcul distribué, des solutions existent. Elles permettent de distribuer l'exécution de simulations déterministes ou de simulations stochastiques pour lesquelles on ne cherche à exécuter qu'un nombre réduit de réplications. Un travail de parallélisation du code (et du générateur de nombres pseudo-aléatoires pour les simulations stochastiques) doit alors être entrepris. Les solutions afin de paralléliser les simulations déterministes sont présentées dans la deuxième partie de ce chapitre.

Même si la distribution avec la méthode MRIP (Multiple Replications In Parallel) de simulations stochastiques comprenant de nombreuses réplications représente une application phare pour l'utilisation d'architectures qui peuvent fédérer la puissance de centaines de milliers de processeurs, la disponibilité d'outils conçus spécifiquement pour la distribution de telles applications sont peu nombreux. La troisième partie présente ainsi les outils actuellement disponibles permettant la distribution rigoureuse de simulations stochastiques selon les réplications.

II Les environnements extensifs pour le calcul distribué

Les simulations de Monté-Carlo travaillent généralement sur une quantité réduite de données mais consomment de nombreux cycles processeurs (Li et Mascagni 2003). Elles sont donc très adaptées aux environnements de calcul distribué tels que les architectures multiprocesseurs ou les fermes de calcul, mais elles représentent surtout des applications phares pour l'exploitation de la puissance des architectures de calcul distribué type calcul pair à pair et de grille de calcul. Par exemple, le projet par QMC@home (Quantum Monte Carlo at home) utilise une méthode de Monté Carlo en chimie quantique pour calculer la réactivité et la structure de molécules. Il utilise à ce jour la puissance combinée de 62663 ordinateurs distribués sur l'Internet pour exécuter leurs simulations.

Dans cette partie nous présentons deux architectures de calcul distribué extensives : d'une part les architectures de calcul pair à pair et d'autre part les grilles de calcul. Enfin

je présente les principaux outils développés pour faciliter l'exécution en environnement distribué.

II.1 Architecture de calcul pair à pair

II.1.1 Historique

Il est envisageable, depuis une quinzaine d'années d'agréger la puissance de calcul de processeurs communicant entre eux par un réseau ou un bus. Selon les besoins, les architectures de calcul distribué se basent sur des paradigmes très différents. Les logiciels architecturés selon une approche pair à pair sont généralement conçus pour le partage de fichiers, le travail collaboratif ou le calcul distribué.

Depuis une vingtaine d'années, on note que le processeur central de nombreux ordinateurs séquentiels est largement sous exploité. Ce fait est constaté dans de nombreuses publications : (Mutka et Livny 1991), (Livny 1999), (Ryu et Hollingsworth 2000) et (Anderson et Kubiatowicz, The worldwild computer 2002). Une solution pour exploiter ces ressources est le calcul distribué pair à pair. Il permet le partage des cycles processeurs inutilisés des stations de travail pour résoudre des problèmes scientifiques.

Historiquement le concept de vol de cycle à grande échelle, ou « global computing » (Cappello, et al. 2005), a émergé afin d'étendre la notion de vol de cycles, déjà présente dans les fermes de calcul, au delà des frontières d'un domaine administré. Le premier article discutant du concept de vol de cycles (Shoch et Hupp 1982) présente le programme « Worm ». Plusieurs idées clef y sont développées comme : l'autoréplication, la migration, ainsi que la coordination distribuée. Le vol de cycle est dés lors étudié dans de nombreux projets tels que MOSIX (Barak et Litman, MOS: a Multicomputer Distributed Operating System 1985) (Barak et Wheeler, MOSIX: An Integrated Multiprocessor UNIX 1989), Condor (Litzkow, Livny et Mutka, Condor - A Hunter of Idle Workstations 1988) (Litzkow et Solomon, Supporting Checkpointing and Process Migration outside the UNIX Kernel 1992) ou Glunix (Ghormley, et al. 1998). Ils permettent tous de distribuer des calculs sur une communauté de machines mettant en œuvre des mécanismes de migration et de délégation de tâches de machine en machine. Ces concepts sont remaniés lors des premières tentatives dans le domaine de l' « internet computing ». Elles sont menées par les projets : ParaWeb (Brecht, et al. 1996), Jet (Pedroso, Silva et Sil 1997), SuperWeb (Alexandrov, et al. 1997), Javeline (Cappello, et al. 1997), PopCorn (Nisan, et al. 1998), Bayanihan (Sarmenta, Hirano et

Ward 1998) et Charlotte (Baratloo, et al. 1999). Tous ces projets explorent de nouveaux aspects en termes de sécurité, d'extensibilité, de tolérance aux fautes, de vérification des résultats et de marché du temps de calcul.

L'avènement du calcul extensif de type « internet computing » est symbolisé par l'initiative SETI@home (Kopela, E. et al. 2001). Ce projet de recherche d'intelligence extraterrestre regroupe pendant plusieurs années une puissance d'environ 60 téraflops (Anderson, D.P. et al. 2002). Elle sert alors de base pour le développement de BOINC (Berkeley Infrastructure for Network Computing) (Anderson, D.P. et al. 2002) (Anderson, BOINC: A System for Public-Resource Computing and Storage 2004), la principale plateforme à l'heure actuelle pour l'« internet computing ».

De nos jours, de nombreux projets font participer des volontaires pour des calculs scientifiques, parmi eux : Climateprediction.net (Stainforth, D.A. et al. 2005), LHC@home (Herr, McIntosh et Schmidt 2006), ou l'initiative « World Comunauty Grid », lancé par IBM, qui comprend entre autres les projets FightAIDS@Home (Chang, et al. 2007) et AfricanClimate@Home.

II.1.2 Le concept d'architecture pair à pair

Dans (Caromel, di Costanzo et Mathieu, Peer-to-peer for computational grids: mixing clusters and desktop machines 2007), on trouve de multiples définitions du concept d'architecture réseau pair à pair : organisation décentralisée et non-hiérarchique de nœuds (Stoica, et al. 2003), utilisation de machines à la périphérie d'internet (Oram 2001). La définition la plus précise se trouve dans (Schollmeier 2001). Une catégorisation des architectures logicielles réseau y est établie à partir de quatre définitions :

- Architecture pair à pair : « *A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P, …) network, if the participants share a part of their own hardware resources (processing power, storage capacity, network link capacity, printers, …). These shared resources are necessary to provide the Service and content offered by the network (e.g. file sharing or shared workspaces for collaboration). They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requestors (Servent-concept).* »

- Architecture pair à pair pure : « *A distributed network architecture has to be classified as a Pure Peer-to-Peer network, if it is firstly a Peer-to-Peer network according to Definition 1 and secondly if any single, arbitrary chosen Terminal Entity can be removed from the network without having the network suffering any loss of network service.* »
- Architecture pair à pair hybride : « *A distributed network architecture has to be classified as a Hybrid Peer-to-Peer network, if it is firstly a Peer-to-Peer network according to Definition 1 and secondly a central entity is necessary to provide parts of the offered network services.* » C'est par exemple le cas du célèbre réseau Napster[10].
- Architecture client/serveur : « *A Client/Server network is a distributed network which consists of one higher performance system, the Server, and several mostly lower performance systems, the Clients. The Server is the central registering unit as well as the only provider of content and service. A Client only requests content or the execution of services, without sharing any of its own resources.* »

Il arrive fréquemment de ne pas classifier la plateforme BOINC comme un système pair à pair (Milojicic, et al. 2002) : l'argument est que pour cette plateforme un serveur central est nécessaire pour contrôler les ordinateurs connectés et qu'il n'y a pas de communications entre les pairs. D'après la classification de (Schollmeier 2001), il serait alors un réseau pair à pair hybride (à la Napster). L'aspect technique réel éloignant la plateforme BOINC du concept d'architecture de pair à pair est en fait que les ressources partagées ne sont pas directement accessibles par les ordinateurs participant au calcul. Cependant, si on considère une définition plus large du terme pair à pair : les ordinateurs participant fonctionnent avec une haute autonomie et le service fourni est réalisé en utilisant une architecture décentralisée et non-hiérarchisée d'ordinateurs à la périphérie d'Internet.

Ce type de plateforme de calcul est ainsi appelé « internet computing », « global computing » (Fedak, G. et al. 2001), « métacomputing » (Baratloo, et al. 1999) ou encore « volunteer grid » (Anderson et Fedak, The Computational and Storage Potential of Volunteer Computing 2006) par opposition à un autre type de calcul pair à pair, le calcul

[10] http://free.napster.com

parasitaire pour lequel les ordinateurs sont exploités et participent à un calcul distribué, sans approbation de leur propriétaire (Barabasi, A. et al. 2001). A l'heure actuelle les plus gros projets de calcul distribué comme « folding@home » utilisent aussi bien la puissance des processeurs des ordinateurs personnels, que celle de leurs cartes graphiques, ou encore celle des consoles de jeux vidéo comme la Playstation 3 et la Xbox 360 et même celle des cerveaux humains avec des projets comme « Google Image Labeler »[11] pour l'annotation d'image et « Foldit »[12] pour trouver les configurations fonctionnelles de protéines.

II.2 La grille de calcul

II.2.1 La notion de ferme de calcul

En 1994, le projet Beowulf de la NASA (Sterling, et al. 1995) montre qu'il est possible d'utiliser un groupe de machines standards pour du calcul haute performance. En 1996, il regroupe 16 ordinateurs personnels de type 486DX4 à 100MHz avec chacun deux interfaces Ethernet à 10 mégabits par seconde et des disques durs de 250 Mo. Le projet mobilise ainsi 1,25 giga-flops pour une simulation informatique. Une puissance de calcul conséquente pour l'époque obtenue pour moins de 50 000 dollars.

A cette même époque, au LIMOS (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes), une simulation est distribuée sur 10 ordinateurs personnels de type 486 à 66 MHz avec chacun 16 Mo de RAM (Random Access Memory) en utilisant la bibliothèque de passage de messages PVM (Parallel Virtual Machine) (Sunderam 1990). La distribution de la simulation permit d'obtenir des temps de calcul jusqu'à 1,6 fois plus courts qu'avec un supercalculateur de l'époque de type IBM Power 2 avec 256 Mo de RAM (Hill, A Design Pattern for Object-Oriented Distributed Simulation of Large Scale Ecosystems 1996) .

Un ensemble d'unités de calculs regroupées physiquement en un même lieu et coordonnées en vue de profiter d'une plus grande puissance de calcul ou de stockage est appelé grappe de calcul, ferme de calcul ou « cluster ». Il existe des fermes de calcul de différentes tailles et basées sur des architectures diverses. L'augmentation de la puissance de calcul d'une ferme se fait facilement et linéairement en ajoutant de nouvelles unités de calcul ou de stockage. Cependant celle-ci est limitée par les besoins

[11] http://images.google.com/imagelabeler
[12] http://fold.it/portal/adobe_main

en bande passante réseau pour les communications entre les machines. Aujourd'hui la ferme de calcul numéro 1 au top 500 des supercalculateurs mondiaux[13] utilise un réseau à 1 téraoctet par seconde pour l'accès au système de fichiers. Les grilles de calcul permettent ainsi de relier entre-elles des fermes de calcul pour fédérer leur puissance.

II.2.2 Historique des grilles

L'idée de grille de calcul apparait dans les premières heures de l'informatique (Laszewski, The Grid-Idea and Its Evolution 2005). En 1969, des références introduisent déjà la vision d'une infrastructure de grille (Tugend 1969) :

« We will probably see the spread of computer utilities, which, like present electric and telephone utilities, will service individual homes and offices across the country. »

Bien plus tard, au milieu des années 1990, après le développement entre autres : du pseudo-parallélisme dans les systèmes d'exploitation, de l'architecture clients-serveurs, d'internet, des autorités de certifications, apparaît le terme de « grille de calcul ». Le sens qu'on lui donne est alors assez vague (Foster et Kesselman, The Grids: Blueprint for a New Computing Infrastructure 1999); il qualifie une nouvelle infrastructure pour le calcul distribué dans le domaine des sciences et de l'ingénierie.

Des progrès considérables ont étés réalisés depuis dans l'utilisation des grilles (Stevens, et al. 1997) (Brunett, S. et al. 1998). Les technologies de grille ont elles aussi muri avec par exemple : le projet Globus (Foster et Kesselman, The Globus project: a status report 1999), le projet IPG (Information Power Grid) de la NASA (Johnston, Gannon et Nitzberg 1999) et le projet ASCI (Accelerated Strategic Computing Initiative) qui relie entre elles les ressources de calcul des sites abritant des armes nucléaires (Beiriger, et al. 2000).

Les grilles de calculs émergent alors comme un nouveau sujet d'une grande importance, séparées du conventionnel calcul distribué par la préoccupation de partage de ressources à grande échelle, le développement d'applications innovantes et une recherche de hautes performances (Foster, Kesselman et Tuecke, The Anatomy of the Grid 2001).

[13] http://www.top500.org

II.2.3 Définition

Le terme de grille est utilisé dans différents domaines, du réseau avancé à l'intelligence artificielle. Cependant, le concept de grille de calcul est bien défini et répond à des problèmes réels (Foster, Kesselman et Tuecke, The Anatomy of the Grid 2001) de partage coordonné de ressources et de résolution de problèmes dans des organisations virtuelles dynamiques et multi-institutionnelles. Le partage des ressources physiques est hautement contrôlé par leur fournisseur et leurs consommateurs, définissant clairement et précautionneusement, qu'est-ce qui est partagé, qui est autorisé à partager ainsi que les conditions de partage.

Un ensemble d'individus et / ou d'institutions partageant ces règles est appelé organisation virtuelle (Foster et Kesselman, The Grid: Blueprint for a New Computing Infrastructure 2nd Edition 2004). Les organisations virtuelles peuvent varier énormément en termes de but, de champ applicatif, de taille, de durée, de structure et de communauté. Néanmoins, une étude approfondie des besoins, amène à constater un ensemble commun de centres d'intérêts et de besoins. Par exemple, les gigantesques quantités de données produites par le LHC (Large Hadron Collider) du CERN ont amené la communauté de la physique des hautes énergies à interconnecter leurs fermes de calcul avec des réseaux haut débit et à mettre en place une infrastructure de type grille de calcul, et ainsi créer l'organisation virtuelle « LHC ».

La réalisation technique se fait avec la mise en place d'une couche logicielle appelée « intergiciel » ou « middleware ». Elle fournit à la couche applicative des services de grille de haut niveau, comme : la délégation de droits, le transfert et la réplication de fichiers. L'intergiciel de grille globus (Foster et Kesselman, The Globus project: a status report 1999) est très largement utilisé. Les grilles américaines OGS (Open Grid Science)[14] et européenne EGEE[15] (The Enabling Grids for E-sciencE project) sont toutes deux basées sur Globus. EGEE utilise gLite (Laure, et al. 2006), un intergiciel développé en surcouche de Globus. Ce projet regroupe à ce jour 120 partenaires, de 48 pays différents, regroupant ainsi 68 000 unités de calcul sur 250 sites, et permet l'exécution de 150 000 jobs de calcul quotidiennement.

[14] http://www.opensciencegrid.org
[15] http://www.eu-egee.org

II.2.4 La maturité

Actuellement le concept de « cloud computing », qui consiste à la fourniture de puissance de calcul par un fournisseur tiers, est considéré comme étant le prochain concept de premier plan dans le domaine du calcul distribué (Bégin 2008). Même si le « cloud computing » vole la vedette aux architectures type grille de calcul, les recherches visant à améliorer les infrastructures de grille sont encore très actives. Dans le dernier volume du magazine scientifique « Future Generation Computer Systems » on ne compte pas moins de six articles concernant les politiques d'ordonnancement sur grille de calcul et de nombreux autres concernent la gestion des données.

En plus de travaux sur l'amélioration de l'allocation des ressources (Lenica, et al. 2006), de la gestion de la distribution des données (Wei, Fedak et Cappello 2007), des politiques d'ordonnancement (Caron, et al. 2008), du développement d'applicatifs (Mateos, Zunino et Campo 2008), un effort est fait aujourd'hui en direction de la standardisation et de la convergence vers une architecture orientée service pour les composants de grille et notamment au sein de globus et de gLite . La Figure 13 présente les services offerts par une grille gLite. Ces services peuvent être regroupés en cinq catégories (Laure, et al. 2006) :

- Le service sécurité, qui permet ou refuse l'accès aux ressources.
- Le service information et suivi, qui permet la publication et la consultation d'informations.
- Le service de gestion des tâches, qui comprend le service d'ordonnancement, d'exécution, de traçabilité et l'installation d'application.
- Le service de gestion de données.
- Le service d'accès à la grille.

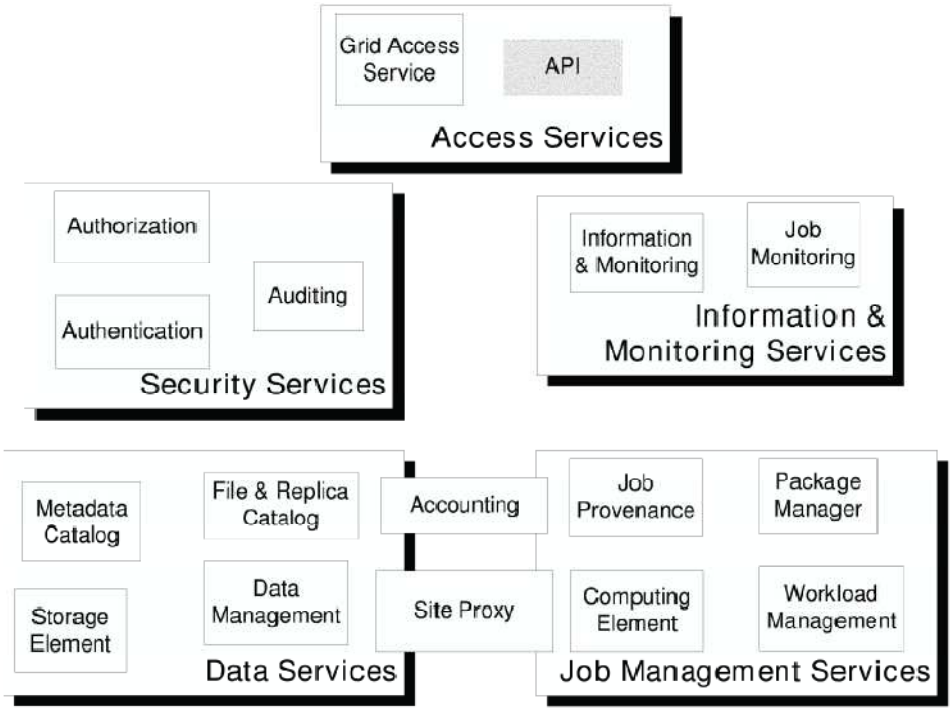


Figure 13 extrait de (Laure, et al. 2006) : Architecture des services dans gLite

L'utilisation des concepts d'architecture orientée service a permis le développement de standards comme OGSA (Open Grid Service Architecture) et OGSI (Open Grid Service Infrastructure) (Tuecke, et al. 2003). Ces standards permettent de fournir des composants de grille normalisés et travaillant de concert pour satisfaire les besoins de l'utilisateur final. Le standard OGSA développé par le Global Grid Forum[16] se base sur le standard WSRF (Web Services Resource Framework), un standard pour la mise en œuvre de services web à état, développé par l'OASIS[17] (Organization for the Advancement of Structured Information Standards).

La Figure 14 montre quels sont les rapports entre ces service et l'intergiciel de grille Globus Toolkit 4. Globus utilise WSRF pour implémenter des services de haut niveau adéquats pour le déploiement d'une grille. Ces services répondent au standard OSGA.

[16] http://www.ggf.org
[17] http://www.oasis-open.org

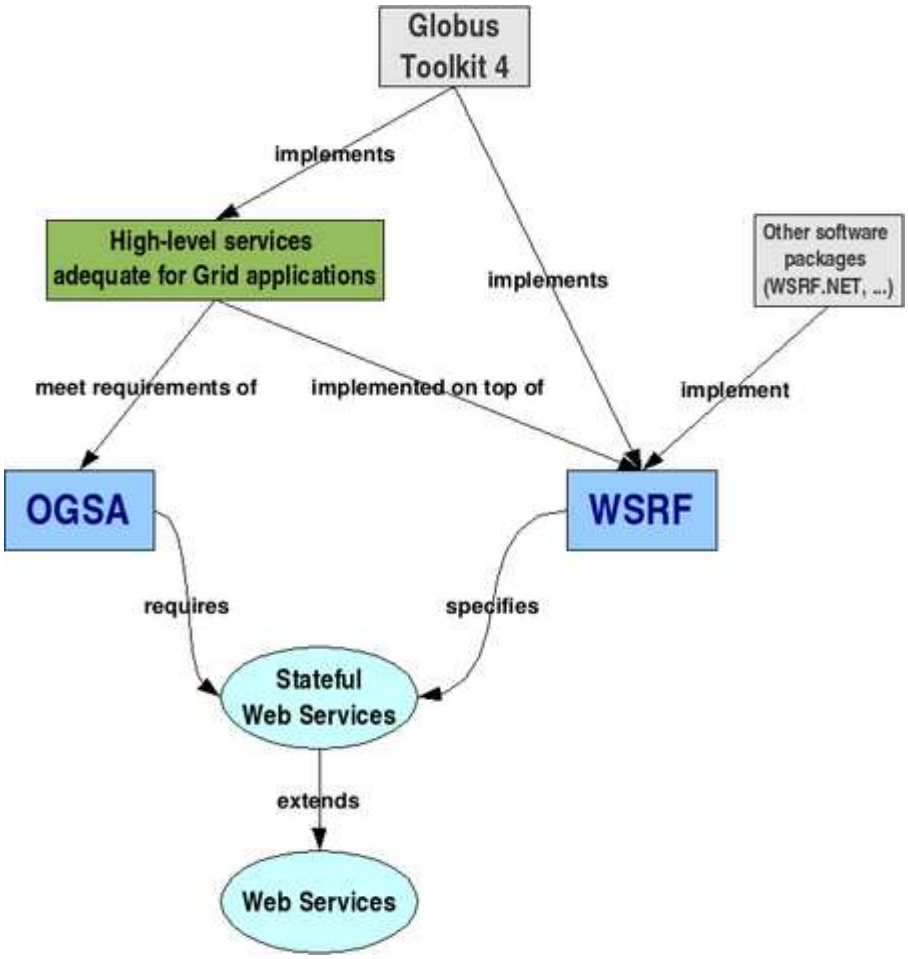


Figure 14 Les standards OGSA et WSRF dans Globus Toolkit 4[18]

La clarification des nouveaux concepts inhérents aux grilles de calcul amène aujourd'hui au développement d'un nouveau système d'exploitation basé sur Linux. Le projet XtreemOS[19] vise à fournir un support natif des organisations virtuelles et à fournir des services de base pour le partage des ressources de calcul et des données à grande échelle sur grille de calcul.

Même si l'ambition des grilles de calcul est de devenir une interface avec la puissance de calcul aussi simple d'utilisation qu'une prise de courant avec la puissance électrique, en pratique leur prise en main nécessite encore de connaître et de comprendre certains concepts internes à l'intergiciel. De plus, la gestion de l'exécution de plusieurs dizaines de milliers de processus en parallèle peut s'avérer être une tâche non triviale. La partie suivante présente ainsi les principaux logiciels de gestion d'exécution sur grille de calcul et grappe de calcul développés afin de masquer cette complexité.

[18] http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s01.html
[19] http://www.xtreemos.eu

II.3 Les outils d'exécution sur grille de calcul

II.3.1 GEL

GEL (Grid Execution Language) (Lian, et al. 2005) est un langage de script conçu pour la description et l'exécution de tâches en environnement distribué. Ce langage permet de masquer la double complexité inhérente à un environnement largement distribué comme une grille de calcul, à savoir des communications avec des temps de latence importants et une grande hétérogénéité. Il fournit des constructions syntaxiques pour des boucles et des opérations conditionnelles pour l'exécution parallèle de jobs. Il permet ainsi d'obtenir une représentation synthétique de l'application parallèle cible.

Ce langage constitue une abstraction par rapport à l'intergiciel. Il est donc possible d'exécuter les scripts aussi bien sur des machines à mémoire partagée, que des grappes de calcul, ou des grilles de calcul. GEL est capable de soumettre des jobs en utilisant PBS (Portable Batch System), SGE (Sun Grid Engine) ou LSF (Load Sharing Facility) et sur une grille de type Globus.

II.3.2 CoG

CoG (Commodity Grid) (Laszewski, Gawor, et al. 2003) (Laszewski, Hategan et Kodeboyina, Work Coordination for Grid Computing 2006) permet l'utilisation, l'administration et le développement d'applications sur grille en utilisant un outil de haut niveau.

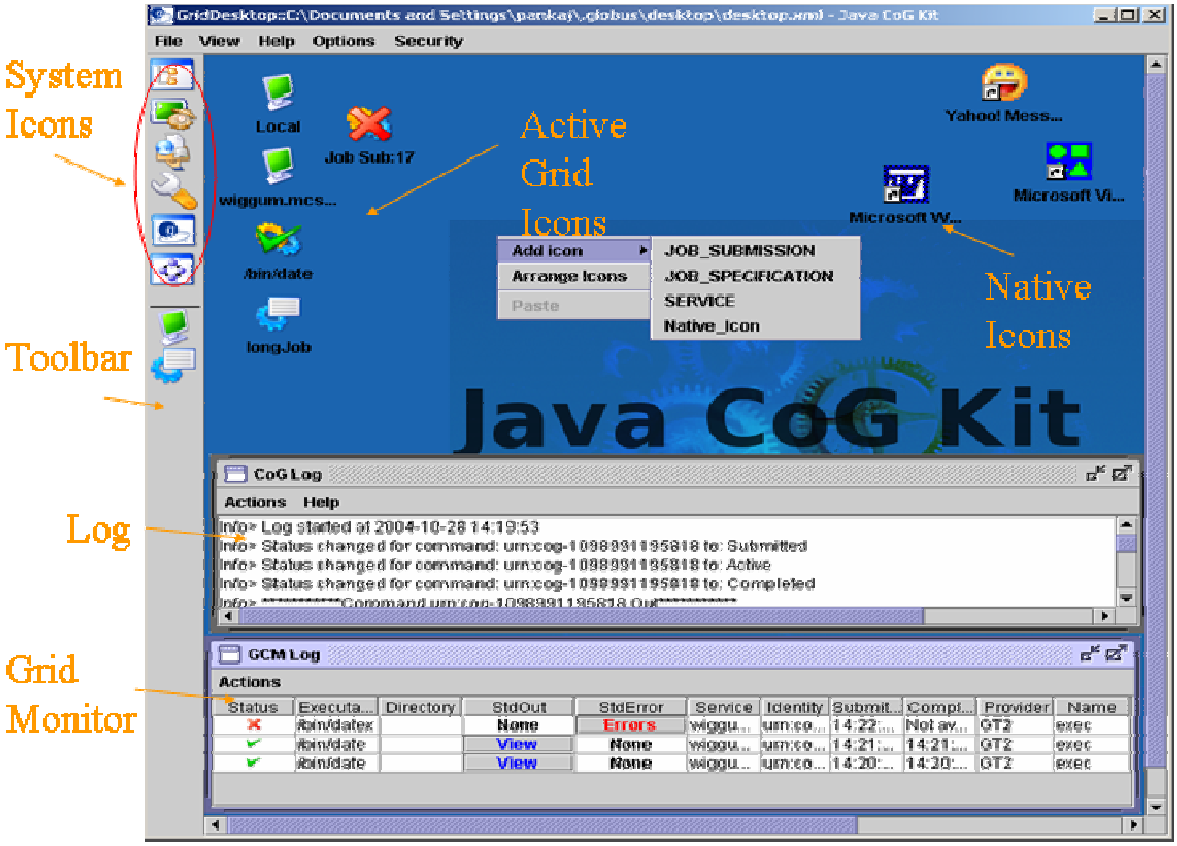


Figure 15 Capture d'écran de Java CoG Desktop

La boite à outil CoG est utilisable en Java et en Python. Elle est conçue en vue d'encourager la réutilisation et d'éviter la duplication de code dans le développement d'application « gridifiées ». Elle fournit des services comme la gestion transparente des données de sortie, avec une récupération automatique des fichiers de sortie et d'erreur standard, et une mise à jour automatique de l'état des jobs lors de leur exécution, l'abstraction de l'intergiciel de grille ou du protocole de transfert de fichier ainsi que la gestion de diagrammes de flux (workflow). La Figure 15 présente Java CoG desktop, une application basée sur CoG pour la soumission et la gestion de jobs via Globus.

II.3.3 Migrating Desktop

Migrating Desktop (Kupczyk, et al. 2005) est un cadriciel graphique visant à faciliter l'utilisation d'application de grille et la mise en pratique du concept « applications à la demande ». Le calcul à la demande, contrairement à l'approche traditionnelle, c'est-à-dire assigner des ressources aux applications, fait référence au concept de regroupement des ressources système et à l'allocation dynamique de celles-ci. Les fonctionnalités principales de Migrating Desktop concernent les applications

interactives de grille, la gestion de fichiers de grille et locaux, la gestion des autorisations pour accéder aux différentes ressources.

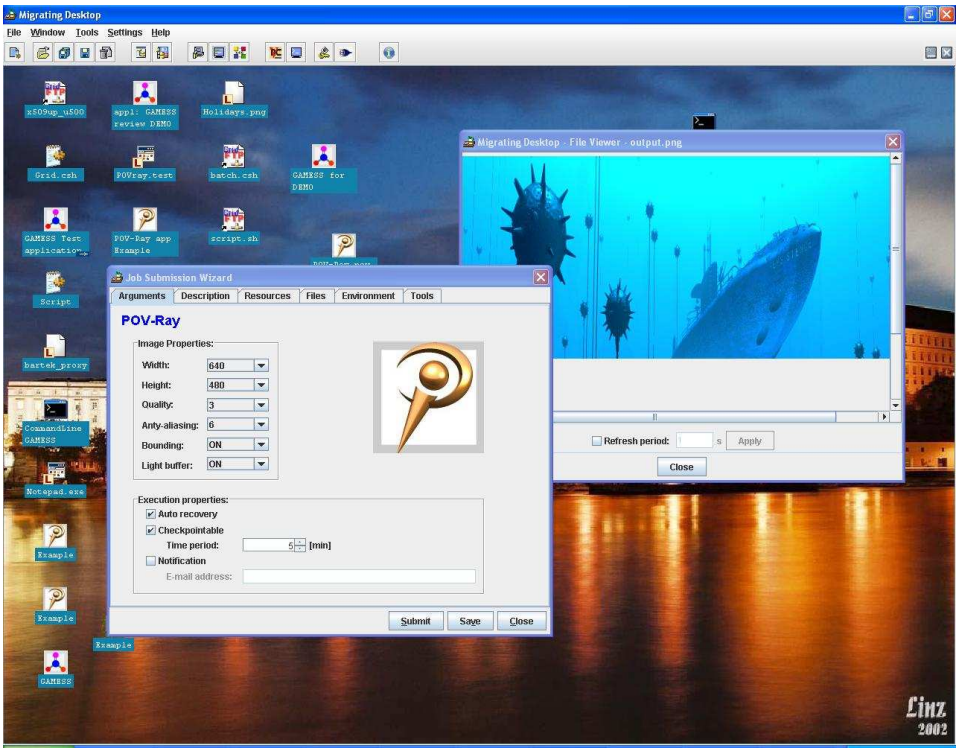


Figure 16 Capture d'écran de Migrating Desktop avec l'application Pov-Ray de synthèse d'images

La Figure 16 présente une capture d'écran de l'application. Les icônes sur le bureau sont des applications prêtes à être exécutées sur la grille. Cet environnement permet de travailler simultanément et de manière transparente avec plusieurs grilles. La communication avec l'environnement de grille s'établit au travers d'un ensemble de services web appelé « Roaming Access Server ». Ces derniers sont actuellement déployés pour la grille de calcul appelée BalticGrid[20].

II.3.4 g-Eclipse

Le projet g-Eclipse (Wolniewicz, et al. 2007) vise à concevoir un environnement de travail intégré pour accéder à la puissance des grilles de calcul. Comme le montre la Figure 17, cet environnement est basé sur « l'écosystème » fiable de la communauté Eclipse. Il permet l'utilisation d'applications conçues pour une exécution sur grille de calcul, la gestion des ressources et le support du cycle de développement de nouvelles applications pour la grille (compilation et débogage distants). Les développeurs de g-Eclipse prévoient d'y intégrer des outils existant, comme :

[20] http://www.balticgrid.org

- Migrating Desktop,
- GridBench (Tsouloupas et Dikaiakos 2003) (Dikaiakos 2007), un logiciel de test de performance pour les environnements de calcul distribué,
- Grid Visualisation Kernel (GVK) (Heinzlreiter et Kranzlmüller 2003), un outil de visualisation interactive de résultats de simulation pour globus.

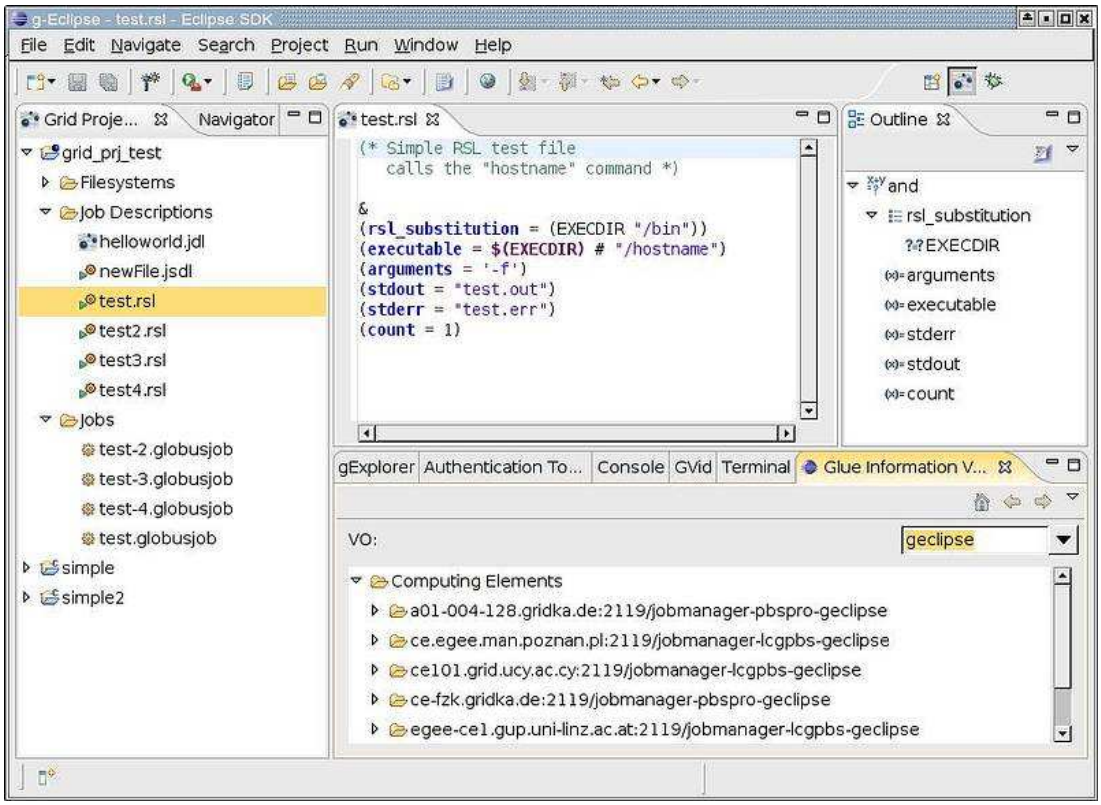


Figure 17 Capture d'écran de g-Ecplipse

G-Eclipse fournit un modèle de grille de haut niveau pour manipuler les objets de grille (jobs, fichiers…) indépendamment de la grille cible. Des instances de ce modèle sont implémentées et fonctionnelles pour : gLite, GRIA (Surridge, et al. 2005), une grille pour l'industrie orientée service, et les services de stockage S3 (Simple Storage Service) et de « cloud computing » EC2 (Elastic Compute Cloud) de Amazon. Une instance du model de grille de g-Eclipse pour la soumission de jobs de calcul sur fermes de calcul supportant PBS (Portable Batch System) est en cours d'implémentation.

II.3.5 Ganga

Ganga (Egede, et al. 2005) est une application implémentée en Python qui permet la définition, la génération et la gestion de jobs pour des environnements distribués. Ganga

fait abstraction de la plateforme d'exécution, que ce soient des processeurs de calcul locaux, une grappe de calcul ou une grille de calcul à grande échelle.

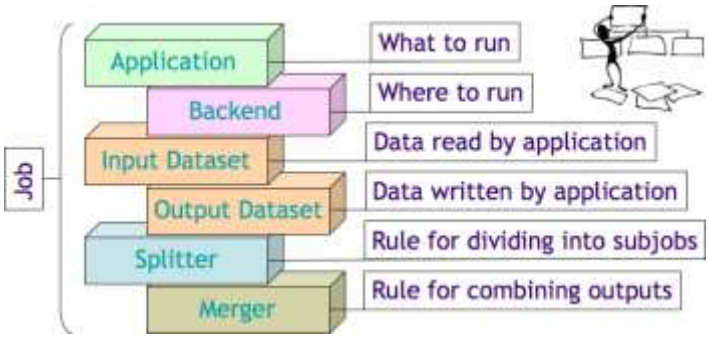


Figure 18 Macro architecture de Ganga[21]

Comme le montre la Figure 18, chaque job Ganga est constitué d'un ensemble de briques. Tous les jobs doivent spécifier l'application à lancer, le système d'exécution à utiliser, des données d'entrée et de sortie. Optionnellement, il est possible de définir des fonctions de découpage des fichiers d'entrée et de regroupement des fichiers de sortie.

Ganga utilise le langage de script iPython[22] qui permet de concevoir facilement des scripts de soumission ou de gestion des jobs. Une interface graphique (Figure 19) est disponible ainsi qu'un module de statistiques.

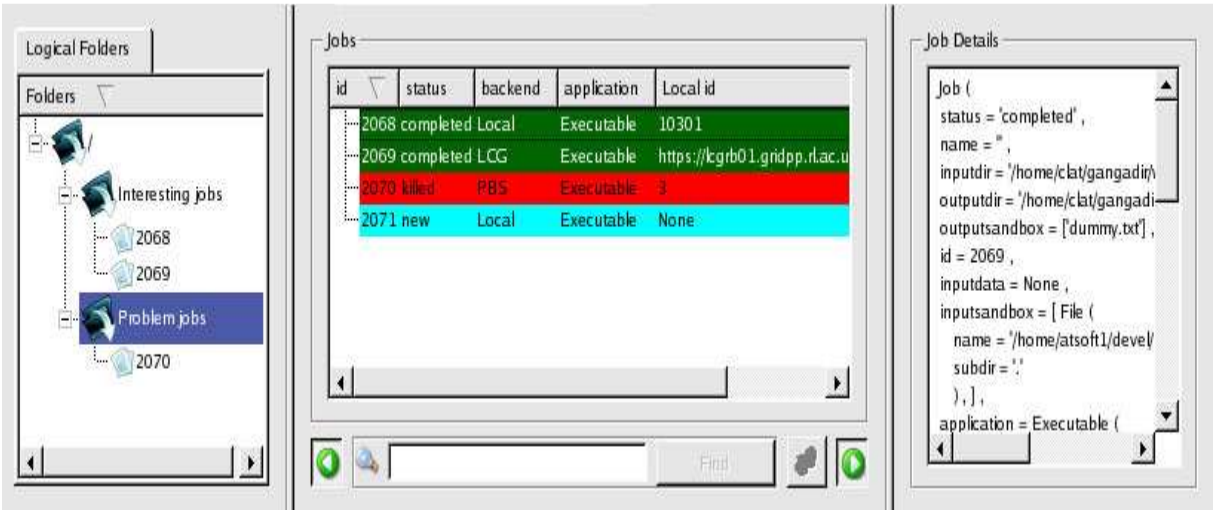


Figure 19 Interface graphique de Ganga

II.3.6 Proactive, une vision avancée de la conception d'applications de grille

Proactive (Baduel, et al. 2006) permet de faciliter l'exécution d'applications distribuées et notamment sur grille de calcul. Cependant, les ambitions de ce logiciel

[21] http://ganga.web.cern.ch/ganga/
[22] http://ipython.scipy.org/moin/

sont plus importantes. En effet, il peut être considéré comme un intergiciel écrit en Java, qui prend en charge les phases de développement, de composition et de déploiement d'applications distribuées.

La Figure 20 présente la macro architecture de Proactive. Cet intergiciel comprend un environnement de développement, une bibliothèque de classe, des fonctionnalités de déploiement et supporte de nombreux environnements d'exécution parallèle.

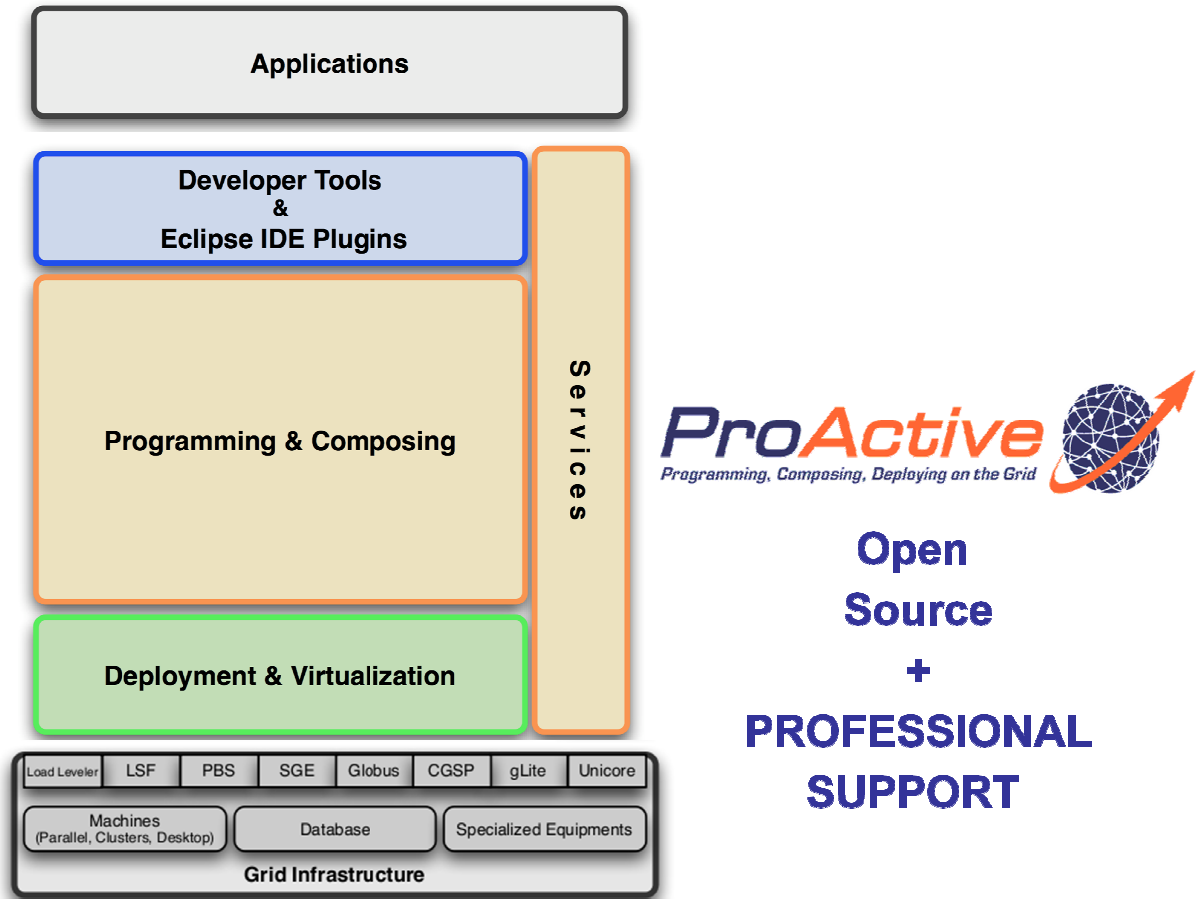


Figure 20 Présentation de proactive[23]

Proactive permet le développement d'applications distribuées en java. L'utilisation de Java permet de masquer l'hétérogénéité des environnements d'exécution ainsi que l'utilisation de RMI (Remote Method Invocation) et de l'API (Application Programming Interface) introspection java.

Une application distribuée Proactive est composée d'objets actifs. Les communications entre les objets actifs se fait via des appels de méthodes distantes asynchrones en utilisant RMI et le mécanisme de communication « wait-by-necessity »

[23] http://proactive.inria.fr/slides.htm

(Caromel, Toward a Method of Object-Oriented Concurrent Programming 1993). Proactive permet de plus de déplacer les objets actifs entre les machines virtuelles java participant au calcul distribué et des communications de groupes vers des objets actifs de même type.

Il est ainsi possible de composer des applications selon une approche nommée Fractale. Cette approche permet de composer des applications de manière simple en utilisant des objets actifs s'exécutant sur une ou plusieurs machines virtuelles java. Les composants communiquent entre eux via des ports clients, des ports serveurs et des liens (bindings). La Figure 21 illustre le principe de composition Fractale dans Proactive et expose les différents types de composants et de composition. Les compositions et les liens sont décrits de manière transversale en dehors du code java grâce à des fichiers XML (eXtensible Markup Language) et à l'utilisation des facilités d'introspection de java.

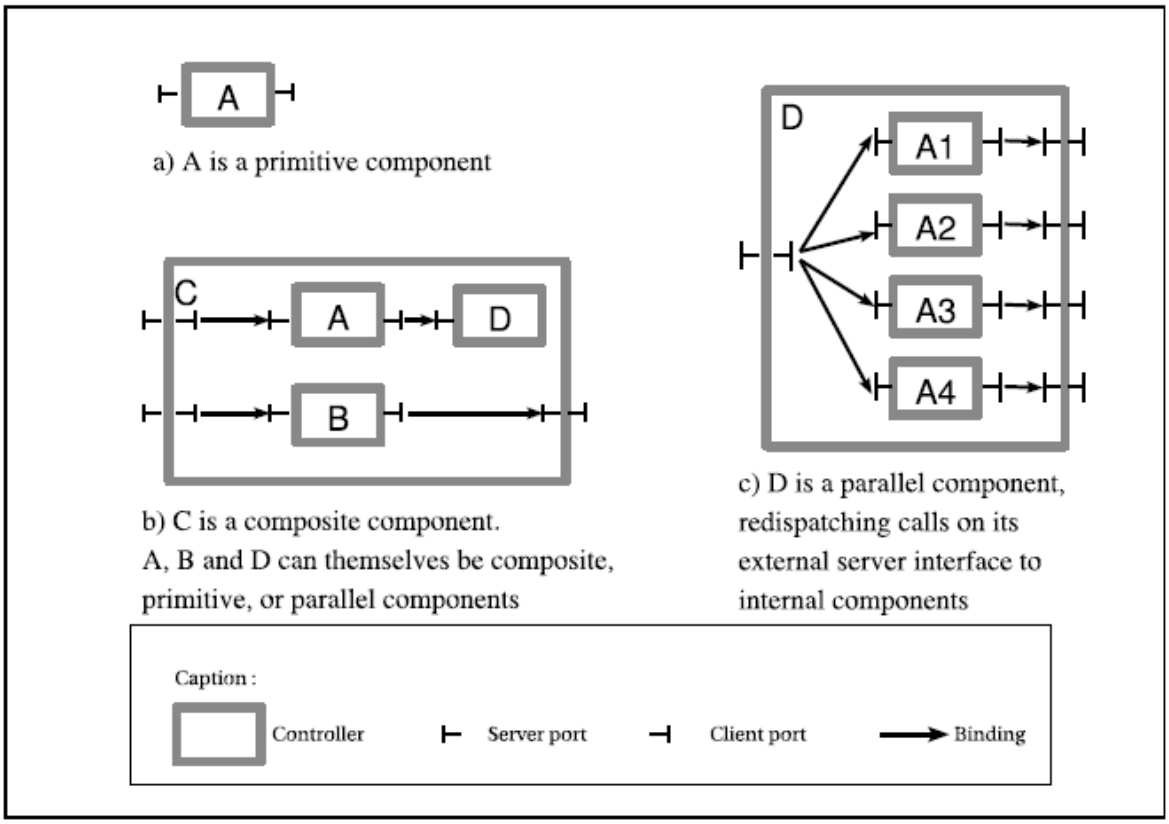


Figure 21 Extrait de (Baduel, et al. 2006) illustration du concept de composition Fractale dans Proactive

Enfin Proactive est basé sur l'utilisation de nœuds virtuels, qui permettent de découpler le code source de l'application distribuée des nœuds de l'environnement réel d'exécution distribuée. Seul des noms de nœuds virtuels sont mentionnés dans le code source. La phase de déploiement permet alors de lier les nœuds virtuels à des machines

virtuels java déployées, ou à déployer, sur des environnements cibles atteints via SSH (Secure Shell), Globus, PBS (Portable Batch System), etc.

Proactive est un environnement complet, prêt pour une utilisation industrielle. Une société SSLL (Société de Services en Logiciel Libre) a ainsi été créée récemment en vue d'offrir l'accélération d'application en utilisant les technologies d'exécution distribuée de Proacitve[24].

Comme nous l'avons vu dans cette partie, les nouvelles architectures de calcul distribué, comme le calcul pair à pair, ou les grilles de calcul, sont des technologies mûres et utilisables en production afin d'atteindre des puissances de calcul très importantes. Celles-ci permettent de relever de nouveaux défis scientifiques, comme le traitement de 15 péta-octets de données produites annuellement par le LHC (Large Hadron Collider). Ces architectures de calcul ne sont pas spécialisées et peuvent être utilisées dans de nombreux domaines, comme la bioinformatique, la physique des particules, les bases de données médicales… La simulation informatique peut elle aussi bénéficier de cette puissance. La prochaine partie présente les concepts et les outils pour la parallélisation pour des simulations déterministes.

III La parallélisation de simulations déterministes

La parallélisation d'une simulation déterministe vise à raccourcir son temps de simulation via une réduction de la complexité du modèle. La séparation des différents processus de la simulation permet ainsi de diminuer le temps d'exécution global. Dans certain cas, la parallélisation est une solution pour permettre de diviser l'espace mémoire nécessaire pour la simulation globale et ainsi d'augmenter la taille du modèle.

Cette partie présente tout d'abord la parallélisation de simulations à évènements discrets déterministes. Elle décrit le caractère historiquement séquentiel de la simulation à événements discrets et décrit succinctement, et de manière non-exhaustive, les méthodes de parallélisation qui lui sont associées (le lecteur trouvera de plus amples informations sur ces techniques de parallélisation dans (Fujimoto 1990) (Overeinde, Hertzberger et Sloot 1991) (Ferscha et Tripathi 1994) et (Lin et Fishwick 1996)). Dans les sous-parties suivantes, elle expose quelques-unes des bibliothèques, des outils et des standards d'interopérabilité pour la simulation distribuée sur grille de calcul.

[24] http://www.activeeon.com/

III.1 La parallélisation des simulations à événements discrets déterministes

III.1.1 L'aspect historiquement séquentiel de la simulation à événements discrets déterministes

Dans la simulation dirigée par les événements, une horloge centrale détient le temps global de simulation, c'est à dire le temps auquel le système physique est simulé. Une structure de donnée, appelée liste d'événements, contient un ensemble de messages et leurs dates de transmission prévues. Chacun de ces messages est retiré de la liste, et transmis à bonne date à son destinataire. A chaque étape, le message qui est associé à la date la plus proche dans le futur est traité. L'envoi de ce message peut à son tour déclencher la création de nouveaux messages ou provoquer l'annulation de messages déjà ordonnancés. Puis l'horloge est avancée à la date du dernier message traité. Une autre forme de simulation à événements discrets est appelée « dirigée par l'horloge ». L'horloge avance d'un « tick » à chaque pas de simulation.

Le concept de simulation distribuée émerge à la fin des années 70. Il est proposé par Bryant (Bryant 1977) et Chandy et Misra (Chandy et Misra, Distributed Simulation: A case study in design and verification of distributed programs 1979). Cependant ce modèle de simulation présente un aspect intrinsèquement séquentiel, du fait de la liste d'événements où les événements sont traités un par un. La structure de données ne peut être facilement répartie tout en préservant l'efficacité du modèle. L'adaptation de ce type d'algorithme pour une exécution parallèle fait apparaitre deux problèmes principaux : les erreurs de causalité et les étreintes fatales ou « deadlocks ».

Plusieurs publications proposent des résolutions aux problèmes des étreintes fatales : (Peacock, Wong et Manning, A distributed approach to queuing network simulation, (), pp. , . 1979), (Peacock, Wong et Manning, Distributed simulation using a network of processor 1979b) et (Holmes 1978), puis Chandy et Misra (Chandy et Misra, Asynchronous distributed simulation via a sequence of parallel computations 1981) et (Reynolds 1982). Dans l'exécution d'une simulation à événements discrets, certaines suites d'instructions doivent être exécutées avant d'autres. Par exemple, si A et B modifient la variable d'état X, l'événement avec la marque temporelle la plus petite doit être exécutée le premier pour éviter une erreur de causalité (Fishwick 1995). Les systèmes physiques obéissent toujours au principe de causalité, qui peut être énoncé de la manière suivante : « le futur n'influence jamais le passé ». Dans un système distribué, les calculs sont répartis sur plusieurs machines, l'avancement du temps simulé peut

donc être différent sur chaque machine. Le problème principal de la simulation distribuée est le maintien de la causalité. Trois approches pour les algorithmes de synchronisation existent. Les approches conservatives (ou pessimistes) fortes et faibles, pour lesquelles l'ordre chronologique des évènements est respecté strictement, sont introduites dans (Chandy et Misra, Distributed Simulation: A case study in design and verification of distributed programs 1979). Dans une autre approche dite optimiste, des évènements qui surviennent à des dates différentes peuvent être traités de manière parallèle.

III.1.2 L'approche pessimiste forte (ou synchrone)

Lors d'une simulation distribuée à horloge partagée, chaque processus de la simulation globale se connecte à un processus central de synchronisation ou processus coordinateur. A chaque « tick » d'horloge la simulation globale avance d'une unité de temps, c'est-à-dire que le processus coordinateur « donne la main » au processus qui doit exécuter l'événement dont la date d'occurrence est la plus petite. Le processus central de synchronisation peut être remplacé par une liste partagée des événements futurs, auquel cas, les événements qui se produisent à la même date sont exécutés en parallèle.

L'approche synchrone ne permet que de découpler faiblement les différents processus par rapport à l'approche séquentielle. Les autres méthodes de parallélisation permettent un découplage par désynchronisation des différents processus logiques, tout en évitant les fautes de causalité en utilisant l'estampillage des messages (Lamport 1978).

III.1.3 Approche pessimiste faible (ou asynchrone)

Dans l'approche asynchrone (Chandy et Misra, Distributed Simulation: A case study in design and verification of distributed programs 1979), un processus peut traiter tout événement qui n'a pas de conséquence sur les autres jusqu'à une date donnée dite barrière de synchronisation. Chaque processus logique utilise une file d'événements par processus duquel il peut recevoir des événements. Si une de ces files est vide, le processus se bloque en attendant de nouveaux messages. Le problème majeur de ce genre d'approche et l' « étreinte fatale » ou « deadlock ». De nombreux algorithmes ont étés conçus pour éviter, détecter et casser ces étreintes, parmi eux :

- L'envoi de messages « null » : après l'exécution d'un événement, un message vide avec uniquement la date de l'événement est envoyé aux autres processus.
- La « conservative time window » : une fenêtre de temps est calculée. Les événements compris dans cette fenêtre sont exécutables sans violation de causalité.
- La détection et le cassage de « deadlock » : si une étreinte est détectée, alors on cherche l'événement avec la date la plus petite dans toutes les files d'attentes et on le considère comme exécutable.

III.1.4 Approche optimiste

L'autre classe majeure d'algorithmes de synchronisation est appelée optimiste. Chaque processus logique a une seule file en entrée. Les événements sont traités dans l'ordre de temps croissant, mais sans se préoccuper si un événement avec un temps antérieur peut apparaître. Dans ce cas, le principe de la causalité peut être rompu.

(Jefferson 1985) propose une solution afin de retrouver une simulation cohérente : le « Time Warp ». Si une faute de causalité survient, la simulation est restaurée dans un état valide par un mécanisme dit de « rollback » et reprend son exécution. Le problème de ce type d'algorithme est qu'il consomme beaucoup de mémoire du fait qu'il doive stocker les états valides avant l'exécution de chaque événement. De nombreux algorithmes ont été développés pour minimiser le besoin en mémoire. La technique du point de synchronisation périodique (Periodic Checkpointing), par exemple. Dans cette technique on sauvegarde l'état du processus de façon périodique, puis on sauvegarde les événements exécutés depuis cet état.

III.2 Quelques outil pour la parallélisation d'applications

III.2.1 Neko

Neko (Urbán, Défago et Schiper 2002) est une plateforme implémentée en Java. Elle permet le prototypage, la conception, le paramétrage, le déploiement et l'analyse de performances d'algorithmes distribuées.

Comme le montre la Figure 22, Necko permet le développement d'application selon une architecture en couche. Une couche est soit active, dans le cas ou un processus léger ou « thread » lui est dédié, soit passive dans le cas contraire. Les couches passives sont des instances de classes héritant de la classe `Layer` et les couches actives héritent de

`ActiveLayer`. L'envoi de message « unicast » ou « multicast » se fait à l'aide des instances de la classe `NeckoMessage` via l'utilisation de communications réseau TCP (Transmission Control Protocol), UDP (User Datagram Protocol) ou d'un réseau simulé. Tous les processus légers de Necko sont reliés à une instance de `NeckoProcess`. Il est ainsi possible de partager des zones de mémoire entre les processus légers reliés au même `NeckoProcess`.

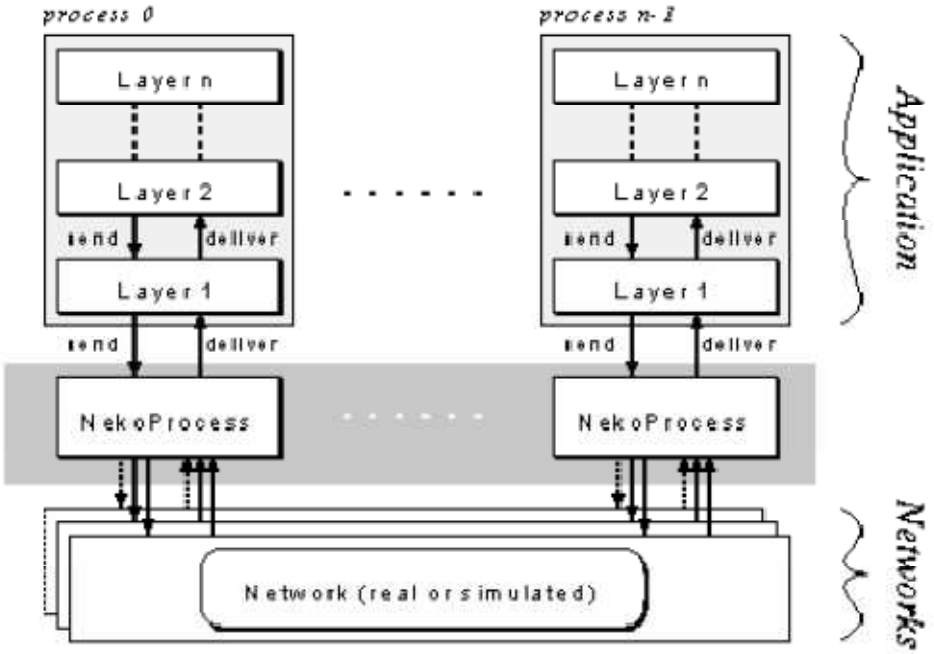


Figure 22 Architecture d'une application neko

III.2.2 Comet

Comet (Peschanski et J.P. 2004) est un intergiciel pour l'implémentation de systèmes répartis. Il permet la définition de composants et d'événements ainsi que des relations de type « hérite de » dans un langage spécifique appelé « Scope ». Les composants peuvent communiquer entre eux de manière distribuée par l'envoi d'événements. Les communications réseau, implémentés dans Comet utilisent les protocoles TCP et UDP.

Les événements sont des objets, se rapprochant de simples structures de données, comprenant des attributs, des constructeurs, et des accesseurs.

Les composants sont des entités définissant :

- les messages qu'ils peuvent recevoir,
- les messages qu'ils peuvent émettre,

- un comportement défini pour la réception d'un message d'un type donné.

Les destinataires des messages ne sont pas spécifiés lors de la conception des composants. Les composants sont reliés entre eux par des connections dynamiques, typées, unidirectionnelles et asynchrones (Peschanski et Briot 2005). Les connexions s'effectuent au travers de la commande « connect » du langage Scope. Celle-ci permet de connecter entre eux deux composants (source vers destination) pour un type de message donné.

Comet permet de plus, de définir des « Rôles » et des « Protocoles » afin de redéfinir dynamiquement et sans interruption de service le comportement des composants.

III.2.3 PVM (Parallel Virtual Machine)

D'après (Geist, et al. 1994), le projet PVM (Parallel Virtual Machine) a débuté en 1989 au laboratoire national d'Oak Ridge. Le prototype PVM 1.0 fut implémenté par Vaidy Sunderdam et Al Geist. Cette version fut utilisée uniquement en interne. La version 2 de PVM a été écrite par l'université du Tennessee, et publiée en mars 1991. Durant l'année suivante, PVM commença à être utilisé par de nombreuses applications scientifiques. Les retours des utilisateurs amenèrent nombre de changements et une réécriture complète du code pour la version 3.0 publiée en février 1993.

Aujourd'hui PVM à atteint la maturité et la dernière version publiée est la version 3.4.5[25]. Dans la version 3 de PVM, la communication entre les processus d'une application distribuée (envoi de message, synchronisation) se fait via l'utilisation de la bibliothèque « libpvm » et au travers d'un processus serveur, le « daemon » PVM.

De nombreuses implémentations de PVM sont maintenant disponibles, notamment en Java[26] en Perl[27] en Python[28]. La version ePVM (Frattolillo 2005) permet une utilisation de PVM sur grille de calcul. Pour cela il suffit que les nœuds de calcul aient accès à internet. Il n'est pas nécessaire qu'ils possèdent une adresse IP (Internet Protocol) publique.

[25] http://www.csm.ornl.gov/pvm/pvm_home.html
[26] http://www.cs.virginia.edu/~ajf2j/jpvm.html
[27] http://www.csm.ornl.gov/pvm/perl-pvm.html
[28] http://pypvm.sourceforge.net

III.2.4 OpenMP

OpenMP[29] (Dagum et Menon 1998) est une API (Application Programming Interface) pour la parallélisation d'application en C, C++ et en fortran. La spécification de l'aspect parallèle d'une application sur des machines à mémoire partagée avec OpenMP s'effectue par inclusion de directives de compilation dans le code, d'utilisation d'une bibliothèque de fonctions et de variables d'environnement.

Alors que la spécification 1.0 de OpenMP date de 1997 pour la version fortran et 1998 pour la version C / C++, la première implémentation pour un réseau de machines à mémoire partagée est publiée en 2000 (Hu, et al. 2000). A l'heure actuelle, les spécifications stables d'OpenMP sont dans leur version 2.5 alors que la soumission de commentaire pour la future version 3.0 s'achevait le 31 janvier 2007.

Il est envisageable de paralléliser des applications en utilisant OpenMP pour du calcul massivement parallèle. Pour preuve il a été utilisé conjointement à MPI (Message Passing Interface) sur le Earth Simulator pour du calcul élément fini distribué (Nakajima 2005). A ma connaissance, OpenMP n'est pas utilisé pour du calcul sur grille.

III.2.5 MPI (Message Passing Interface)

MPI (Message Passing Interface) est un standard pour la programmation parallèle proposé pour la première fois par Walker dans (Walker 1994). La conception de MPI est issue d'un effort collectif de chercheurs Européens et Américains de différentes organisations et institutions.

MPI a été écrit pour obtenir de bonnes performances aussi bien sur des machines parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Il inclut des normes de communication point à point et collectives, une bibliothèque de fonctions et un compilateur associé. Il est grandement disponible sur de très nombreux matériels et systèmes d'exploitation. MPI est portable (implémenté sur différentes architectures de mémoires) et rapide (optimisé pour le matériel sur lequel il s'exécute).

D'après (Laforenza 2002) de nombreuses solutions existent pour exécuter des applications distribuées utilisant MPI sur grille : PACX-MPI (Brune, Fagg et Resch 1999), MPI Connect (Fagg, London et Dongarra 1998), Stampi (Imamura, et al. 2000),

[29] http://www.openmp.org

MPICH/Madeleine III (Aumage, Eyraud et Namyst 2001), MagPIe (Kielmann, et al. 2002), MPICH-G2 (Karonis, Toonen et Foster 2003)[30], MetaMPICH (Clauss, Pöppe et Bemmerl 2004)[30].

Les travaux de recherches concernant l'utilisation de MPI sur grille sont nombreux. Le magazine « Future Generation Computer Systems » a publié en février 2008 une section spéciale intitulée : « *Grid computing and the message passing interface* ». Cette section présente des projets comme Migol (Luckow et Schnor 2008) : un intergiciel de grille pour l'exécution parallèle avec MPI basé sur Globus, permettant une tolérance accrue aux fautes et MGF (Gregoretti, et al. 2008) : une librairie basée sur MPICH-G2 qui permet l'utilisation transparente de ressources de grille. Ces deux projets permettent de prendre en compte les spécificités d'une architecture grille, et notamment la possibilité d'établir des communications inter-grappe de calcul entre des nœuds appartenant à des réseaux privés et qui ne possèdent pas d'adresse IP (Internet Protocol) publique.

III.2.6 CORBA (Common Object Request Broker Architecture)

CORBA (Common Object Request Broker Architecture) est un intergiciel orientée objet pour l'appel de méthodes résidant dans le même espace d'adressage ou dans des espaces d'adressages distants. CORBA est un standard proposé par l'OMG (Object Managment Group).

Une plateforme d'objets répartis comme CORBA comprend un bus d'objets réparti (ou ORB Object Request Broker). Une description claire de ce composant est faite dans (Peschanski et Briot 2005) : « *Ce logiciel dédié, […] représente la clef de voûte de toute l'architecture intergicielle. L'ORB permet à des objets implémentés dans des environnements variés et localisés sur des machines réparties géographiquement d'interagir entre eux. Il implémente les protocoles fondamentaux pour le bon fonctionnement des objets répartis, en particulier la gestion du cycle de vie des objets (déploiement sur des sites distants, localisation d'instances, etc.) ainsi que les invocations distantes de méthodes. Le critère fondamental pour un bus d'objet réparti est de proposer ce genre de service de manière transparente, en s'abstrayant dans les plus larges mesures possibles des contraintes liées à la nature répartie et hétérogène de l'infrastructure sous-jacente.* »

[30] Références parues depuis l'article de 2002

CORBA utilise de plus un langage de définition d'interface appelé IDL (Inteface Definition Language) qui permet de décrire l'interface que présente un objet pour la communication avec les autres objets de l'application distribuée.

CORBA a été initialement conçue pour le développement d'applications distribuées. Il permet aujourd'hui de faire communiquer des applications distantes écrites dans des langages différents. Des technologies similaires ont été implémentées par Microsoft avec DCOM (Distributed Component Object Model) et par Sun avec RMI (Remote Method Invocation) de la plateforme Java.

Des travaux ont été menés pour l'utilisation de CORBA sur grille de calcul. Parmi eux, ceux présentés dans l'article (Chunlin et Layuan 2003) présentent un intergiciel de grille basé sur CORBA et ceux présentés dans l'article (Wang 2008) présentent l'implémentation d'un algorithme génétique sur grille calcul utilisant la bibliothèque paCO (Parallel CORBA) (Denis, et al. 2003). PaCO est une bibliothèque permettant à une application distribuée de communiquer en utilisant aussi bien une approche de passage de messages, à la MPI, que des invocations distantes de méthodes, à la CORBA, dans le but affiché de simplifier l'implémentation de simulations distribuées sur grille de calcul.

III.3 Quelques outil pour la parallélisation de simulations à événements discrets

III.3.1 Maisie

Maisie[31] est un langage de simulation basé sur C et développé par le département de science de l'Université de Californie. Il permet l'exécution séquentielle ou distribuée de simulations à événements discrets.

Maisie permet de définir des entités, sous forme de structures en C. Celles-ci communiquent entre elles via l'envoi de messages asynchrones. L'environnement de développement Maisie comprend un compilateur : « mc ». « mc » prend en paramètre le type d'exécution souhaité pour la simulation. L'utilisateur a le choix entre :

- une exécution séquentielle,
- une distribution conservative (pessimiste) de sa simulation à événement discret,

[31] http://pcl.cs.ucla.edu/projects/maisie/

- une distribution optimiste de sa simulation.

Dans Maisie, les communications interprocessus lors de l'exécution des simulations distribuées utilisent MPI.

III.3.2 Distributed SimJava

SimJava (McNab et Howell 1996) (Howell et McNab 1998) est un paquetage écrit en Java pour l'implémentation de simulation à événements discrets. C'est un projet similaire à Sim++ / SimPack, avec des fonctionnalités pour l'animation graphique.

Une simulation SimJava est un ensemble d'entités exécutées chacune par un processus léger (thread). Ces entités sont interconnectées via des ports et communiquent en délivrant des événements. Une classe centrale contrôle tous les processus légers, gère l'avancement de la simulation et délivre les événements.

Le projet Distributed SimJava (Page, Moose et Griffin 1997) visait à utiliser les RMI (Remote Methode Invocation) du JDK 1.1 (Java Developpement Toolkit) afin de distribuer des simulations Simjava. L'architecture utilisée dans Distributed Simjava est une architecture de type client-serveur. Le serveur maitre encapsule une instance de la classe centrale de la simulation : `Sim_system`. Celui-ci coordonne les activités des entités, qui sont des instances de la classe `Sim_entities`. Les entités peuvent êtres distribuées sur un réseau.

III.3.3 La distribution de simulations à évènements discrets basée sur DEVS

DEVS (Discrete Event System Specification) est un formalisme mathématique de modélisation et de simulation à événements discrets (Ziegler, Kim et Praehofer 2000). Il a été proposé par Ziegler en 1976 dans l'ouvrage (Zeigler 1976). DEVS permet la mise en place de modèles modulaires et hiérarchisés.

Il y a deux sortes de modèles DEVS : atomique ou couplé. Les modèles atomiques sont indivisibles, dynamiques et sous forme états-transitions. Ils comportent des ports d'entrée, de sortie, un ensemble d'états internes, des fonctions de transitions, de sortie et d'avancement temporel. Les modèles couplés décrivent comment associer entre eux les modèles qui les composent. Ils comportent des ports d'entrée, de sortie, un ensemble de modèles et leurs couplages.

Les bases mathématiques de DEVS issues de la théorie des systèmes permettent de vérifier le comportement de systèmes spécifiés avec DEVS grâce au concept d' « untimed

behaviour » (Hong et Kim 1996) (Kim, Cho et Lee 2001) (Hwang, Identifying equivalence of DEVSs: Language approach 2003). D'autres travaux ont été menés pour la vérification des systèmes temps-réel (Hwang, Tutorial: Verification of Real-time System Based on Schedule-Preserved DEVS 2005).

Dans (Ziegler, Kim et Praehofer 2000), les auteurs présentent des solutions pour spécifier avec DEVS des simulations à événements discrets distribuées, selon une approche pessimiste et optimiste. De plus, différentes publications relatent des travaux pour l'exécution de simulations spécifiées avec DEVS sur des environnements de calcul distribué. Parmi elles, on trouve DEVS/Grid (Seo, et al. 2004), un cadriciel basé sur globus qui permet de partitionner de manière optimum les modèles hiérarchiques et de les exécuter sur grille de calcul. Le logiciel DEVS/P2P (Cheon, et al. 2004) se base sur le standard d'implémentation pour les architectures pair à pair de Sun Microsystems JXTA (JuXTApose) pour l'exécution distribuée selon une approche pair à pair de simulations DEVS.

III.4 Standards d'interopérabilité pour la simulation distribuée

III.4.1 DIS

Les environnements virtuels distribués sont nombreux. Un état de l'art de ces systèmes datant d'une dizaine d'années (Torguet 1998) en recense à l'époque plus d'une dizaine (RB2, VLNET, MASSIVE-2, etc.), ainsi que deux normes : VRML (Virtual Reality Modeling Language) qui permet la modélisation de mondes virtuels au sein d'internet et HLA (High Level Architecture) présenté plus bas. Parmi eux, DIS (Distributed Interactive Simulation) est un standard ouvert (IEEE 1278) pour la simulation distribuée de jeu de guerre en temps réel qui fait suite au projet SIMNET (SIMulation NETworking) présenté dans (Calvin, et al. 1993).

Au milieu des années 1980, la DARPA (Defense Advanced Research Projects Agency) du département américain de la défense crée le projet SIMNET dans lequel des objets autonomes interagissent entre eux par envoi d'événements. Dans SIMNET les objets ne transmettent que des informations sur leurs changements d'état pour limiter la transmission d'informations redondantes. Ce projet spécifie des concepts qui seront repris dans DIS, par exemple le processus de « Dead Reckoning » qui permet d'extrapoler l'état d'un objet en se basant sur l'historique de ses états antérieurs.

SIMNET présente différents inconvénients comme l'utilisation du « multicasting » lié aux réseaux de type Ethernet et des PDU (Protocol Data Unit) trop spécifiques.

DIS améliore ces aspects en se détachant du protocole réseau de bas niveau et en se basant sur un nombre de PDU plus élevé et des PDU plus génériques. Dans DIS, chaque entité est exécutée sur une station de travail par un logiciel nommé « joueur » qui est présumé honnête et ne triche pas sur les événements qu'il envoie. Sur chaque station, les autres objets de la simulation son générés par un processus appelé Fantôme basé sur un algorithme de « Dead Reckoning » mis à jour à chaque itération de la simulation. Il n'y a pas de base de données centrale. Chaque entité met à jour son environnement en se basant sur les PDU envoyés par les autres unités.

Les PDU de DIS représentent par exemple le tir d'une balle ou une détonation. Ce standard est donc limité à la spécification de simulations distribuées temps réel militaire. Cependant DIS a permis la naissance d'autres standards et notamment de HLA pour la simulation distribuée.

III.4.2 HLA

HLA (High Level Architecture) est une architecture pour la simulation distribuée normalisée par la norme IEEE 1516 « Standard for Modeling and Simulation: High Level Architecture - Framework and Rules ». HLA est présentée par le « U.S. Defense Modeling and Simulation Office » en 1995 (Miller 1996). Cette norme remédie à certaines lacunes de DIS. Elle permet par exemple le développement de simulations à évènements discrets en temps logique, et simplifie l'interopérabilité et la réutilisabilité des composants de simulations. Les aspects qui ont guidé la conception de HLA sont (Drira, Martelli et Villemur 2001) :

- Aucune unique simulation monolithique ne peut satisfaire les besoins de l'utilisateur.
- Toutes les utilisations d'une simulation et les manières intéressantes de la combiner avec d'autres ne peuvent être anticipées.
- Les capacités technologiques à venir et la variété de configurations opérationnelles doivent pouvoir être utilisables.

C'est de fait un candidat idéal pour distribuer des simulations à grande échelle. HLA met en œuvre des ensembles de simulations interopérables, qui sont appelées

fédérations, et des composants de simulation, appelés fédérés. La norme définit un ensemble de règles pour l'interopérabilité des simulations au sein d'une fédération. Ces règles spécifient les responsabilités des fédérations et du RTI (RunTime Infrastructure) au travers duquel les fédérés ne peuvent communiquer entre eux, comme illustré dans la Figure 23.

HLA définit d'autres aspects du développement de simulations distribuées comme le DDM (Data Distribution Managment) et l'OMT (Object Model Template) qui ne sont pas présentés ici. De plus, un processus de développement lui est associé : le FEDEP (Federation Development and Execution Process).

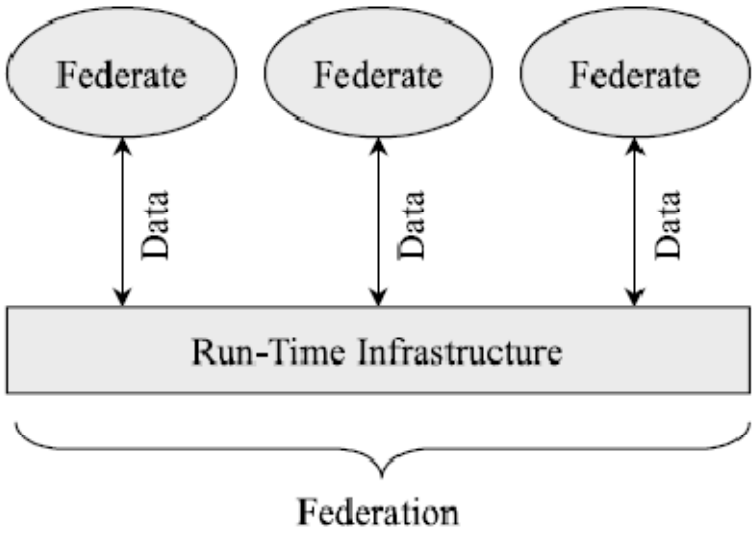


Figure 23 Extrait de (Petty 2002) : Communication entre les Fédérés au travers du RTI dans HLA

Ces dernières années, des solutions ont été développées pour exécuter des simulations HLA sur grille. Certaines recherches concernent la migration des fédérés entres les différents éléments de calcul d'une grille globus (Zajac, Bubak, et al. 2003) (Cai, et al. 2005). D'autres visent à implémenter un RTI pour l'intergiciel de grille globus et notamment en se basant sur OGSA, l'architecture de service implémentée dans globus (Zajac, Tirado-Ramos, et al. 2004), (Xie, et al. 2005) ou (Zhang, et al. 2005) (Chen, et al. 2008).

Dans cette partie nous avons vu qu'il était possible de paralléliser une simulation déterministe afin de réduire son temps global d'exécution en utilisant la puissance d'environnements distribués comme des fermes ou des grilles de calcul. Cependant l'interdépendance des processus et la limitation en bande passante réseau ne permettent pas d'obtenir une réduction du temps global de simulation qui diminue

linéairement avec le nombre de processeurs utilisés sur des grilles calcul, comprenant plusieurs dizaines de milliers de processeurs et des bandes passantes réseau relativement faibles. Contrairement aux simulations déterministes, les simulations stochastiques sont naturellement adaptées à l'exploitation intensive d'environnements massivement distribués. C'est l'objet de la partie suivante.

IV Approche MRIP (Multiple Replication In Parallel)

Les simulations stochastiques présentent un aspect intrinsèque naturellement parallèle du fait que l'obtention des indicateurs statistiques requiert l'exécution de nombreuses expériences indépendantes. L'indépendance des expériences permet une exécution distribuée sans communication interprocessus, mais constitue cependant une contrainte forte qui doit être respectée pour obtenir des résultats de simulations valides.

MRIP (Multiple Replication In Parallel) est une approche de parallélisation consistant à exécuter en parallèle le même programme de simulation de façon indépendante en utilisant des flux de nombres pseudo-aléatoires de qualité et non-corrélées entre eux.

Cette partie présente tout d'abord les bibliothèques de tests statistiques actuellement disponibles pour détecter de manière empirique les faiblesses statistiques et les corrélations dans les séquences de nombres pseudo-aléatoires. Dans un deuxième temps, sont exposées les bibliothèques de génération de nombres pseudo-aléatoires parallèles. Enfin je fais état des environnements de distribution de simulations automatique selon l'approche MRIP.

IV.1 Test des générateurs de nombres pseudo-aléatoires

IV.1.1 Les batteries de test

John Von Neuman (1951) : (Random number generators) « *… probably… can not be justified, but should merely be judged by their results. Some statistical study of the digits generated by a given recipe should be made, but exhaustive tests are impractical. If the digits work well on one problem, they seem usually to be successful with others of the same type* »

Les générateurs de nombres pseudo-aléatoires modernes utilisés en simulation sont conçus selon des études théoriques qui permettent de connaître certaines de leurs

propriétés statistiques comme la période du cycle de génération. Cependant nous n'avons pas les outils mathématiques à l'heure actuelle pour trouver tous les défauts d'un algorithme de génération de nombres pseudo-aléatoires et seuls des tests statistiques empiriques sur une partie du cycle de génération permettent de valider une utilisation en simulation.

Les tests statistiques considèrent le générateur à tester comme une boite noire. Ils calculent des données statistiques issues de fractions de la séquence de sortie du générateur avec pour objectif de trouver des défauts significatifs (L'Ecuyer, Random Number Generators and Empirical Tests 1998). Il y a une infinité de tests possibles, cependant les générateurs de nombres pseudo-aléatoires sont généralement testés en utilisant des simulations dont on sait qu'elles sont sensibles à certains types de corrélations, et dont les résultats sont connus par des méthodes exactes (Hellekalek, Don't trust parallel Monte Carlo 1998) et des tests statistiques.

Les tests statistiques les plus courants sont : le calcul du Chi2, le test de Kolmogorov-Smirnov, le calcul de moyenne et de l'entropie des suites générées (Hamming 1980), le test d'équidistribution, le test sériel (serial test), le test de distance (gap test), le test du poker, le test du collectionneur de coupon (coupon collector's test), le test de permutation, le test de croissance, décroissance (run test), le test du maximum de t, le test de collision et le test d'auto-corrélation. Ces tests sont présentés dans (D. Knuth 1969). D'autres tests statistiques ont été conçus plus récemment (Matsumoto et Nishimura, A Nonempirical Test on the Weight of Pseudorandom Number Generators 2002) (Matsumoto et Nishimura, Sum discrepancy test on pseudorandom number generators 2003).

En règle générale, les tests intéressants sont ceux dont on suppose qu'un mauvais résultat peut avoir un impact négatif sur notre simulation. Les tests biens conçus doivent couvrir une grande partie des problèmes que l'on rencontre couramment dans les simulations. Une description détaillée de la plupart des tests se trouve dans (Rütti, A Random Number Generator Test Suite for the C++ Standard. Diploma Thesis. 2004). On les retrouve parmi d'autres dans les batteries de tests statistiques.

Quelques unes des batteries de tests actuellement disponibles permettent de tester des flux de nombres aléatoires parallèles. Une batterie de tests pour générateurs de nombres pseudo-aléatoires parallèles devrait permettre d'évaluer la qualité de chacune

des séries utilisées ainsi que les corrélations inter-flux (Srinivasan, Ceperley et Mascagni, Random Number Generators for Parallel Applications 1999). La première publication que j'ai pu trouver à ce sujet date de la fin des années 1980 (Durst 1988), depuis plusieurs méthodes d'autres tests ont été publiées.

Parmi elles, on trouve le test d'intercalage (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003), qui consiste à intercaler des séquences de *n* flux de nombres pseudo-aléatoires par bloc et à faire passer une batterie de tests séquentiels à la série de nombres ainsi obtenue.

Le test de blocage (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003) est basé sur le théorème central limite, d'après lequel la somme de n variables aléatoires de moyenne 0 et de variance unitaire approche une distribution normale centrée réduite :

$$\frac{R_1 + \cdots + R_n - n/2}{\sqrt{n}\sqrt{\frac{1}{12}}} \xrightarrow{\mathcal{L}} N(0,1)$$

Où N (0,1) est la loi normale centrée réduite. Ce test est effectué pour n séquences aléatoires uniformes indépendantes R_1, R_2, ..., R_n dans l'intervalle [0, 1]. Si *n* flux de nombres aléatoires sont indépendants, leur somme doit satisfaire à un test d'ajustement à la loi normale. Plus le nombre de séquences de départ est important, plus l'approximation est justifiée.

A ma connaissance, les autres tests statistiques pour des générateurs de nombres pseudo-aléatoires parallèles publiés sont : le test de la transformée de Fourier (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003) et le test des sommes exponentielles (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003). Les autres tests sont des tests empiriques basés sur des applications : les articles (Coddington et Ko, Techniques for Empirical Testing of Parallel Random Number Generators 1998) et (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003) font état de quelques uns d'entre eux. Parmi les tests pour les générateurs de nombres pseudo-aléatoires parallèles basés sur les applications, on peut citer : le test utilisant le modèle d'Ising (P. Coddington, Tests of random number generators using Ising model simulations 1996b), le test utilisant des marches aléatoires (Vattulainen, Framework for testing random numbers

in parallel calculations 1999) ou les tests basés sur des calculs de probabilités présentés dans (Liang et Withlock 2001).

Enfin une méthode différente est proposée dans (De'Matteis et Pagnutti, Controlling correlations in parallel Monte Carlo 1995). Elle permet de contrôler l'influence de la corrélation inter-séquence sur les résultats d'une simulation stochastique distribuée selon l'approche MRIP en cours d'exécution. Elle consiste à étudier l'évolution conjointe des variances de quelques résultats de chacun des processus. Un estimateur permet ensuite de savoir si la corrélation entre les séries utilisées par chaque processus affecte les résultats ou non. Elle nécessite cependant qu'un système de points de synchronisation entre les réplications soit mis en place.

Ce manuscrit présente les batteries : RNGTS, Diehard (G. Marsaglia 1985) de George Marsaglia, DieHarder, DieHarder une évolution de DieHard, TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007) de Pierre L'Ecuyer et la batterie présente dans la bibliothèque de génération parallèle de nombres aléatoires SPRNG (Mascagni et Srinivasan, Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation 2000). On trouve d'autres batteries de tests moins importantes qui ne sont pas présentées ici : un ensemble de tests est disponible sur la page internet de I. Vattulainen[32] et qui est ENT[33] une batterie de tests embryonnaire comprenant seulement 5 tests.

Je ne présente pas non plus les batteries de tests pour les générateurs de nombres pseudo-aléatoires cryptographiques. Ces batteries recherchent des régularités ou des défauts dans des suites de bits générées. Les générateurs actuellement utilisés en simulation sont linéaires, ce qui induit une structure au niveau des bits générés. Un bon générateur pour la simulation pourra ainsi être rejeté par ces batteries. Parmi elles on trouve la batterie STS (Statistical Test Suite) du NIST (National Institute of Standards and Technology) (Soto 2000) ou Crypt-XS[34].

IV.1.2 DieHard

Diehard [35] est une batterie de tests pour générateurs de nombres pseudo-aléatoires écrite en fortran. Chaque test permet de détecter un défaut statistique particulier dans

[32] http://www.helsinki.fi/~vattulai/rngs.html
[33] http://www.fourmilab.ch/random/
[34] http://www.isi.qut.edu.au/resources/cryptx/
[35] http://stat.fsu.edu/pub/diehard/

une séquence de bits. La version la plus complète de Diehard est composée de 18 tests indépendants. Les résultats des tests sont des p-valeurs comprises entre 0 et 1. Un test est considéré comme échoué si cette valeur est très proche de 1 (par exemple $p > 0,95$) ou de 0 (par exemple $p < 0,05$).

DieHard a été le standard à la fin du précédent millénaire en matière de test de nombres pseudo-aléatoires. Cette batterie comporte cependant des défauts majeurs (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007). Premièrement, la séquence de tests à effectuer sur un générateur ainsi que les paramètres des tests sont fixés dans le package.

Deuxièmement, la taille des séquences testée est faible. L'intégralité des tests est exécutée en une poignée de secondes même sur un ordinateur personnel de faible puissance correspondant aux standards actuels alors que les simulations stochastiques prennent des heures voire des années d'exécution. DieHard permet de se faire rapidement une idée de la qualité d'un générateur de nombres pseudo-aléatoires mais ne teste pas des portions du cycle de génération de longueur significative par rapport à celles utilisées couramment en simulation.

Troisièmement, Diehard est conçue pour tester des générateurs de nombres avec une résolution de 32 bits, les générateurs avec une définition inférieure seront donc systématiquement rejetés.

Enfin, la suite de bits à tester doit être placée dans un fichier binaire stocké sur un disque dur. Cette solution est utilisable du fait de la faible quantité de nombres pseudo-aléatoires consommée par la batterie, mais elle est intrinsèquement lente et consomme de l'espace de stockage.

IV.1.3 DieHarder

DieHarder[36] est une batterie de tests sous licence GPL (Gnu Public License) écrite en C. Elle se présente sous la forme d'une bibliothèque et d'un outil en ligne de commande. Son nom a été choisi en hommage à la batterie de tests de Marsaglia. Elle implémente tous les tests de DieHard en y ajoutant la possibilité de les paramétrer, ainsi que des tests issus de STS (Statistical Test Suite) et des tests développés par l'auteur Robert G. Brown.

[36] http://www.phy.duke.edu/~rgb/General/dieharder.php

Dans DieHarder, les principaux défauts de DieHard ont été comblés :

- Il n'y a ainsi plus besoin de fichier intermédiaire entre la batterie de tests et le générateur.
- Les tests eux-mêmes ont été améliorés.
- Le nombre de p-valeurs utilisées pour le test final d'uniformité de Kolmogorov-Smirnov a été grandement augmenté.
- Dans la mesure du possible, le nombre d'échantillons du générateur de nombres pseudo-aléatoires pris en compte par un test est passé en paramètre.

L'outil en ligne de commande permet de tester facilement tous les générateurs de la GSL (GNU Scientific Library). Pour tester d'autres générateurs, l'utilisateur doit utiliser un fichier intermédiaire ou l'API (Application Programming Interface) de la bibliothèque de test en C.

IV.1.4 RNGTS

RNGTS est présentée dans le mémoire (Rütti, A Random Number Generator Test Suite for the C++ Standard. Diploma Thesis. 2004). Comme le montre le Tableau 3 elle comprend un large nombre de tests provenant de diverses batteries implémentées ainsi que des tests publiés dans plusieurs publications scientifiques. Elle implémente : les tests de Knuth (D. E. Knuth, The art of computer programming, Seminumerical Algorithms 1997), le model physique de Ferrenberg (Ferrenberg, Landau et Wong, Monte Carlo simulations: Hidden errors from "good" random number generators 1992), les tests conçus par Vattulainen (Vattulainen, Ala-Nissila et Kankaala, Physical models as tests of randomness 1995), les tests Diehard, d'autres tests conçus plus récemment par G. Marsaglia (Marsaglia et Tsang 2002), des tests pour les générateurs de nombres pseudo-aléatoires parallèles provenant de SPRNG (Mascagni et Srinivasan, Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation 2000) et les tests publiés dans (Gonnet 2003) et dans (Maurer 1992).

Cette bibliothèque est implémentée en C++. Elle produit des fichiers résultats en XML. Elle fournit de plus la possibilité de tester des générateurs parallèles en intercalant les tirages des nombres de plusieurs générateurs de nombres pseudo-aléatoires par

intercalage (Srinivasan, Mascagni et Ceperley, Testing parallel Random Number Generators 2003) (voir paragraphe IV.1.1).

Tableau 3 Détail et provenance des tests de la batterie RNGTS

Provenance	Tests
Knuth	Equidistribution Test (Frequency Test), Gap Test, Serial Test, Poker Test (Partition Test), Run Test, Maximum-of-t Test, Collision Test (Hash Test), Serial correlation Test
Ferrenberg	Ising Model Test
Vattulainen	n-block Test, 2-d Random Walk, Random Walkers on a line (S_n Test), 2D Intersection Test, 2D Height Correlation Test
Diehard	Birthday-Spacing's Test, Overlapping Permutations Test, Ranks of 31x31 and 32x32 matrices Test, Ranks of 6x8 matrices Test, Monkey Test on 20-bit Words, Monkey Tests OPSP, OQSO, DNA, Count the 1's in a Stream of Bytes, Count the 1's in Specific of Bytes, Parking Lot Test, Minimum Distance Test, Random Sphere Test, The Squeeze Test, The Craps Test
Marsaglia	Gorilla Test, GCD Test
SPRNG	Sum of distributions (for parallel streams), FFT, Blocking Test
Gonnet	Repeating Time Test (Repetition Test)
Maurer	Maurer's Universal Test

IV.1.5 La batterie de tests de SPRNG

SPRNG (Mascagni et Srinivasan, Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation 2000) est une bibliothèque de génération de nombres pseudo-aléatoires parallèles. En plus des générateurs, elle implémente les tests de Knuth et de Diehard et plusieurs tests pour rechercher des corrélations inter-flux dans les générateurs de nombres pseudo-aléatoires parallèles.

Elle offre la possibilité d'intercaler les flux séquentiels par « interleaving » et de tester la séquence obtenue (test d'intercalage). Elle implémente également deux tests intrinsèquement dédiés aux générateurs de nombres pseudo-aléatoires parallèles. Le « blocking test », qui fait la somme de plusieurs distributions aléatoires de nombres indépendants et la compare à une distribution gaussienne. Enfin, elle implémente le test de la transformée de Fourier, ou une transformée de Fourier rapide est appliquée sur une matrice dont chaque ligne est une suite de nombres générée avec des flux différents.

IV.1.6 TestU01

TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007) est une suite de tests programmée en C. Elle comporte des générateurs de nombres pseudo-aléatoires, des tests statistiques et empiriques, et des batteries de tests prédéfinies. Ces batteries définissent des suites de tests de différentes intensités :

- La plus petite batterie, « SmallCrush », comporte 10 tests. Il ne faut que quelques secondes pour l'exécuter entièrement sur un processeur actuel. Elle permet de se faire une idée de la qualité du générateur testé.
- La batterie « Crush » comporte 96 tests et s'exécute en environ une heure.
- La plus importante batterie de TestU01 est appelé « BigCrush ». Elle comprend 106 tests et son exécution nécessite plusieurs heures de calcul sur un processeur actuel.

Les autres batteries de TestU01 sont : « Rabbit » pour le test de séquences de bits, « Pseudo-DIEHARD » une batterie comportant la plupart des tests de DieHard, « FIPS-140-2 », une implémentation des tests préconisés par le NIST (National Institute of Standards and Technology) pour le test des générateurs de nombres-pseudo aléatoires cryptographiques. Cette libraire fournit enfin la possibilité de tester des générateurs de nombres pseudo-aléatoires parallèle par intercalage des flux (interleaving).

IV.1.7 Test basé sur l'utilisation de plusieurs générateurs de nombres pseudo-aléatoires

En 1992, un article (Ferrenberg, Landau et Wong, Monte Carlo simulations: Hidden errors from "good" random number generators 1992) paru dans une revue de Physique montre qu'en utilisant un très bon générateur de nombres pseudo-aléatoires pour l'époque et largement testé, les résultats de simulation obtenus sont biaisé. Ce biais est dû à de faibles corrélations dans la séquence de nombres générés, non détectés par les tests de l'époque. Cet article est appelé depuis « l'affaire Ferrenberg » (Rütti, Troyer et Petersen, A Generic Random Number Generator Test Suite 2004) et la simulation de Ferrenberg est devenue un test standard pour les générateurs de nombres pseudo-aléatoires (P. Coddington, Tests of random number generators using Ising model simulations 1996b).

Le fait qu'un générateur passe avec succès tous les tests statistiques communément utilisés n'apportent aucune garantie quant à la qualité des résultats d'une simulation qui l'utilise. Le meilleur des tests est encore de tester une simulation avec des générateurs de nombres pseudo-aléatoires de natures différentes et d'en comparer les résultats, comme le fait remarquer Pierre L'Ecuyer dans (L'Ecuyer, 1998).

IV.1.8 Récapitulatif

Le Tableau 4 présente un récapitulatif des caractéristiques des batteries de tests de générateurs de nombres pseudo-aléatoires présentées dans cette partie.

Tableau 4 Récapitulatif des caractéristiques des batteries de tests de générateurs de nombres pseudo-aléatoires

Nom de la batterrie	Langage	Nombre de tests implémentés	Nombre de tests parallèles disponibles
DieHard	Fortran	18	0
DieHarder	C	24	0
RNGTS	C++	34	4
SPRNG	C++	13	4
TestU01	C	Plus de 100	1

Parmi les batteries de tests présentées dans cette partie, les plus récentes permettent de tester les générateurs de nombres pseudo-aléatoires parallèles. Plusieurs bibliothèques proposent de tels générateurs. Celles-ci sont présentées dans la prochaine partie.

IV.2 Outil de génération de séquences parallèles de nombres pseudo-aléatoires

IV.2.1 rStream

L'idée d'une interface unifiée orientée objet pour la génération parallèle de flux de nombres pseudo-aléatoires a été présentée dans (L'Ecuyer, Simard, et al. 2002). Elle a ensuite été implémentée dans la bibliothèque rStream (L'Ecuyer et Leydold, rstream: Streams of random numbers for stochastic simulation 2005). rStream est intégrée dans

le logiciel de calcul statistique R [37] et constitue l'ensemble des classes dédiées à la génération de nombres pseudo-aléatoires. L'idée centrale de cette bibliothèque est de conceptualiser les flux de nombres pseudo-aléatoires comme des objets « stream » composés de multiples sous-flux ou « substream ».

Un seul algorithme de génération est implémenté dans cette bibliothèque. C'est un générateur à récurrences multiples sur 32 bits, parallélisé selon l'approche du découpage en séquence (cf. Chapitre 1) en utilisant une technique de division de cycle. Les flux correspondent à des séquences de nombres espacées d'un nombre z de tirages. Chaque flux est ensuite partitionné en v sous-flux espacés de w tirages[38].

IV.2.2 SSJ

SSJ (Stochastic Simulation in Java) (L'Ecuyer et Buist, Simulation in Java with SSJ 2005) est une bibliothèque pour simulation stochastique écrite en java et implémentée par P. L'Ecuyer. La partie génération de nombres pseudo-aléatoires parallèles de SSJ conceptualise les flux de nombres aléatoires de la même manière que dans rStream. Elle est plus riche que rStream et comporte plusieurs générateurs de nombres pseudo-aléatoires. Pour certains d'entre eux, aucune technique de division de cycle n'est disponible, comme pour le générateur Mersenne Twister 19937. L'instanciation des sous-séquences est alors effectuée en tirant au hasard une position dans le cycle de génération par la technique dite des séquences indexées. Dans ce cas le non chevauchement des sous-séquences n'est pas garanti. La distance minimale entre n statuts générés de la sorte est de $1 / n^2$ fois la période du générateur cible en moyenne (Wu et Huang 2006). Dans ce cas, les chevauchements sont hautement improbables du fait de la période de 2^{19937} tirages du générateur.

IV.2.3 SPRNG

SPRNG, qui est présentée dans sa version 1.0 dans (Mascagni, Ceperley et Srinivasan, SPRNG: A Scalable Library for Pseudorandom Number Generation, 2000), est une bibliothèque de génération de nombres pseudo-aléatoires en environnement distribué. Cette bibliothèque inclut :

- Plusieurs générateurs portables et testés.
- Une initialisation sans communications inter-processeur.

[37] http://www.r-project.org
[38] http://www.iro.umontreal.ca/~simardr/ssj/doc/pdf/guiderng.pdf

- Une reproductibilité en utilisant les paramètres pour discriminer les flux.
- Une reproductibilité en utilisant un unique germe global.
- Une minimisation des corrélations inter-processeurs grâce aux générateurs utilisés.
- Une interface uniforme en C, C++, FORTRAN et MPI.
- Une bonne extensibilité.
- Une suite de tests intégrée.

La technique de parallélisation des générateurs utilisée dans cette bibliothèque est la paramètrisation. Les autres méthodes ne sont délibérément pas prises en compte. SPRNG implémente des générateurs linéaires congruentiels, des générateurs à registres à décalages bouclés, des générateurs à récursions multiples combinés et des générateurs de Fibonacci décalés. La version 4.0 est aujourd'hui disponible en C++ ou en Fortran[39].

IV.2.4 JAPARA

JAPARA (Coddington et Newell, JAPARA – A Java Parallel Random Number Library for High-Performance Computing 2004) est une bibliothèque implémentée pour étendre la librairie standard de java et notamment le paquetage `java.util.random` à la génération de séquences de nombres pseudo-aléatoires parallèles.

Les classes de génération de nombres pseudo-aléatoires du JDK (Java Development Toolkit) sont synchronisées. Elles sont ainsi utilisables dans un contexte multithread. Cependant ce type d'utilisation d'un générateur aléatoire séquentiel présente tous les désavantages liés à l'approche de parallélisation dite « serveur central » présentée dans la partie III.3.1 du chapitre 1 de ce manuscrit. Pour palier à ce manque, la bibliotèque JAPARA propose trois générateurs de nombres pseudo-aléatoires parallèles implémentant l'API commune aux générateurs de nombres pseudo-aléatoires du paquetage de développement de java :

1. Un générateur congruentiel linéaire avec un modulo premier, recommandé dans (L'Ecuyer, Blouin et Couture, A Search for Good Multiple Recursive Generators 1993).

[39] http://sprng.cs.fsu.edu

2. Un générateur à récursions multiples développé par L'Ecuyer (L'Ecuyer, Combined Multiple Recursive Generators 1996) (L'Ecuyer, Good Parameter and Implementations for Combined Multiple Recursive Random Number Generators 1999) et utilisé dans rStream.
3. Un générateur de Fibonacci décalé multiplicatif avec grand décalage présenté dans (Coddington, Mathew et Hawick, Interfaces and Implementations of Random Number Generators for Java Grande Applications 1999).

Les générateurs 1 et 2 sont parallélisés en utilisant une approche de découpage en séquence. Le troisième est parallélisé selon la technique des séquences indexées.

JAPARA permet la sauvegarde de l'état d'un générateur de nombres pseudo-aléatoires dans un fichier. Cependant il est conçu pour une utilisation dans laquelle plusieurs processus légers partagent un espace mémoire. Et il ne fournit pas de fonctionnalité pour initialiser facilement des générateurs qui l'exécutent sur des machines différentes.

JAPARA se conforme à l'API (Application Programming Interface) de la classe `java.util.Random` conçue pour la génération de séquences de nombres pseudo-aléatoires en séquentiel. La génération des séquences de nombres pseudo-aléatoires parallèles utilise un mécanisme de germes. Celui-ci ne permet pas de changer la méthode de parallélisation appliquée à un générateur séquentiel et limite l'utilisateur aux trois générateurs parallèles implémentés dans la bibliothèque.

IV.2.5 Des générateurs de nombres pseudo-aléatoires parallèles

En plus des bibliothèques de générateurs de nombres pseudo-aléatoires parallèles il existe des générateurs de nombres pseudo-aléatoires conçus, et implémentés, pour générer des nombres pseudo-aléatoires parallèles.

Parmi eux, on trouve PLFG (Parallel Lagged Fibonacci Generator) (Tan 2002), un générateur de nombres pseudo-aléatoires parallèle implémenté au dessus de MPI. Il utilise l'approche de parallélisation par indexation de séquence d'un générateur séquentiel de Fibonacci décalé recommandé par (D. E. Knuth, The art of computer programming, Seminumerical Algorithms 1997). Le générateur séquentiel présente une période de $2^{29} \times (2^{23209} - 1)$ tirages sur une architecture d'ordinateur 32 bits et de $2^{61} \times (2^{23209} - 1)$ tirages sur une architecture 64 bits. L'indexation des séquences

utilise le générateur Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) pour remplir la table d'historique de 23209 nombres du générateur de Fibonnaci décalé. Il est ainsi possible de générer $\frac{2^{19937}-1}{23209}$ séquences de nombres pseudo-aléatoires parallèles avec PLFG. Les auteurs présentent le fait que la période du générateur augmente avec l'augmentation de la taille des registres des machines comme une caractéristique « intéressante » de leur implémentation. Cependant le fait que le générateur ne produise pas les mêmes séquences de nombres sur différentes architectures pose problème pour son utilisation dans un environnement de calcul distribué hétérogène.

Un autre algorithme de génération de nombres pseudo-aléatoires en parallèle est basé sur le générateur Mersenne Twister. Ce générateur est de type *« Twisted Generalised Shift Feedback Register »*, un type particulier de générateur à registre à décalage bouclé. L'article (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000) présente un algorithme de création de générateur de type Mersenne Twister (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997). Cet algorithme permet de générer des paramètres pour un générateur de nombres pseudo-aléatoires de type Mersenne Twister générique. Les périodes des générateurs paramétrés s'échelonnent de 2^{521} à 2^{44497}. L'implémentation en C de l'algorithme[40] permet de paramétrer jusqu'à 65536 générateurs pseudo-aléatoires générant des séquences de nombres hautement indépendantes en théorie.

IV.2.6 Récapitulatif

Le Tableau 5 récapitule les caractéristiques des bibliothèques de génération de nombres pseudo-aléatoires parallèle présentées dans cette partie.

[40] http://www.math.sci.hiroshima-u.ac.jp/~m-mat

Tableau 5 Récapitulatif des caractéristiques des bibliothèques de génération de nombres pseudo-aléatoires parallèle

Nom de la bibliothèque	Version actuelle	Nombre de générateurs	Langage	Utilisation de MPI	Méthodes de parallèlisation
rStream	1.2.2	1	Java	Non	Découpage en séquence
SSJ	2.0	13	Java	Non	Découpage en séquence, Séquences indéxées
SPRNG	4.0	5	C, C++ et Fortran	Oui	Paramétrisation
JAPARA	-	3	Java	Non	Découpage en séquence, Séquences indéxées

Les bibliothèques de génération de nombres pseudo-aléatoires parallèle permettent d'exécuter et de concevoir des simulations stochastiques distribuées. Pour faciliter la tâche des modélisateurs, plusieurs projets logiciels ont été développés pour distribuer automatiquement des simulations selon l'approche MRIP. Ces projets ont des objectifs différents et proposent différentes fonctionnalités comme : la prise en charge de la parallélisation du générateur de nombres pseudo-aléatoires ou la gestion d'un environnement complet d'exécution distribuée. Ces environnements sont présentés dans la partie suivante.

IV.3 Logiciel de distribution automatique des réplications

IV.3.1 Akaroa

Akaroa (Pawlikowski et Yau, AKAROA: a Package for Automatic Generation and Process Control of Parallel Stochastic Simulation 1993) est un progiciel pour la simulation stochastique parallèle à état stationnaire ou terminante. Akaroa permet la parallélisation de simulations séquentielles, dans le respect de la rigueur statistique avec une analyse appropriée des données de sortie. Il permet par exemple d'arrêter automatiquement tous les processus parallèles de la simulation une fois l'état stationnaire atteint. Il a été conçu sous la direction de K. Pawlikowski et développé pendant plus de 10 ans. Il fonctionne sous UNIX sur des systèmes multiprocesseurs avec un nombre de processeurs illimité en théorie.

L'architecture d'Akaroa est présentée Figure 24. Ce logiciel est basé sur une architecture maître-esclaves. Le maître prend en compte les simulations et distribue les réplications aux processus esclaves exécutés sur des machines connectées à un réseau local. Les communications interprocessus utilisent le protocole TCP / IP.

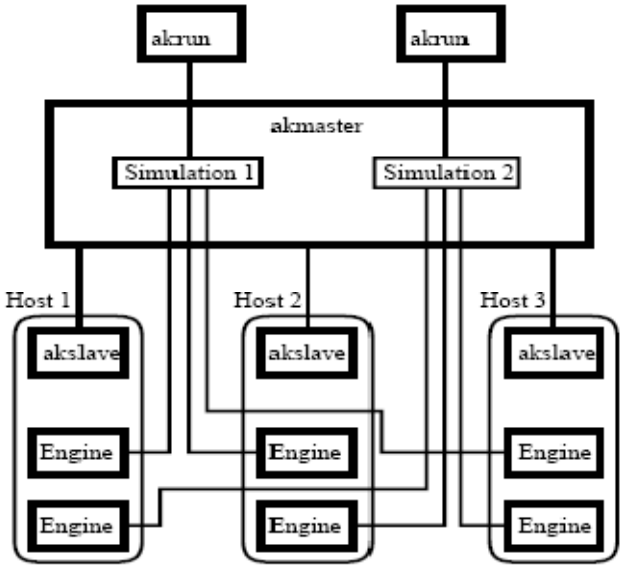


Figure 24 Extrait de (Ewing, Pawlikowski et McNickle 1999) Architecture d'Akaroa

La Figure 25 présente le traitement des données statistiques de simulation dans Akaroa. Chaque processus analyse séquentiellement les données résultats et calcule un estimateur local de performance. A intervalle régulier, un processus esclave envoie ses estimateurs locaux à un analyseur global. Les mesures statistiques provenant des différents processus esclaves sont alors combinées pour donner un ensemble d'estimateurs globaux.

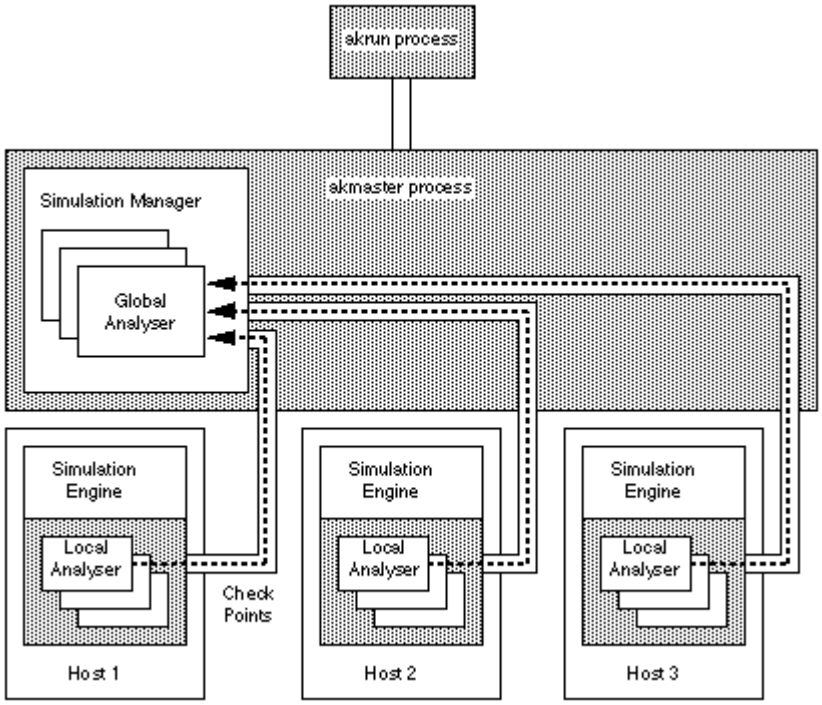


Figure 25 [41] Traitement des données statistiques dans Akaroa

Akaroa gère la génération de nombres pseudo-aléatoires en parallèle. La version 2 (Ewing, Pawlikowski et McNickle 1999) utilise 25 générateurs multiplicatifs congruentiels avec un modulo égal à 2^{31}-1 pris au sommet de la liste des 200 générateurs recommandés dans (Fishman et Moore, An exhaustive analysis of multiplicative congruential generators with modulus M=2^31-1 1986), ainsi que 25 générateurs dont les multiplicateurs sont les inverses (modulo M) des premiers. Ces générateurs sont combinés par le processus maître pour générer une séquence de 10^{11} nombres pseudo-aléatoires.

La distribution des nombres se fait selon une approche « serveur central », en distribuant des parties de la séquence à chaque processus. L'ensemble des nombres de la séquence globale ne sont pas envoyés directement. Le serveur central est chargé d'allouer des blocs de nombres à chaque processus esclave. Il envoie les paramètres de génération des blocs alloués aux processus esclaves sur demande. Les sous-séquences sont ensuite générées localement par chaque processus esclave.

Les générateurs de nombres pseudo-aléatoires utilisés sont obsolètes. De plus cette solution présente les désavantages liés à la distribution de type « serveur central ». Les sous-séquences sont allouées dynamiquement, il est donc difficile de répéter à

[41] http://www.cosc.canterbury.ac.nz/research/RG/net_sim/simulation_group/akaroa/architecture.chtml

l'identique une exécution donnée. Enfin, le serveur peut devenir un goulot d'étranglement lors d'une utilisation importante de nombres pseudo-aléatoires par les processus esclaves.

IV.3.2 EcliPSe

D'après (Knop, Mascarenhas, et al. 1994), EcliPSe est une application de distribution de simulations selon l'approche MRIP. Elle fait partie intégrante de la suite logicielle ACES, qui comprend :

- ParaSol : Une bibliothèque de distribution de simulations à événements discrets selon l'approche SRIP.
- CONCH : Une bibliothèque de passage de messages en environnement hétérogène.
- Ariadne : Une bibliothèque de gestion de processus légers modulaire, permettant la migration de threads d'un processus à un autre.

D'après (Knop et Rego, Parallel Cluster Labeling on a Network of Workstations 1995), EcliPSE est conçu pour fournir à l'utilisateur :

- Simplicité : peu de modifications sont nécessaires pour faire d'une application séquentielle une application permettant d'exécuter des réplications de manière distribuée.
- Flexibilité : en plus de la distribution de réplications, EcliPSe permet le calcul parallèle avec des communications interprocessus.
- Portabilité : une application peut être distribuée sur des machines avec des architectures hétérogènes reliées entres elle par un réseau étendu.
- Extensibilité : des mécanismes réduisant les goulots d'étranglement dus à la sérialisation permettent aux applications d'être exécutées sur un nombre important de processeurs.
- Tolérance aux fautes : EcliPSe à la capacité de se remettre de l'écroulement de plusieurs machines participant au calcul et d'autres problèmes liés à l'exécution d'applications de longue durée.

D'après ses concepteurs, le progiciel EcliPSe permet d'exécuter les réplications d'un même algorithme en parallèle avec des données différentes (à la MPI). Ce qui est

surprenant, c'est que ce logiciel ne prend pas en compte la génération de flux de nombres pseudo-aléatoires parallèles pour la distribution des réplications d'une simulation stochastique.

IV.3.3 ASDA

ASDA (Bruschi, et al. 2004) est un environnement de développement qui permet de distribuer automatiquement le programme de simulation selon la méthode SRIP, MRIP ou SRIP et MRIP simultanément.

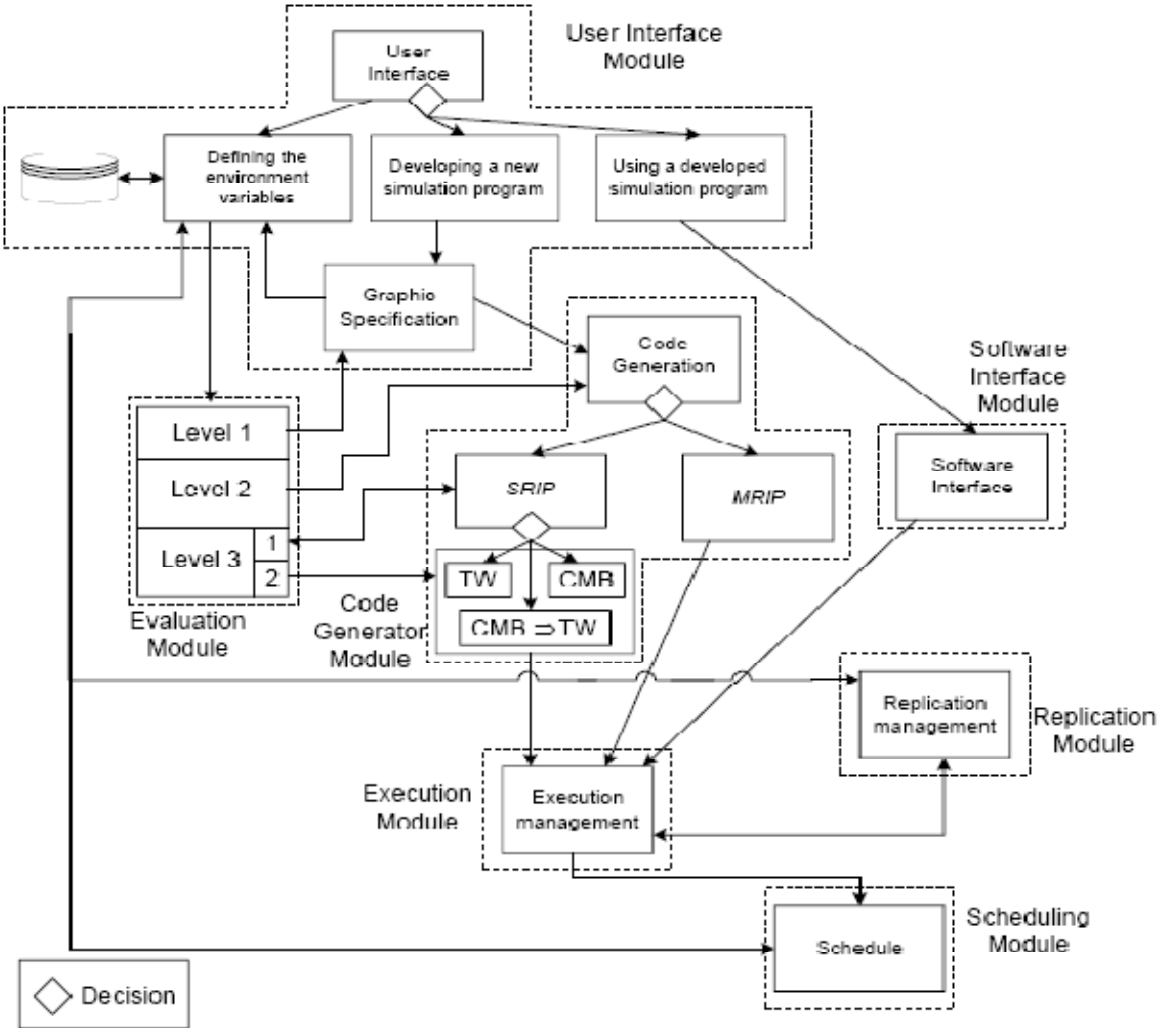


Figure 26 Extrait de (Bruschi, et al. 2004) : Architecture de ADSA

Comme le montre la Figure 26, cet environnement propose une approche ingénierie des modèles. Une interface graphique (Figure 27) permet ainsi de concevoir un modèle. ASDA génère ensuite le code d'exécution parallèle. Il assiste alors l'utilisateur dans le choix de la méthode de parallélisation. Enfin il gère l'exécution à l'aide d'un

ordonnanceur et contrôle le nombre de réplications afin d'obtenir des résultats statistiques corrects.

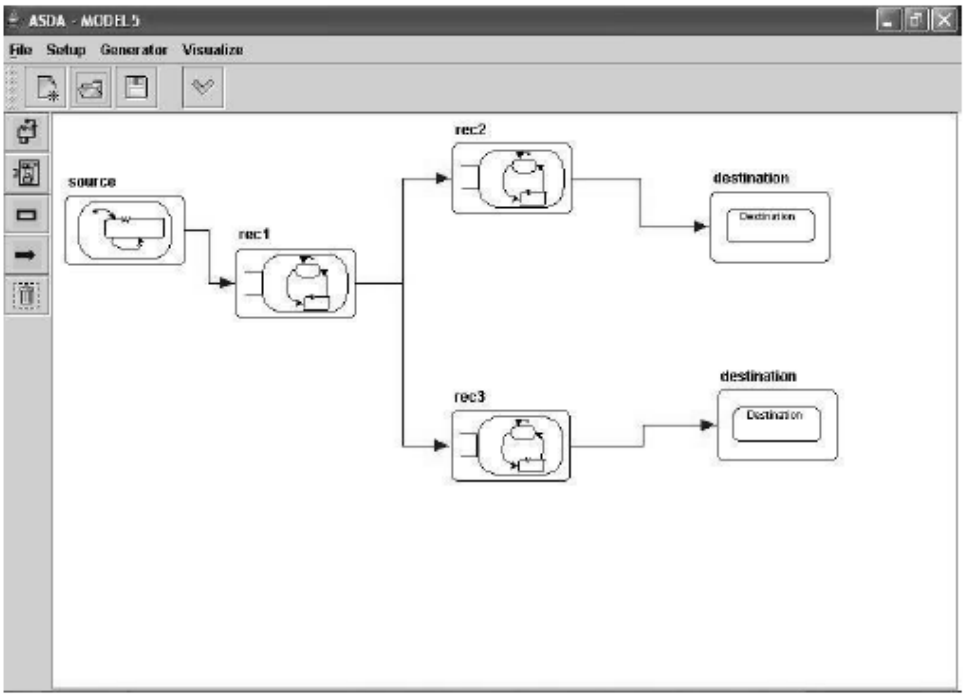


Figure 27 Extrait de (Bruschi, et al. 2004) : Interface graphique de ADSA

Les avantages d'ADSA sont les suivants :

- Il offre un environnent facile à appréhender et à utiliser.
- Il permet la génération complète de programmes de simulation distribuée par un utilisateur inexpérimenté grâce à une approche ingénierie des modèles.
- Il offre la flexibilité pour les utilisateurs plus avisés de modifier les programmes générés.
- Il autorise l'utilisation de programmes de simulation séquentiels déjà développés.
- Il guide l'utilisateur dans son choix entre différentes approches de simulation distribuée.
- Il rend facile l'obtention de données crédibles.
- Il minimise le temps de simulation en offrant un ordonnancement efficace.

La publication (Bruschi, et al. 2004) fait très peu cas de la génération de flux de nombres pseudo-aléatoires. Elle présente simplement les réplications d'une simulation

stochastique comme des exécutions du même programme séquentiel basées sur des « germes » différents pour le générateur de nombres pseudo-aléatoires. Il semble que la génération parallèle de nombres pseudo-aléatoires ne soit pas prise en compte par cette application. Elle n'est à mon sens clairement pas satisfaisante pour la parallélisation des simulations stochastiques.

IV.3.4 PMCD

PMCD (Parallel Monte Carlo Driver) (Mendes et Pereira, Parallel Monte Carlo Driver (PMCD)-a software package for Monte Carlo simulations in parallel 2003) utilise MPI pour distribuer l'exécution de simulation de type Monté-Carlo sur des machines sous UNIX. Ce logiciel est écrit en Fortran. Il prend en charge la génération des fichiers d'entrée ainsi que la récupération des fichiers de sortie pour chaque réplication. Il permet l'exploration de paramètres et la génération de scenarios.

Comme le montre la Figure 28, le `Bayesian Layer` choisit un scénario d'exécution parmi ceux indiqués par l'utilisateur. L'`Input Preparator` génère des données d'entrée du modèle de manière aléatoire selon les lois de probabilité indiquées par l'utilisateur et en fonction du scénario. Le `Workload Controller` distribue l'exécution de tâches sur les nœuds. Chaque jeu de données d'entrée est ainsi exécuté indépendamment des autres.

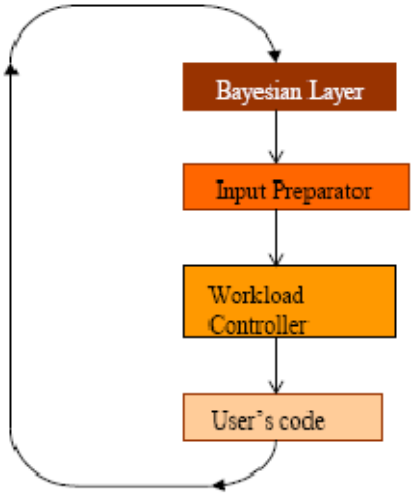


Figure 28 Extrait de (Mendes et Pereira, Software Manual for the Parallel Monte Carlo Driver 2007) : Exécution d'une simulation par PMCD

La génération de séquences de nombres pseudo-aléatoires non-corrélées est prise en charge par le logiciel. Chaque nœud d'exécution se voit affecté une sous-séquence d'un générateur congruentiel linéaire parallélisé par saut de grenouille (leap frog). La formule récursive des générateurs utilisés est :

$$X_{k+1} = a^n R^k mod\ m$$

Ou a est le terme multiplicatif, n est l'identifiant du nœud (de 1 au nombre de nœuds) et m le modulo. Le terme additif est nul. Cette solution utilise ainsi un générateur linéaire congruentiel de nombres pseudo-aléatoires de faible qualité. Les corrélations longues-portées de ce générateur peuvent devenir des corrélations inter-séquences lors de l'exécution parallèle des réplications. De plus PMCD ne permet pas à l'utilisateur de choisir ni le générateur de nombres pseudo-aléatoires ni la méthode de parallélisation. Il est ainsi impossible de mesurer l'effet des corrélations intra et inter-séquences sur les résultats d'une simulation.

IV.3.5 Scripts pour la distribution de simulation selon MRIP

Un ensemble de scripts shell en bash et de programmes en Fortran ont été conçus pour la distribution de simulation stochastique (Badal et Sempau 2006). Ils distribuent les réplications d'une simulation stochastique.

La génération de nombres pseudo-aléatoires en parallèle est implémentée dans un code en Fortran et se base sur l'algorithme RANECU (L'Ecuyer, Efficient and Portable Combined Random Number Generators 1988). Cet algorithme est ancien, mais il présente encore de bonnes propriétés statistiques au vu des standards actuels. L'algorithme est parallélisé par un découpage en séquence en utilisant la technique de division de cycle expliquée dans (L'Ecuyer, Efficient and Portable Combined Random Number Generators 1988). Le code Fortran peut être modifié facilement pour utiliser la parallélisation par sauts de grenouille (leap frog).

L'exécution de la simulation se fait selon une approche maître-esclave. Les réplications de la simulation sont exécutées en utilisant ssh sur les esclaves, qui sont appelés « clones ». Les transferts des fichiers d'entrée et de sortie s'appuient sur scp.

Cette solution comporte trois défauts majeurs. Tout d'abord, elle implémente seulement un algorithme de génération de nombres pseudo-aléatoires et n'offre pas la possibilité d'utiliser facilement différents algorithmes. De plus, la distribution d'une simulation stochastique n'est pas prévue pour des simulateurs qui ne sont pas implémentés en Fortran. Enfin l'exécution distribuée basée sur ssh nécessite un accès réseau direct aux machines cibles et ne permet pas de profiter des architectures de grille ou de fermes de calcul utilisant une exécution par lot de jobs.

IV.3.6 Récapitulatif

Le Tableau 6 présente un récapitulatif sur les environnements de distribution de simulation selon les réplications.

Tableau 6 Récapitulatif pour les environnements de distribution de simulation selon l'approche MRIP

Nom du logiciel	Type de générateur de nombre pseudo-aléatoire	Méthode de parall-élisation	Protocole utilisé pour l'éxécution distribuée
Akaroa	LCGs datant de 1986	Approche serveur central et utilisation d'une liste de générateurs	TCP/IP
EcliPSe	Aucun	Aucune	TCP/IP
ASDA	Non expliqué (certainement absent)	Non expliquée (certainement absente)	TCP/IP
PMCD	LCG sans terme additif	Sauts de grenouille (leap frog)	MPI
Scripts	RANECU	découpage en séquence	SSH/SCP

V Conclusion

Il existe aujourd'hui des plateformes de calcul distribué mûres pour la production de résultats scientifiques à grande échelle comme les grilles de calcul. Diverses méthodes, standards et outils pour la parallélisation sont couramment utilisées pour accélérer des simulations déterministes.

Dans le domaine de la parallélisation des simulations stochastiques, en revanche seul un nombre restreint d'outils permettent la distribution rigoureuse de simulations stochastiques. Parmi eux, on trouve des outils de distribution automatique de simulations stochastiques et des bibliothèques de génération de nombres pseudo-aléatoires parallèles.

Les environnements de distribution automatique de simulations stochastiques existant sont à mon avis inutilisables pour exploiter la puissance d'environnements de calcul fondés sur le paradigme du sac de travail comme les grilles de calcul du fait qu'ils présentent des défauts majeurs :

Ils utilisent soit des communications réseau entre les éléments de calcul qui peuvent être impossibles ou ralentir le temps global de simulation (TCP / IP), soit MPI (Message Passing Interface) ou SSH (Secure SHell) qui ne sont pas utilisables facilement sur grille de calcul.

Alors que la méthode la plus fiable pour détecter un biais induit par l'utilisation d'un générateur de nombres pseudo-aléatoires dans les résultats d'une simulation stochastique est d'exécuter la simulation avec divers générateurs de nombres pseudo-aléatoires parallèles et de comparer les résultats, on ne trouve dans chacun de ces outils qu'un nombre restreint de mécanismes de génération de nombres pseudo-aléatoires parallèles généralement limité à une seule méthode de parallélisation appliquée à un seul générateur de nombres pseudo-aléatoires.

Ces environnements de parallélisation sont intrusifs par rapport au simulateur et nécessitent le changement de l'algorithme de génération de nombres pseudo-aléatoires d'origine du simulateur pour celui proposé par l'outil de parallélisation. Ceci oblige à posséder le code source du simulateur et à le modifier. La nécessité de changer le générateur de nombres pseudo-aléatoires limite de plus la portée de l'outil de parallélisation aux langages de programmation supportés par celui-ci.

Les bibliothèques de génération de nombres pseudo-aléatoires parallèles ne sont pas plus adaptées à l'exécution de simulations stochastiques sur grille de calcul que les outils précédemment cités. En effet, elles facilitent : l'exécution parallèle de simulations stochastiques dans le cadre ou plusieurs processus légers partagent un espace mémoire commun avec le concept de flux et de sous-flux de nombres aléatoires (rStream, SSJ), ou l'exécution parallèle multiprocessus sur des fermes de calcul avec l'utilisation de MPI (SPRNG). Le problème est différent pour la génération parallèle de nombres pseudo-aléatoires sur grille de calcul pour laquelle en pratique chaque job de simulation comporte un fichier d'initialisation pour le générateur de nombres pseudo-aléatoires permettant la génération parallèle de séquences de nombres indépendantes sans communication interprocessus

Certaines bibliothèques comme CLHEP, JAPARA ou SPRNG permettent l'initialisation des générateurs à partir de fichiers. Cependant aucune fonctionnalité n'est fournie à l'utilisateur pour paralléliser un générateur et générer facilement de nombreux fichiers permettant d'initialiser le générateur de nombres pseudo-aléatoires d'un job avec une séquence indépendante de celles utilisées par les autre jobs. De plus aucune traçabilité n'est incluse dans le processus de génération de ces fichiers. Si le test des séquences de nombres pseudo-aléatoires séquentielles est une tâche difficile, celle de tester l'indépendance des séquences de nombres pseudo-aléatoires utilisées par les processus d'une simulation stochastique est impossible en pratique du fait de l'explosion combinatoire du temps de calcul. Il est ainsi important de disposer de mécanismes de parallélisation rigoureux qui permettent une traçabilité des fichiers d'initialisation des générateurs de nombres pseudo-aléatoires produits par le processus de parallélisation.

Enfin, il est important de rendre disponible des outils pour faciliter l'exécution d'une même simulation stochastique avec des flux de nombres pseudo-aléatoires parallèles différents afin de pouvoir mesurer aisément l'impact d'un générateur de nombres pseudo-aléatoires parallèle sur les résultats d'une simulation stochastique distribuée.

Chapitre 3: Propositions

I Introduction

Comme nous l'avons vu dans les chapitres précédents, d'une part la distribution de simulations stochastiques est une tâche complexe, d'autre part les outils actuellement à la disposition des concepteurs de simulateur et des modélisateurs ne sont pas satisfaisants pour la distribution à grande échelle de simulations stochastiques sur des environnements largement distribués comme les grilles de calcul.

Afin de pallier à ces manques, je propose une suite logicielle, appelée Dist, comportant deux composants. Ces composants logiciels sont tous deux basés sur le concept novateur de statut générique pour les générateurs de nombres pseudo-aléatoires parallèles. L'utilisation de statut générique permet de disposer d'un format clair et documenté pour la sauvegarde de l'état d'un générateur séquentiel de nombres pseudo-aléatoires et d'assurer une traçabilité lors de la parallélisation d'un générateur de nombres pseudo-aléatoires séquentiel.

Le premier composant de la suite logicielle proposée est une bibliothèque de génération parallèle de nombres pseudo-aléatoires. Elle vise à externaliser le choix du générateur de nombres pseudo-aléatoires parallèle, du processus de développement d'un simulateur. Avec l'utilisation de cette bibliothèque, la distribution d'un simulateur

stochastique est une étape antérieure à la phase de développement et le choix du générateur de nombres pseudo-aléatoires parallèle utilisé est reporté à l'exécution sans nécessité de modifier le code source du générateur.

Le second composant est un logiciel qui permet la distribution de simulateurs stochastiques selon une approche « boite noire ». Il est ainsi adaptable à tout type de simulateur stochastique, de générateur de nombres pseudo-aléatoires et d'environnements d'exécution distribuée.

Ce chapitre présente la suite logicielle Dist. La première partie présente les aspects généraux de conception de cette suite logicielle. Dans une deuxième partie, je décris le concept central de la suite Dist : les statuts génériques pour les générateurs de nombres pseudo-aléatoires. Ce chapitre présente ensuite dans trois parties séparées les trois composants logiciels de la suite Dist : DistTools la bibliothèque de classes qui fait office de fondation, la bibliothèque de génération de nombres quasi et pseudo-aléatoires distribuée DistRNG et DistMe pour la distribution automatique et rigoureuse de simulations stochastiques.

II Conception de la suite Dist

La suite Dist a été conçue afin de simplifier la distribution de simulations stochastiques en environnement distribué. Cette partie présente la conception générale de cette suite. Elle aborde tout d'abord l'architecture logicielle générale de la suite Dist. Dans un deuxième temps elle questionne l'intérêt pour la communauté scientifique de disposer d'une telle suite. Enfin, elle présente rapidement le domaine de l'ingénierie des modèles sur laquelle la conception de Dist repose.

II.1 Architecture générale

La suite logicielle Dist est conçue pour automatiser la distribution des simulations stochastiques. Le concept central de cette suite est le modèle des algorithmes de génération de nombres pseudo-aléatoires (PRNG) qu'elle utilise. Dans Dist un PRNG est modélisé par l'état de ses paramètres à un instant t et par des métadonnées qui renseignent sur son utilisation en calcul distribué (cf. partie III de ce chapitre).

La Figure 29 présente les cas d'utilisation sous forme de diagramme UML de la suite logicielle Dist. L'utilisateur décrit une simulation stochastique séquentielle, puis lance sa

distribution. Il a alors la possibilité d'utiliser la distribution de plans d'expériences et de paralléliser les réplications indépendantes de cette simulation stochastique. Durant le processus de distribution, l'utilisateur est amené à déterminer le générateur de nombres pseudo-aléatoires parallèle qu'il souhaite utiliser lors de l'exécution de sa simulation. Enfin, il choisit une plateforme de calcul distribuée cible pour l'exécution de la simulation. Des jobs de simulations prêts pour l'exécution sont alors générés automatiquement.

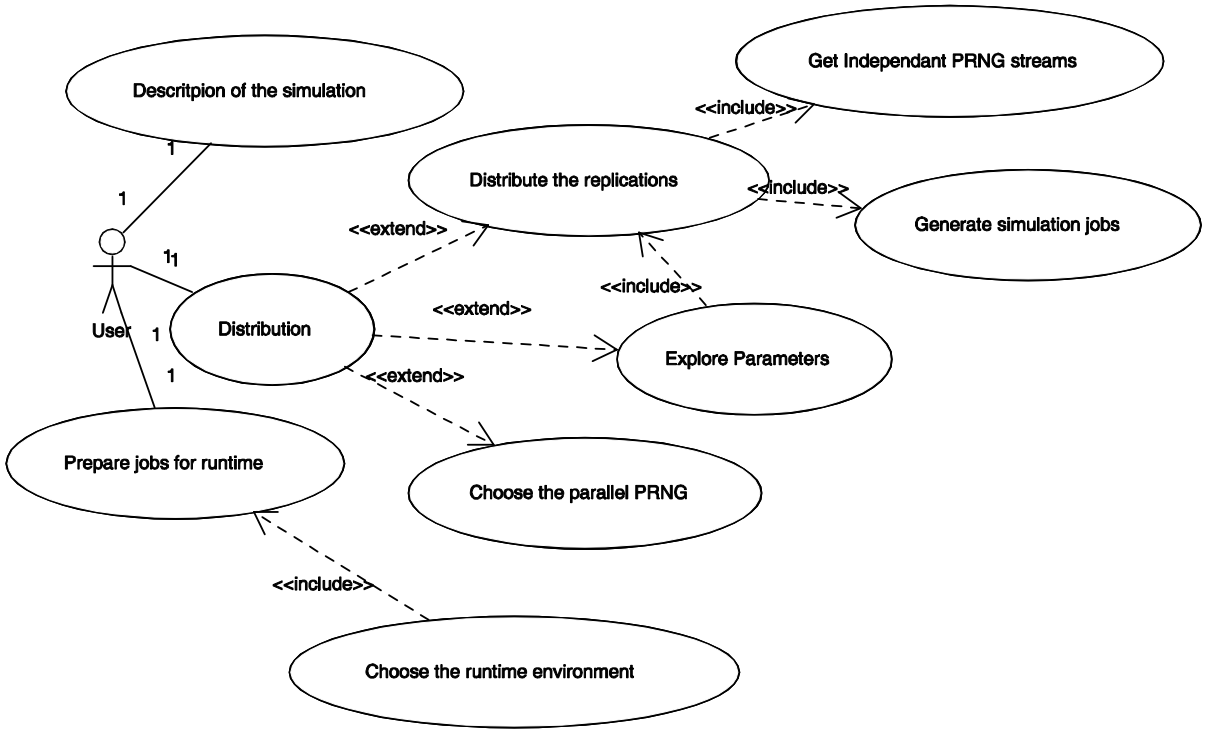


Figure 29 Diagramme UML de cas d'utilisations pour la suite logicielle Dist

L'environnement de distribution Dist est composé de trois progiciels. La Figure 30 représente leurs interdépendances. Le progiciel DistTools regroupe les classes fondations de la suite logicielle Dist. Il en implémente les concepts centraux comme :

- le concept de statuts,
- la sélection de statuts permettant la génération de séquences de nombres pseudo-aléatoires indépendantes dans un dépôt,
- la sérialisation transversale des statuts vers une représentation XML documentée ou des formats propriétaires.

Les classes fondations sont utilisées par DistMe, le progiciel de parallélisation de simulations stochastiques ainsi que par DistRNG, la bibliothèque de génération de nombres quasi-aléatoires et de séquences parallèles de nombres pseudo-aléatoires.

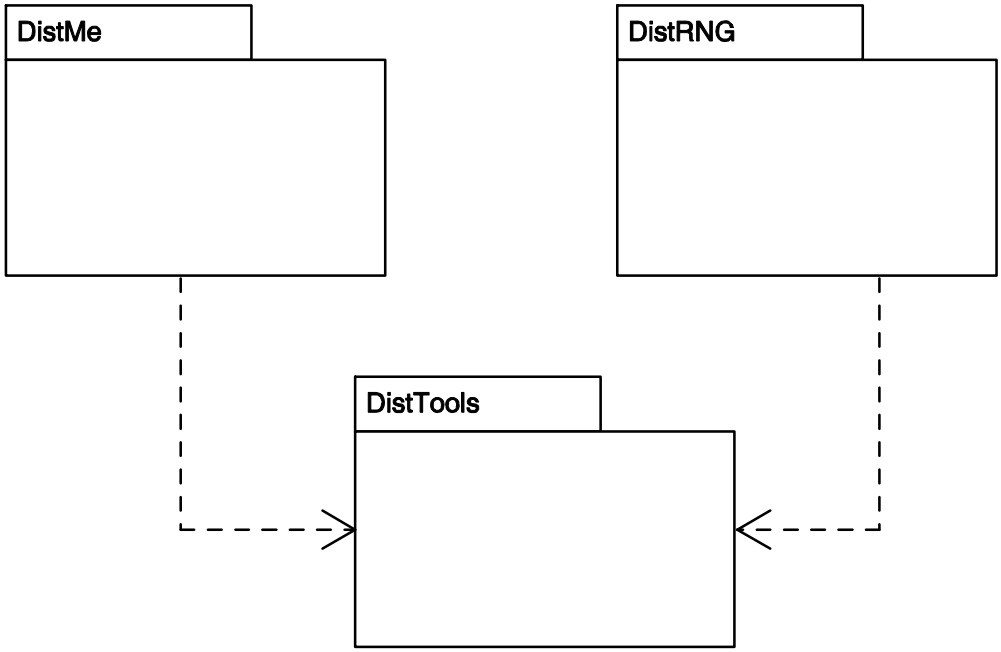


Figure 30 Digramme UML de package de la suite logicielle Dist

II.2 Questionnement sur l'intérêt de la suite logicielle Dist.

Bien que les simulations stochastiques comprenant de nombreuses réplications soient intrinsèquement parallèles, notre expérience dans ce domaine nous a montré que paralléliser une simulation est une tâche complexe. Lors de la soumission d'un article sur la partie distribution de simulations stochastiques appelée DistMe de Dist, un des relecteurs expose son point de vue sur ce progiciel :

« *The paper as it is now is clearly focused and very well explained. It treats a problem that can frequently pass inadverted and that is, as mentioned by the authors, very difficult to spot. The tool described in the article serves to guarantee that no bias is introduced in the random number generator and therefore I think it is quite an interesting job.* »

Cependant, un deuxième soulève une question intéressante à laquelle je souhaite répondre ici sur l'utilité de la suite Dist :

« *I think that the idea of helping a user to split Monte Carlo jobs witch enforces a correct use of the RNGs is a good one. In the paper the authors show the skill of using*

object-oriented techniques however they do not demonstrate what is the added value of creating a complex, object-oriented API for a relatively simple task such as generation of input files for a batch job. Similar functionality, albeit less generic, may be implemented in a few lines of a scripting language, provided that the RNG status files are available on the local disk. »

La principale contribution de DistMe est de permettre la distribution rigoureuse de simulations stochastiques et ce même pour des numériciens non-spécialistes en calcul stochastique distribué. DistMe encapsule la gestion des statuts des générateurs de nombres pseudo-aléatoires et en masque ainsi la complexité à l'utilisateur. Gérer avec quelques scripts un grand nombre de statuts pour différents générateurs peut être source d'erreur et j'en ai fait les frais (voir chapitre 1). L'exécution d'une simulation distribuée sur une grille de calcul nécessite le plus souvent la génération de milliers d'expériences indépendantes et l'automatisation de cette tâche permet d'éviter les erreurs humaines.

Ainsi la suite logicielle Dist offre une approche transversale pour la distribution des simulations stochastiques. Comme nous le verrons dans ce chapitre le développeur qui utilise la partie générateur de nombres pseudo-aléatoires de la suite Dist, n'a pas à se soucier de l'aspect distribué ou non de sa simulation au moment de son développement. Il n'a même pas à choisir l'algorithme de génération de nombres pseudo-aléatoires qu'il souhaite utiliser. Il peut ainsi se concentrer sur la modélisation de son problème. La génération de séquences de nombres pseudo-aléatoires parallèles en accord avec l'état de l'art, le choix du générateur de nombres pseudo-aléatoires et la génération des jobs de simulations indépendants peuvent être effectués dans un deuxième temps, de manière transverse par rapport au simulateur, sans modification du code et pour différents environnements de calcul distribué.

DistMe génère les scripts de lancement des jobs d'une simulation distribuée pour différentes plateformes et masque ainsi une partie de la complexité liée à l'utilisation d'environnements d'exécution parallèle ou massivement parallèle.

II.3 Une approche « ingénierie des modèles »

Cette partie n'est pas un état de l'art dans le domaine de l'ingénierie dirigée par les modèles, cependant le travail de développement logiciel présenté dans ce manuscrit a

été mené selon une approché liée à l'IDM (Ingénierie Dirigée par les Modèles). Je présente donc ici les concepts liés à cette approche.

En novembre 2000, l'OMG (Object Management Group) rend public MDA (Model Driven Architecture) (Soley et Group 2000). MDA est basée sur une utilisation intensive de l'UML (Unified Modeling Language) et place la modélisation au centre du processus de développement (Figure 31) en découplant la logique métier, la logique applicative et le code. Dans MDA, la phase 1 est de construire un PIM (Plateform Independent Model) représentant la logique métier. Dans la phase 2, le PIM est transformé en PSM (Plateforme Specific Model), qui est un modèle comprenant des parties techniques propres à la plateforme cible. La phase 3 est une génération du code applicatif à partir du PSM.

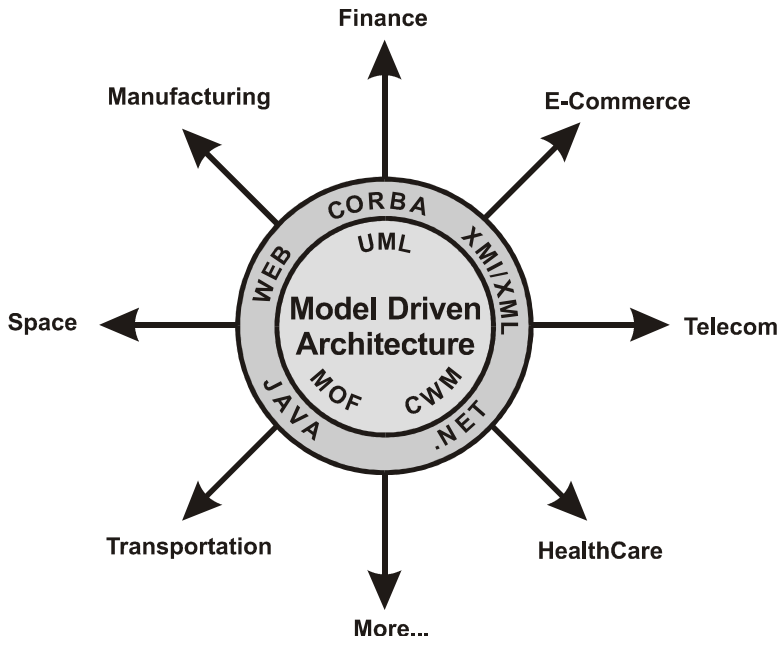


Figure 31 Illustration de MDA[42]

Un article, (Bézevin 2005), explique que dans une vision plus globale, MDA est une approche de l'IDM basée sur les standards de l'OMG. L'IDM est ainsi une vision plus large du processus de développement logiciel que MDA et ne nécessite pas l'utilisation de technologies orientées objet. Ainsi, pour (Greenfield et Short 2003) le tout objet a montré ses limites et le développement logiciel basé sur les modèles est une voie prometteuse : « *We suggest that the current software development paradigm, based on*

[42] http://www.omg.org/mda

object orientation, may have reached the point of exhaustion, and we propose a model for its successor »

Ainsi ces dernières années, des recherches sur les concepts fondateurs de l'IDM sont en maturation. En 2004 on trouvait des papiers titrant :

- « *Towards a Basic Theory to Model Driven Engineering* » (Favre 2004)
- « *In Search of a Basic Principle for Model Driven Engineering* » (J. Bézivin 2004)

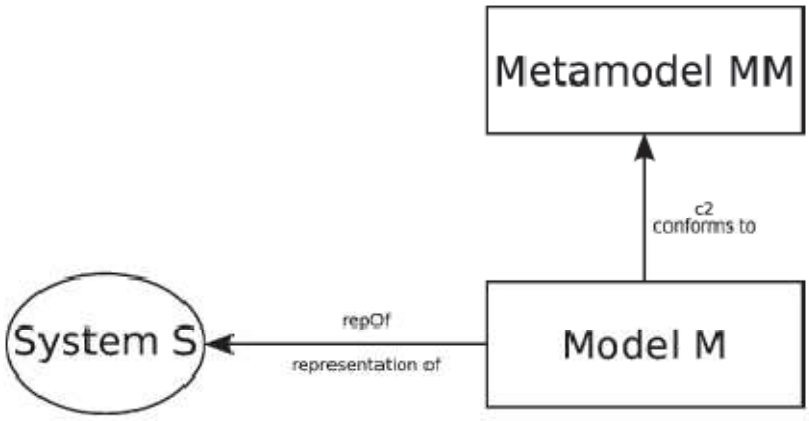


Figure 32 Extrait de (Bézivin, Barbero et Jouault 2007) deux concepts fondateurs de MDE

Depuis, trois ans se sont écoulés et il semble que les deux relations : « conforme à » et « représentation de », illustrées en Figure 32, soient largement acceptées comme faisant partie des concepts fondateurs de l'IDM. Cependant l'IDM n'a bien sûr pas encore atteint toute sa maturité. En effet, en 2007, une nouvelle relation dite « extension de » est introduite dans (Barbero, et al. 2007). De plus, certains auteurs proposent de distinguer la notion de système de celle de modèle. L'article (Bézivin, Barbero et Jouault 2007) conclu par : « *In this very rapidly evolving context, the distinction between systems and models seems necessary to define both the goals and the techniques of MDE* ».

Si l'IDM est une méthodologie en construction, elle présente une approche unificatrice du processus de développement, grâce à l'utilisation importante des modèles et des transformations de modèles. Elle permet un développement souple et itératif (Jézéquel, et al. 2006). Des outils de développement comme GMF (Eclipse Graphical Modelling Framework) (Ehrig, Ermel et Hansgen 2005), GME (Generic Modelling Environment) (Ledeczi, et al. 2001) et AToM^3 (A Tool for Multiformalism Meta-modelling) (Vangheluwe et Lara 2004) intègre l'approche IDM et permettent la

définition de méta-modèles et de transformations. AToM³ permet même la complétion de modèles (Sen, Baudry et Vangheluwe 2007) comme les environnements de conception modernes proposent la complétion de code.

III Statuts génériques

Les générateurs de nombres pseudo-aléatoires sont des algorithmes déterministes. On appelle « statut » l'ensemble des valeurs des paramètres d'un algorithme de génération de nombres pseudo-aléatoires. Un statut permet entre autre la sauvegarde puis la restauration de l'état d'un générateur. Contrairement à la conception de bibliothèque telles que « SSJ » (L'Ecuyer et Buist, Simulation in Java with SSJ 2005) ou « rStream » (L'Ecuyer et Leydold, rstream: Streams of random numbers for stochastic simulation 2005b), qui visent à encapsuler et à masquer l'initialisation des générateurs pseudo-aléatoires, je pense que le concept de statut doit constituer un élément central pour une bibliothèque de génération de nombres pseudo-aléatoires et pour un outil de distribution de simulations stochastiques parallèle ou non.

III.1 Les fichiers statuts dans CLHEP

Le format d'un statut est généralement lié à la bibliothèque de génération de nombres pseudo-aléatoires avec laquelle il est généré. Le Code 4 présente un fichier statuts généré en appelant `saveStatus` sur une instance de la classe `MTwistEngine` de la bibliothèque d'outil pour la physique des hautes énergie CLHEP (A Class Library for High Energy Physics) (Lönnblad 1994). Cette classe implémente l'algorithme Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997).

```
9876
264775515 4114332253 1693459191 1297443001 1812818420 390720553
1760801786 111913646 1261209382 1738580187 1759767223 54905799 553355281
39659507 497259375 1725209625 352772144 1691117641 1506880178 1389450757
960780497 582165663 1132749798 420530616 323055843 612563404 1731170488
1036460796 1128296847 964758519 794943841 731690510 984819129 764846241
1253276905 367050022 1452395346 1452191025 1595612734 1190923226
1888268903 510135841 458398934 952243819 1812103019 221128356 39010554
48997976 1460767063 213893624 1146382177 1202087495 544350193 122224664
1320665146 1437121558 1645892194 1743285185 1085054950 1111073014
```

```
537523505 1556959762 1866178558 588707324 1241404564 1415811824
1759681885 1313644106 999414262 1845651540 329184860 1554926281
2095295052 174189579 792553342 193112426 366963614 1744436161 742314800
759227466 1921816375 369528592 413617915 513709437 1130960544 1584124393
1967512242 1619392283 362674046 1245829496 1488080741 142998155
1082210214     …     3828382448 2436847781
128
```

Code 4 Extrait d'un statut généré par la classe MTwistEngine de CLHEP

Le statut expose les paramètres de l'algorithme Mersenne Twister 19937, cependant il ne contient aucune méta-information pour donner du sens à ces données. Dans sa version 32 bits, cet algorithme travaille sur un tableau de 624 entiers codés sur 32 bits, et utilise un entier qui correspond à l'index d'un nombre dans le tableau. On peut comprendre sans regarder le code source que les nombres de la deuxième ligne qui sont au nombre de 624 sont des représentations ASCII des entiers du tableau interne à l'algorithme et que le nombre présent à la troisième ligne du fichier est l'indice courant dans le tableau. Il faut étudier le code source de CLHEP pour comprendre l'intérêt du nombre présent à la première ligne. Celui-ci correspond à un code interne à CLHEP permettant de vérifier que le fichier est bien un statut pour la classe `MTwistEngine`.

La compréhension de la structure des statuts générés par la classe de CLHEP `HepJamesRandom`, présentée dans le Code 5, est encore plus complexe. Cette classe implémente l'algorithme James Random (James 1990). Cet algorithme est initialisé en utilisant un tableau de 97 nombres doubles précisions, ainsi que trois nombres doubles précisions et un entier sur 32 bits. Après une étude approfondie du code source et un effort de rétro-ingénierie, on comprend que les nombres à virgule flottante en double précision sur 64 bits sont sauvegardés sous la forme d'une paire d'entiers non signés sur 32 bits.

```
Uvec
2226355706
1070696234
0
1072335322
1072617941
0
```

```
…
1071574158
1073741824
1072693247
2684354560
7
```

Code 5 Extrait d'un statut généré par la classe HepJamesRandom de CLHEP

Lors du passage de la version 2.0.1.2 à la version 2.0.2.0, le format des statuts pour la classe James Random de CLHEP à été modifié. Le Code 6 présente l'ancien format, dans lequel le tableau de 97 nombres à virgule flottante double précision est codé selon une représentation ASCII avec un double par ligne. Même si une option de compilation permet d'utiliser l'ancien format avec les nouvelles versions de CLHEP, le nouveau format est par expérience incompatible avec les anciennes versions de CLHEP.

```
0
0.27387481927871704
0.4186791181564331
0.18749880790710450
…
0.07554703950881958
0.6935640573501587
0.9640908241271973
0.4831882119178772 0.45623308420181274 0.9999998211860657
7
```

Code 6 Extrait d'un statut généré par la classe HepJamesRandom de CLHEP pour les versions antérieures à 2.0.2.0

Enfin, mauvaise nouvelle pour ceux qui avaient passé du temps à paralléliser les générateurs de la bibliothèque CLHEP (moi entre autre) et capitalisé sur la création de statuts aux formats CLHEP : cette bibliothèque ne sera plus développée mais simplement maintenue !

Les algorithmes de génération de nombres pseudo-aléatoires de la bibliothèque CLHEP sont implémentés dans des classes regroupées dans le paquetage « Random ». La Figure 33 présente un extrait de la hiérarchie des classes de ce paquetage. Les classes qui implémentent les algorithmes de génération de nombres pseudo-aléatoires héritent

de la superclasse `HepRandomEngine`. Les sous-classes présentées dans le diagramme correspondent aux implémentations des algorithmes :

- Ranecu (L'Ecuyer, Efficient and Portable Combined Random Number Generators 1988)
- Mersenne Twister 19937 dans sa version 32 bits (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997)
- James Random (James 1990)

Chaque algorithme de ce paquetage permet de générer une séquence cyclique de nombres pseudo-aléatoires et possède son propre format de statut. Chacune de ces classes redéfinit la méthode `saveStatus`, produisant des formats de statut très différents les uns des autres et sans information commune.

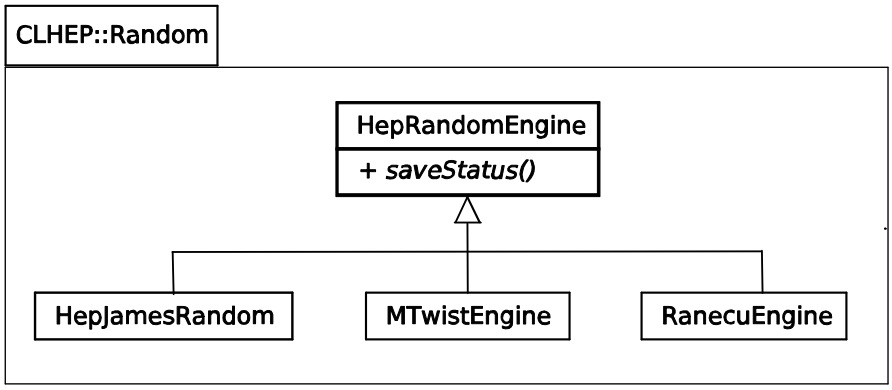


Figure 33 Diagramme de classe UML présentant un extrait de l'architecture du paquetage Random de CLHEP

Dans CLHEP, lors de la restauration de l'état d'un générateur de nombres pseudo-aléatoires, un statut contenu dans un fichier dans un format dépendant d'un algorithme est lu par un objet instance d'une classe qui implémente cet algorithme de génération de nombres pseudo-aléatoires (Figure 34). Cette classe est ensuite utilisée pour générer une séquence spécifique de nombres pseudo-aléatoires. Le format du statut est ainsi spécifique de l'algorithme mis en œuvre.

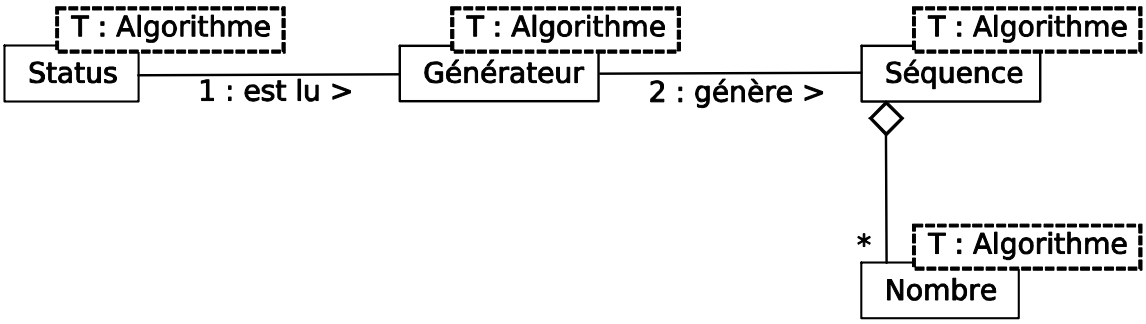


Figure 34 Diagramme UML de collaboration entre générateur et statut dans les implémentations d'algorithme de génération de nombres pseudo-aléatoires comme CLHEP

En résumé les statuts générés par CLHEP n'ont pas de méta-informations qui les rendent compréhensibles, ne contiennent pas de méta-information sur la séquence de nombres pseudo-aléatoires qu'ils permettent de générer et ont un format propre à une classe.

III.2 La conception de statuts génériques

Nous pensons que la conception d'un statut de CLHEP n'est pas du tout adaptée au calcul distribué. Je propose donc l'utilisation d'un format de statut générique contenant des méta-informations le rendant lisible et facilitant son utilisation en environnement de calcul distribué.

Le concept de statut générique est exposé sur la Figure 35. Un statut générique comprend tout le nécessaire à l'initialisation d'un l'algorithme de génération de nombres pseudo-aléatoires et des métadonnées sur la séquence de nombres qu'il permet de générer. Le format des données est spécifique à un algorithme de génération cependant les métadonnées sont génériques et donc leur format est indépendant de l'algorithme de génération concerné. Il est ainsi possible de vérifier l'utilisation correcte du statut dans un contexte de calcul distribué, sans même avoir besoin d'une implémentation ou même de connaître le générateur associé. Le format générique doit de plus comprendre dans sa partie « donnée » d'initialisation pour le générateur de nombres pseudo-aléatoires des méta-données renseignant sur la disposition des paramètres de l'algorithme.

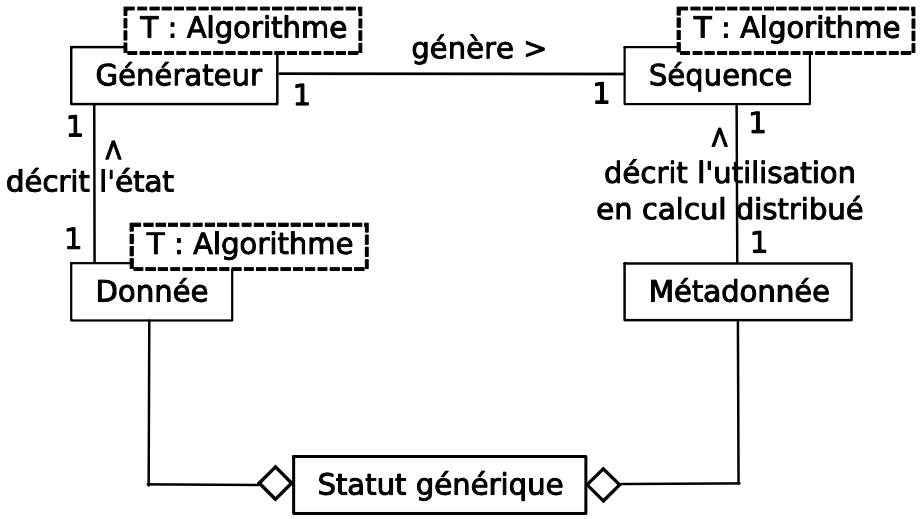


Figure 35 Diagramme UML de collaboration de générateurs de nombres pseudo-aléatoires et des statuts génériques

La Figure 36 représente le diagramme d'instanciation des statuts génériques pour un générateur de type « James Random ». L'instance du statut générique agrège une partie métadonnée et une partie donnée. La partie « donnée » est instance de la classe `Donnée`, elle même instance de la métaclasse `Donnée` paramétrée pour l'utilisation de l'algorithme de type « James Random » (JR). La partie « métadonnée » est quand à elle systématiquement instance d'une classe non paramétrique `Métadonnée`.

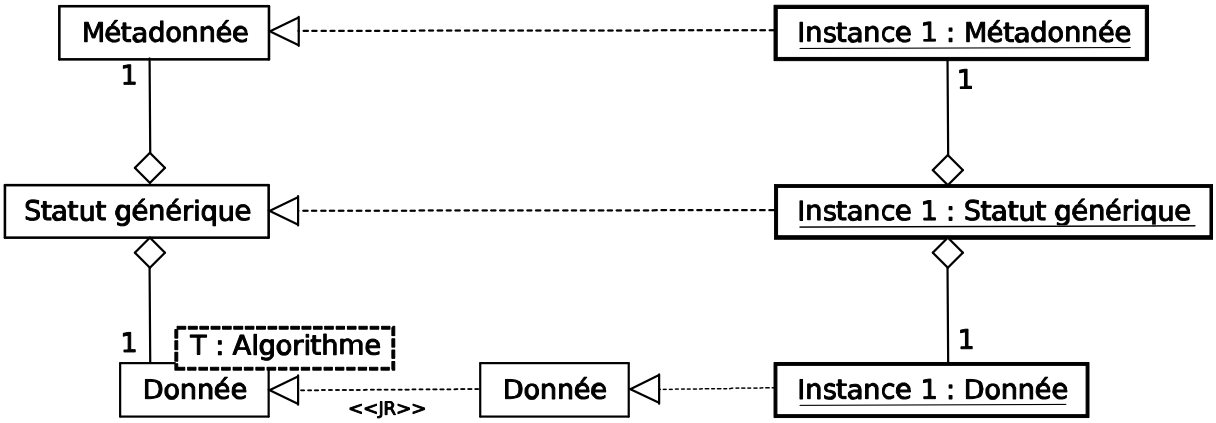


Figure 36 Diagramme UML d'instanciation pour un statut générique

La conception des statuts génériques est suffisamment flexible pour être utilisée aussi bien avec des séquences de nombres pseudo-aléatoires qu'avec des séquences de nombres quasi-aléatoires en calcul distribué. La partie « métadonnée » reste la même et la distribution des séquences utilise les mêmes mécanismes pour les deux types de séquences. Des formats de statuts génériques ont ainsi étés conçus pour les algorithmes de Sobol (Sobol 1976) et de Van der Corput.

III.3 Statuts XML

Afin de donner corps au concept de statut générique, j'ai conçu un format ouvert en XML pour la sauvegarde des statuts des algorithmes de génération de nombres pseudo-aléatoires et quasi-aléatoires. Le langage XML permet d'une part, de générer des statuts lisibles grâce aux méta-informations contenues dans balises XML (elles donnent un sens aux données qu'elles contiennent), et d'autre part, de placer dans le fichier des méta-informations sur l'utilisation dans un contexte de calcul distribué de la série de nombres pseudo-aléatoires générée à partir du statut. Le Code 7 expose le format de ces statuts dans une version en attente de commentaires. Ce format est publié dans (R. Reuillon, D. Hill, et al., Rigorous distribution of stochastic simulations using the DistMe toolkit 2008a).

Pour tous les statuts XML, la balise XML racine est `Status`. Ensuite se trouvent trois méta-informations sur la séquence de nombres pseudo-aléatoires générée. Tout d'abord le tag XML `distance` contient un nombre entier qui indique le nombre d'itérations de l'algorithme depuis l'initialisation avec un statut appelé statut d'origine, dont la distance est fixée arbitrairement à 0. Cette indication est utile pour éviter les problèmes de chevauchement lors de parallélisation des algorithmes par un découpage en séquence. Dans le format XML proposé, les attributs `subSerie` et `group` sont des chaines de caractères correspondant respectivement à la désignation de la sous-série et à celle du groupe dont les significations sont expliquées au paragraphe suivant.

```
<Status>
  <distance></distance>
  <subSerie></subSerie>
  <group></group>
  <data class="Nom de l'algorithme">
  </data>
</Status>
```

Code 7 Format d'un statut générique en XML

La Figure 37 présente un diagramme de classes illustrant les concepts de groupes et de sous-séries pour un générateur. Tout d'abord ce modèle s'applique à tous les algorithmes de génération de nombres pseudo-aléatoires. Les statuts sont regroupés

dans des sous-séries, elles mêmes regroupées dans des groupes. La notion de groupe correspond à un groupe de sous-séries indépendantes entre elles. Toutes les sous-séries appartenant à un même groupe ont été générées de sorte à être indépendantes entre elles. Une sous-série peut contenir plusieurs statuts. Ces statuts sont issus d'une parallélisation par un découpage en séquence de la sous-série. Ces statuts sont distingués entre eux par un attribut distance, qui permet de connaître le nombre d'itérations de l'algorithme de génération qui séparent chaque statut des autres. Le statut dont la distance est 0 est appelé le statut origine. Chaque statut est lié à des données spécifiques pour l'initialisation du générateur de nombres pseudo-aléatoires correspondant.

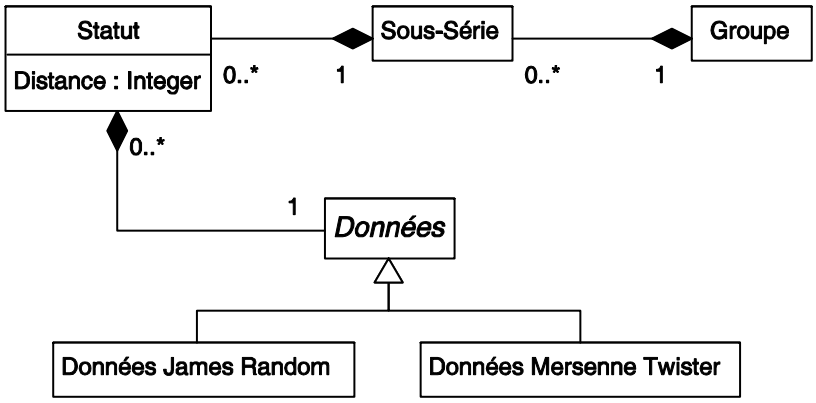


Figure 37 Diagramme UML de classes représentant les métadonnées et les données d'un statut

Dans le cas de la parallélisation d'un générateur de nombres pseudo-aléatoires par paramétrisation: l'algorithme de paramétrisation est mis en œuvre pour générer des statuts pour le générateur de nombres pseudo-aléatoires correspondants à des sous-séries indépendantes entre elles. Ces statuts appartiennent chacun à des sous-séries distinctes. Ces sous-séries sont regroupées dans un même groupe de sous-séries indépendantes. Chaque sous-série contient ainsi un unique statut permettant d'initialiser le générateur de nombres pseudo-aléatoires. Ce statut contient un attribut `distance` fixé arbitrairement à 0. Il est considéré comme le statut d'origine pour chacune des sous-séries. Une sous-série peut ensuite être parallélisée par découpage en séquence. Le générateur est déroulé à partir du statut origine d'une sous-série. Les statuts générés de la sorte appartiennent alors à la même sous-série mais ont un attribut distance différent comptabilisant le nombre d'itération de l'algorithme depuis le statut origine. Cet exemple est illustré par la Figure 38.

Cette notion de sous-série indépendante est utilisable avec toutes les techniques de parallélisation décrite dans le chapitre 1 de ce manuscrit. Par exemple, dans le cas d'une parallélisation par indexation de séquences. Les statuts sont générés aléatoirement, il est ainsi impossible de connaître le nombre d'itérations séparant chaque statut. On considère ces statuts comme menant à des sous-séries indépendantes comme dans le cas de la paramétrisation.

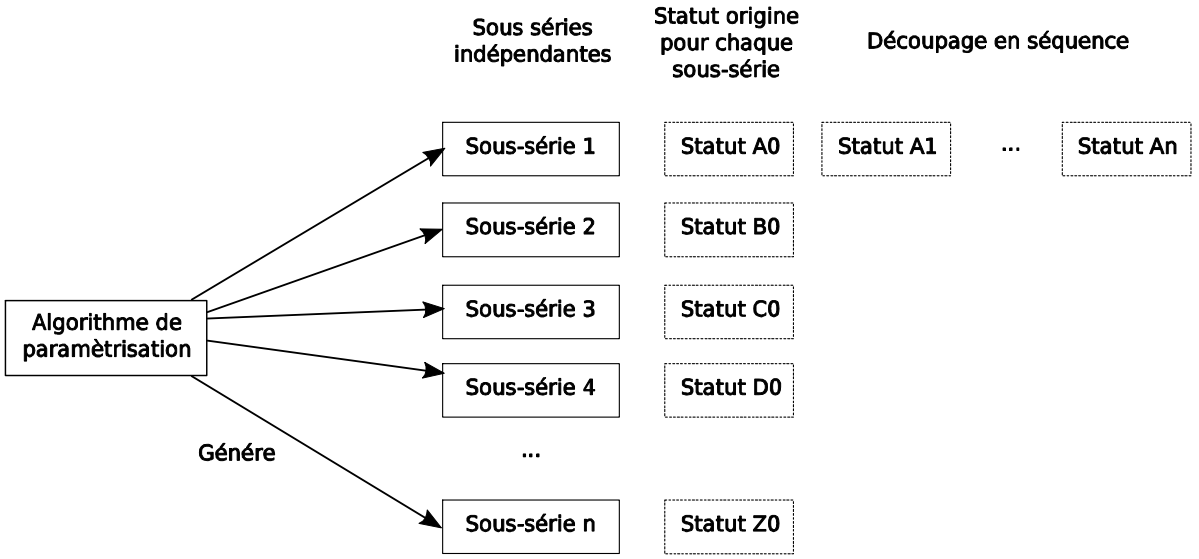


Figure 38 Utilisation des notions de groupes et de sous-séries pour une parallélisation d'un algorithme de génération de nombres pseudo-aléatoires par paramétrisation

Pour la parallélisation par saut de grenouille, un statut permet d'initialiser le générateur de façon à ce qu'il génère la sous-série voulue (un nombre tout les *n* de la série initiale du générateur de nombre pseudo-aléatoires parallélisé). Chaque statut constitue ainsi le statut origine d'une sous-série parmi les *n* sous-séries indépendantes.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Status" type="s"/>
  <xsd:complexType name="s">
    <xsd:sequence>
      <xsd:element name="distance" type="xsd:integer" />
      <xsd:element name="subSerie" type="xsd:string" />
      <xsd:element name="group" type="xsd:string" />
      <xsd:element name="data" type="dataJR" />
    </xsd:sequence>
  </xsd:complexType>
    …
</xsd:schema>
```

Code 8 Extrait du schéma XML vérifiant la conformité d'un statut XML pour l'algorithme James Random (Partie commune)

Les statuts XML sont des modèles en XML de l'état d'un algorithme de génération de nombres pseudo-aléatoires. Le Code 8 présente un extrait du méta-modèle auquel doit être conforme tout statut XML générique pour l'algorithme James Random. Le tag XML `data` contient un attribut `class` indiquant le nom de l'algorithme de génération de nombres pseudo-aléatoires. A l'intérieur de la balise `data` se trouvent les données pour initialiser l'algorithme de génération de nombres pseudo-aléatoires.

```
<xsd:complexType name="data">
  <xsd:attribute name="class" type="algo" />
</xsd:complexType>
<xsd:simpleType name="algo">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="JR" />
  </xsd:restriction>
</xsd:simpleType>
<xsd:complexType name="dataJR">
  <xsd:complexContent>
    <xsd:extension base="data">
      <xsd:sequence>
        <xsd:element name="u">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="value"
                           type="tabElement"
                           minOccurs="97"
                           maxOccurs="97" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="c" type="xsd:double" />
        <xsd:element name="cd" type="xsd:double" />
        <xsd:element name="cm" type="xsd:double" />
        <xsd:element name="j97" type="xsd:int" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
```

Code 9 Extrait du schéma XML vérifiant la conformité d'un statut XML pour l'algorithme James Random (Partie propre à James Random)

Les trois éléments : `distance`, `subSerie` et `data` sont communs à l'ensemble des méta-modèles pour tous les statuts en XML, quelque soit l'algorithme de génération de nombres pseudo-aléatoires correspondant. Le tag `distance` est renseigné sous forme d'un entier, alors que les tags `subSerie` et `group` contiennent des chaînes de caractères. La partie variable, selon les méta-modèles, est la structure du tag `data`.

Le Code 9 présente un extrait du schéma XML définissant le méta-modèle pour la partie donnée des statuts génériques en XML pour le générateur James Random. Pour qu'un statut XML soit conforme à ce schéma, l'attribut `class` du tag `data` doit contenir la valeur « JR ». Et sa structure doit contenir une séquence `u` de 97 valeurs entières ainsi que 3 nombres doubles précisions : `c`, `cd` et `cm` ainsi qu'un entier `j97`. Cette structure correspond aux paramètres de l'algorithme décrit dans (James 1990).

```
<Status>
  <distance>15000000000000</distance>
  <subSerie>1</subSerie>
  <group>Demo</group>
  <data class="JR">
    <u>
      <value index="0">0.27387481927871704</value>
      <value index="1">0.4186791181564331</value>
       …
      <value index="96">0.9640908241271973</value>
    </u>
    <c>0.4831882119178772</c>
    <cd>0.45623308420181274</cd>
    <cm>0.9999998211860657</cm>
    <j97>7</j97>
  </data>
</Status>
```

Code 10 Exemple de statut générique en XML pour l'algorithme James Random

Un statut conforme à la description XML ci-dessus est présenté en Code 10. Sa distance par rapport au statut origine est de 15 000 milliards de tirages. Ce statut fait partie de la sous-série « 1 » qui appartient aux groupes « Demo » généré pour cet algorithme. On retrouve ensuite le tag XML `data`, qui comporte l'attribut `class`, dont la valeur est « JR » pour James Random . A l'intérieur de ce tag, sont présentes les données pour initialiser l'algorithme « James Random ». On y trouve un vecteur de 97 nombres doubles précisions, trois nombres doubles précisions et un nombre entier.

Bien que l'utilisation de statut constitue une bonne solution pour l'initialisation de générateurs de nombres pseudo-aléatoires en environnement distribué, l'utilisation de statuts est rudimentaire ou absente dans les bibliothèques de génération parallèle de nombres pseudo-aléatoires actuelle. Dans SPRNG (Mascagni et Srinivasan, Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation 2000), la gestion des statuts est similaire à celle de CLHEP. La sérialisation se fait à l'aide de la méthode virtuelle `pack_rng` de la classe `sprng`. Celle-ci est redéfinie pour chaque générateur. La sauvegarde l'état du générateur se fait dans un format non documenté.

Dans la version actuelle (2.0) de SSJ (L'Ecuyer et Buist, Simulation in Java with SSJ 2005) la gestion des statuts est inexistante. Il n'y a pas de possibilité d'initialiser un générateur à partir d'un fichier ou d'une chaîne de caractère.

Dans la suite logicielle Dist, la classe `statut`, les classes `data` pour différents algorithmes de génération de nombres pseudo-aléatoires et quasi-aléatoires, ainsi que les mécanismes de sélection et de sérialisation des statuts sont implémentés dans la bibliothèque fondatrice DistTools. Celle-ci est présentée dans la partie suivante.

IV DistTools

DistTools est un progiciel qui implémente les fondations de la suite logicielle Dist. Cette partie présente les parties essentielles de DistTools. Elle décrit tout d'abord les classes représentant les statuts, qui contiennent les données pour l'initialisation et l'exécution des générateurs quasi-aléatoires et des générateurs pseudo-aléatoires parallèles. Elle expose ensuite les classes pour la gestion et l'interface avec une base de données de statuts. Elle présente enfin les classes de sérialisation et de sérialisation des objets statuts vers leurs représentations sous forme de chaînes de caractères.

IV.1 Les statuts dans DistTools

La gestion des statuts est le cœur du projet Dist. Le concept de statut générique trouve son implémentation au sein du progiciel DistTools. DistTools implémente ainsi une classe paramétrique `CStatus` qui agrège des métadonnées sur une séquence de nombres et les données d'initialisation du générateur de nombres pseudo-aléatoires ou quasi-aléatoires. La classe `CStatus` est paramétrée par une classe correspondant au type de données qu'elle peut contenir.

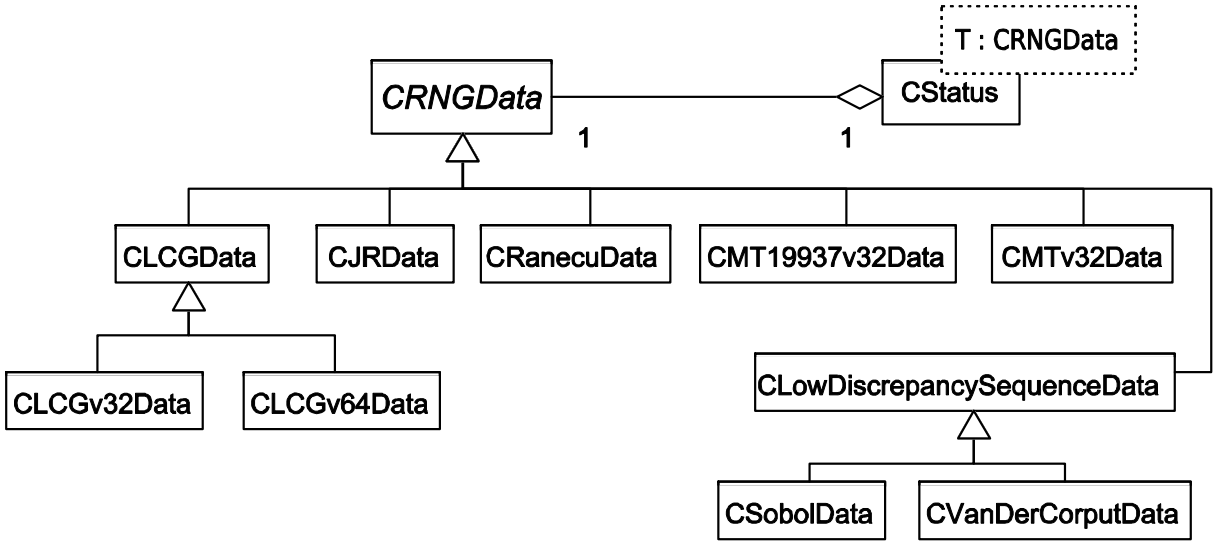


Figure 39 Diagramme UML de classe du package « status » de DistTools

La Figure 39 présente le paquetage `status` de DistTools. On y retrouve la classe paramétrique `CStatus`. Chaque instance de cette classe est composée d'une instance d'une des classes héritant de `CRNGData`. Chaque sous-classe `CRNGData` correspond au support d'un algorithme de génération de nombres quasi-aléatoires ou d'un générateur de nombres pseudo-aléatoires par la suite logicielle Dist. La correspondance entre les algorithmes et les classes de données de DistTools est présentée dans le Tableau 7.

Tableau 7 Correspondance entre les algorithmes de génération de nombres pseudo-aléatoires ou quasi-aléatoires et les classes du paquetage « status » de DistTools

Désignation du générateur	Nom de la classe correspondante
Ranecu (L'Ecuyer, Efficient and Portable Combined Random Number Generators 1988)	CRanecuData
Générateur linéaire congruentiels utilisant des	CLCGv32Data

nombres sur 32 bits	
Générateur linéaire congruentiels utilisant des nombres sur 64 bits	CLCGv64Data
Mersenne Twister 19937 dans version 32 bits (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997)	CMT19937v32Data
Mersenne Twister générique version 32 bits (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000)	CMTv32Data
James Random (James 1990)	CJRData
Sobol (Sobol 1976)	CSobolData
Van Der Corput	CVanDerCorputData

IV.2 Flux de nombres pseudo-aléatoires parallèles

La gestion des flux de nombre pseudo-aléatoires parallèles est implémentée dans DistTools sous forme d'un dépôt de statuts et d'un mécanisme de sélection de statuts dans ce dépôt. Dans la version actuelle, le dépôt est un système de base de données relationnelles en local. Il permet la gestion d'un grand nombre de statuts (le nombre maximal de statuts dépend de l'espace disque disponible sur la machine.)

La Figure 40 présente les classes de gestion et de sélection des statuts. `IStatusProvider` est une interface permettant d'utiliser différents types de dépôts de statuts. Cette interface est sans état et expose des méthodes dont les prototypes sont basés sur des types simples. Elle peut donc potentiellement être implémentée simplement par des types de dépôts de statuts variés. A l'heure actuelle, seule la classe `CLocalStatusProvider` présente une implémentation de l'interface `IStatusProvider`.

Cette classe est conçue selon de patron de conception singleton et implémente des méthodes de stockage et de sélection de statuts dans une base de données locale de type H2[43]. L'utilisation d'une fonction de hachage SHA-1 (NIST 1995) sur la version sérialisée

[43] http://www.h2database.com

en XML des statuts (voir partie IV.3) permet de garantir l'unicité des statuts stockés dans la base et d'éviter une insertion multiple d'un même statut.

La classe `CStatusHub` permet la sélection d'un ensemble de statuts dans un dépôt de statuts implémentant l'interface `IStatusProvider`. Cette sélection est effectuée selon une stratégie s'appuyant sur l'utilisation des métadonnées des statuts génériques. La classe `CLargeSequenceSplitting` sélectionne des statuts appartenant à un même groupe de sous-séries indépendantes entre elles et avec une distance minimale de n tirages (n est passé en paramètre). La seconde stratégie, implémentée dans la classe `CStrictSubSerie`, sélectionne un statut pour chaque sous-série parmi les statuts d'un groupe de sous-séries indépendantes. Le statut choisi est celui qui présente la distance minimale.

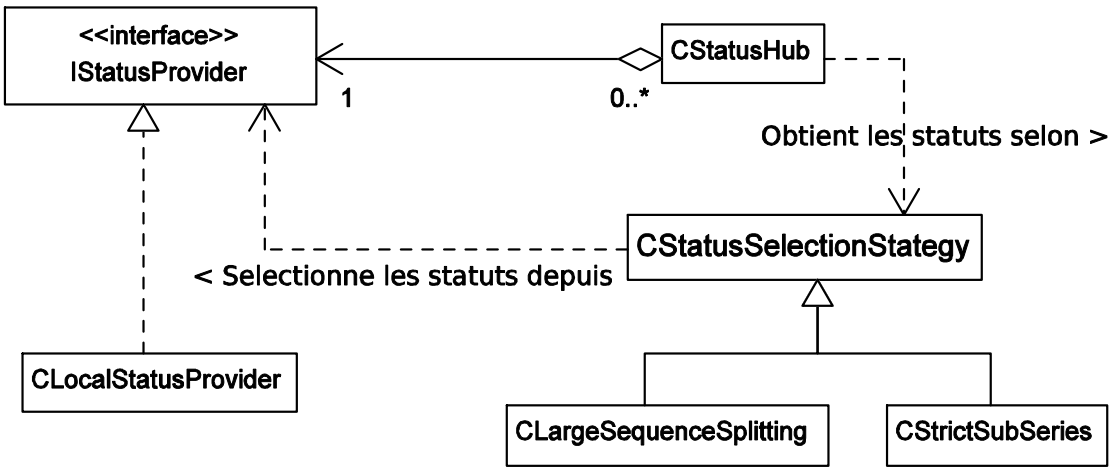


Figure 40 Diagramme UML de classe du package « status provider » de DistTools

A court terme une implémentation de `IStatusProvider` fournira la possibilité de se connecter à un service web mettant à disposition des utilisateurs des dizaines de milliers de statuts pour différents algorithmes de génération, centralisant et factorisant ainsi la tâche de génération des statuts et de test des séquences de nombres pseudo-aléatoires parallèles.

IV.3 La sérialisation des statuts

Même si le concept de statut générique est implémenté dans le paquetage `status`, le paquetage `serializer` permet de lier le concept des statuts génériques à des statuts physiques sous forme de fichiers, ou de chaînes de caractères utilisables par diverses

librairies de génération de nombres pseudo-aléatoires. Ainsi DistTools fournit la possibilité de sérialiser et de désérialiser les objets statuts.

Le paquetage `serializer`, est présenté dans la Figure 41. La classe `CStatusSerializer` est une classe abstraite, superclasse de toutes les classes de sérialisation de statuts de la suite logicielle Dist. La classe `CSerializerXML` s'appuie sur la bibliothèque open source XStream[44] pour permettre la sérialisation d'objets java en XML. La classe `CStatusSerializerXML` décore la classe `CSerializerXML` selon le patron de conception « décorateur » (Gamma, et al. 1995) et fournit une sérialisation générique, transversale, et non intrusive des objets statuts en chaîne de caractère au format XML.

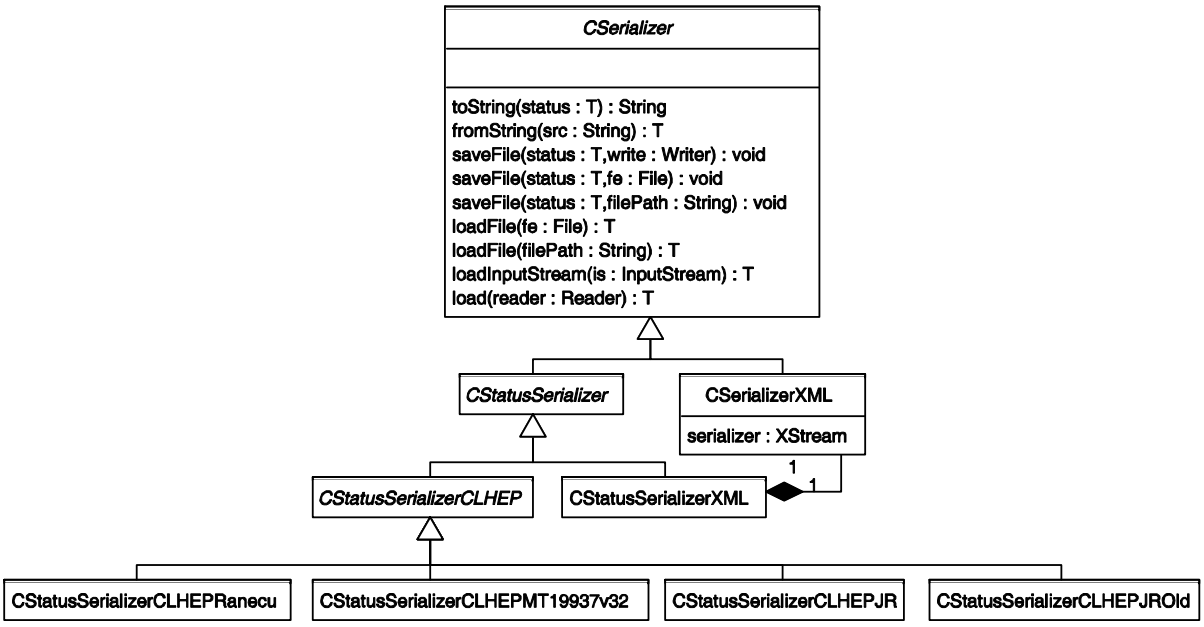


Figure 41 Diagramme UML de classe du package « serializer » de DistTools

DistTools autorise la sérialisation et la désérialisation non-intrusive et transversale des statuts aux formats CLHEP. La classe `CStatusSerializerCLHEP` est une superclasse pour les classes de sérialisation de, et vers, les statuts CLHEP. Ces derniers ne sont pas génériques ; DistTools implémente donc une classe de sérialisation par type de fichier statuts à destination de CLHEP. Ces classes ont été conçues grâce à un travail de retro-ingénierie sur les classes de génération de nombres pseudo-aléatoires de CLHEP. On remarquera que deux classes différentes sont implémentées pour la sérialisation des statuts pour l'algorithme James Random dans la bibliothèque CLHEP :

[44] http://xstream.codehaus.org/

`CStatusSerializerCLHEPJR` et `CStatusSerializerCLHEPJROld`. Les deux formats de statuts sont ainsi pris en compte (voir partie III.1 de ce chapitre).

Si les statuts génériques de DistTools peuvent être sérialisés et utilisés par des bibliothèques tierces, la suite logicielle Dist propose une bibliothèque de génération de nombres pseudo-aléatoires et quasi-aléatoires, tirant le meilleur parti du concept de statut générique pour des simulations stochastiques en environnement distribué. La partie suivante présente cette bibliothèque.

V DistRNG

DistRNG est une bibliothèque de génération de nombres quasi-aléatoires et pseudo-aléatoires. Cette bibliothèque est basée sur l'utilisation intensive de statuts génériques au format XML implémentés dans DistTools. Cette partie présente les paquetages de classe, principaux composant de DistRNG : le paquetage regroupant les algorithmes de génération de nombres pseudo-aléatoires, le paquetage de formatage des nombres de sortie des algorithmes et le paquetage d'instanciation des générateurs.

V.1 La génération de nombres quasi et pseudo-aléatoires

Le premier paquetage présenté dans cette partie est le paquetage contenant les algorithmes de génération de nombres quasi-aléatoires et les générateurs de nombres pseudo-aléatoires. Comme décrit sur la Figure 42, ce paquetage comporte une unique superclasse paramétrique `CRNG`. Chaque sous-classe implémente un algorithme de génération de nombres, soit pseudo-aléatoires, soit quasi-aléatoires. Chaque classe de ce paquetage hérite de la méta-classe `CRNG` paramétrée par un type de statuts. Par exemple, la classe `CJR` hérite de la classe `CRNG` paramétrée par la classe `CStatus<CJRData>`. Une instance d'une classe héritant de `CRNG` est ainsi systématiquement composée d'une instance de la classe `CStatus`, elle-même composée d'une instance d'une classe héritant de `CRNGData` correspondant à l'algorithme implémenté. Cette solution permet de séparer les algorithmes de génération de leur partie métadonnées et données sans perdre la généricité dans l'utilisation des générateurs.

Dans la version actuelle de DistRNG on trouve une implémentation des algorithmes :

- Linéaire congruentiels utilisant des nombres sur 32 ou 64 bits,

- Mersenne Twister 19937 dans sa version 32 bits (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997),
- Mersenne Twister générique dans sa version 32 bits (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000),
- James Random (James 1990),
- Sobol (Sobol 1976),
- Van Der Corput.

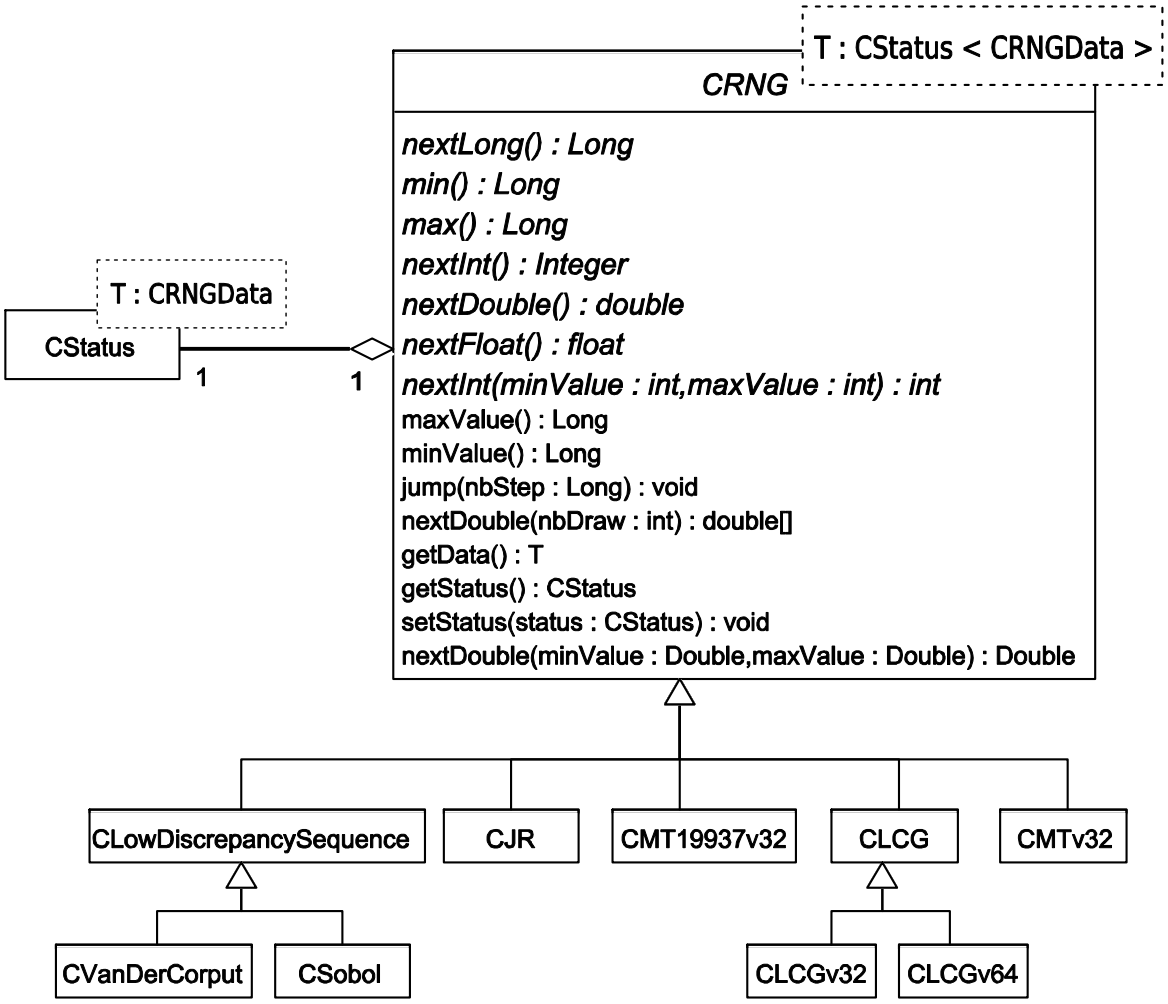


Figure 42 Diagramme UML de classe du paquetage RNG de DistRNG

DistRNG est conçue pour être hautement extensible. L'ajout d'un générateur supplémentaire nécessite trois opérations :

1. l'ajout d'une classe qui hérite de `CRNGData` dans DistTool pour représenter le statut du générateur,

2. l'encapsulation de l'algorithme au sein d'une classe qui hérite de `CRNG` dans DistRNG,
3. l'inscription du type du statut et du générateur au sein des deux paquetages conçus pour la construction des objets statuts et des générateurs à partir de leur représentation XML.

L'implémentation d'un algorithme de génération de nombres pseudo-aléatoires ou quasi-aléatoires est en outre facilitée avec DistRNG, par l'utilisation du paquetage spécialisée dans le changement de formats des nombres, présenté dans la partie suivante.

V.2 Le formatage des nombres

Le paquetage de formatage du projet DistRNG permet de déléguer le changement de format des nombres générés à des classes spécialisées. Par exemple, la classe de formatage `CRNGFormaterFrom32BitsInt` prend en charge le transtypage rigoureux des nombres générés par un générateur conçu pour générer nativement des entiers sur 32 bits vers : des nombres entiers sur 64 bits, des nombres à virgule flottante simple précision et de nombres à virgule flottante double précision. Nous avons vu précédemment que cette tâche est difficile et peut être source d'erreur ou de comportement non documenté (cf. classe `MTwistEngine` de la bibliothèque CLHEP.) Contrairement aux autres bibliothèques de génération de nombres pseudo-aléatoires, au sein de DistRNG les opérations de formatage des nombres sont implémentés en dehors des classes de génération. Ceci permet de factoriser cette tâche difficile et de réutiliser ces parties de code entre les différents algorithmes de génération.

Le Code 11 est un extrait de la classe `CRNGFormaterFrom32BitsInt`. Ce code génère un nombre double précision sur 64 bits à partir de deux nombres aléatoires de type entier sur 32 bits. Il fait appel par deux fois à la méthode `nextInt` du générateur qui lui est associé afin de générer deux nombres entiers et il les combine pour générer un nombre à virgule flottante double précision entre 0 et 1. Ce code n'est pas trivial et ne doit, par conséquent, pas être laissé à la charge de chaque classe de génération de nombres pseudo-aléatoires, afin d'éviter les erreurs d'implémentation comme dans la bibliothèque CLHEP.

```
public double nextDouble() {
  int x = rng.nextInt();
  int y = rng.nextInt();
  return ((((long) (x >>> 6)) << 27) + (y >>> 5)) /
                                        (double)(0x001FFFFFFFFFFFL);
}
```

Code 11 Méthode nextDouble de la classe CRNGFormaterFrom32BitsInt

V.3 La parallélisation des générateurs

DistRNG est conçue pour générer des flux de nombres pseudo-aléatoires parallèles. Plusieurs techniques de parallélisation des générateurs de nombres aléatoires, correspondant à l'état de l'art dans le domaine, sont implémentées à l'heure actuelle dans DistRNG.

La Figure 43 présente les classes conçues pour la parallélisation des générateurs dans DistRNG. La classe `CSequenceSplitting` permet de générer des générateurs de nombres pseudo-aléatoires indépendants selon la technique de découpage en séquence. Elle profite des techniques de division de cycle de chaque générateur en appelant la méthode virtuelle `jump` de la classe CRNG. Cette méthode peut être redéfinie par chaque générateur afin de mettre en œuvre une technique de division de cycle sur ce générateur.

La classe `CIndexSequence` permet de paralléliser un générateur selon la technique d'indexation de séquences (Coddington et Newell, JAPARA – A Java Parallel Random Number Library for High-Performance Computing 2004). Pour cette technique, l'état du générateur est généré aléatoirement en utilisant un autre générateur de nombres pseudo-aléatoires. Les méta-informations (annotations java) correctement renseignées sur les attributs des classes d'initialisation des générateurs pseudo-aléatoires (c'est-à-dire les attributs des classes héritant de `CRNGData`) permettent de fixer aléatoirement une valeur aux attributs de la partie données des statuts de manière transversale. De ce fait, dans DistRNG l'indexation de séquences peut être effectuée en utilisant n'importe quel générateur de la bibliothèque ainsi que des générateurs externes comme les générateurs de qualité cryptographique générant des séquences de bits indépendants de la bibliothèque java GNUCrypto[45]. Les statuts générés par indexation sont considérés

[45] http://www.gnu.org/software/gnu-crypto

comme des statuts menant à des sous-séries indépendantes et donc appartenant à un même groupe.

Le Code 12 présente la partie « donnée » des statuts pour le générateur Mersenne Twister 19937 version 32 bits implémenté dans DistTools. La méta-information (l'attribut java) `@ARandomizeInteger` sur l'attribut de classe `state` permet à la classe effectuant la parallélisation par indexation de séquence de DistRNG de savoir qu'il faut renseigner aléatoirement de tableau de nombres avec des entiers sur 32 bits, sur toute la plage de définition des entiers. La méta-information `@ARandomizeInteger (min=624,max=624)` signifie qu'il faut fixer l'attribut `indice` à 624.

```
public final class CMT19937v32Data extends CRNGData {

      public CMT19937v32Data() {
            super( );
            indice = 624;
      }

      @ARandomizeInteger ()
      public Integer[] state = new Integer[624];

      @ARandomizeInteger(min=624,max=624)
      public int indice;

}
```

Code 12 Classe de DistTools implémentant la partie donnée des statuts pour le générateur Mersenne Twister 19937 dans sa version 32 bits

Enfin, DistRNG est la seule bibliothèque en Java implémentant la paramètrisation pour l'algorithme Mersenne Twister générique. Cet algorithme écrit par Makoto Matsumoto permet la génération de multiples générateurs de Mersenne Twister indépendants (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000) (jusqu'à 65536 pour un même groupe). Une version écrite en C est disponible sur internet[46]. Nous avons implémenté ce code en java dans la classe `CMTv32Parametrizer`.

[46] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html

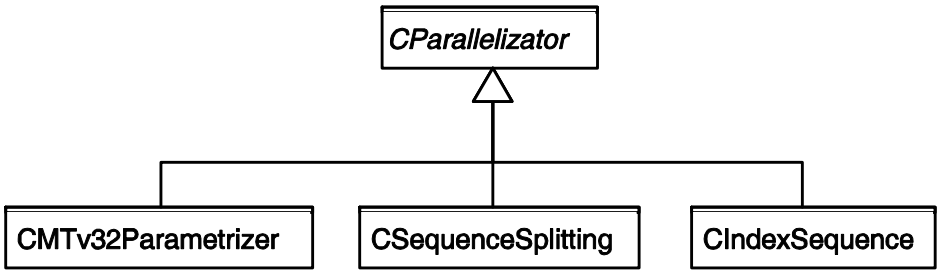


Figure 43 Diagramme UML de classe du paquetage « parallelization » de DistRNG

V.4 L'instanciation des générateurs

Nous avons vu précédemment que la méthode la plus fiable - afin de tester si un algorithme de génération de nombres pseudo-aléatoires biaise les résultats d'une simulation donnée - est de comparer les résultats de cette simulation en utilisant des générateurs de nature différente (L'Ecuyer, Random Number Generators and Empirical Tests 1998). DistRNG tire le meilleur parti des statuts génériques en XML afin de faciliter cette tâche.

Dans le paquetage `serializer` de DistRNG, la classe `CRNGSerializer` est spécialisée dans la sérialisation-désérialisation des générateurs. Elle fournit des méthodes pour la sérialisation et la désérialisation des classes qui héritent de `CRNG` depuis et vers des statuts XML. Ces opérations sont implémentées de manière transversale et peuvent ainsi être utilisées avec tous les algorithmes de génération de nombres pseudo-aléatoires ou quasi-aléatoires de DistRNG sans ajout de code spécifique au sein des classes de génération.

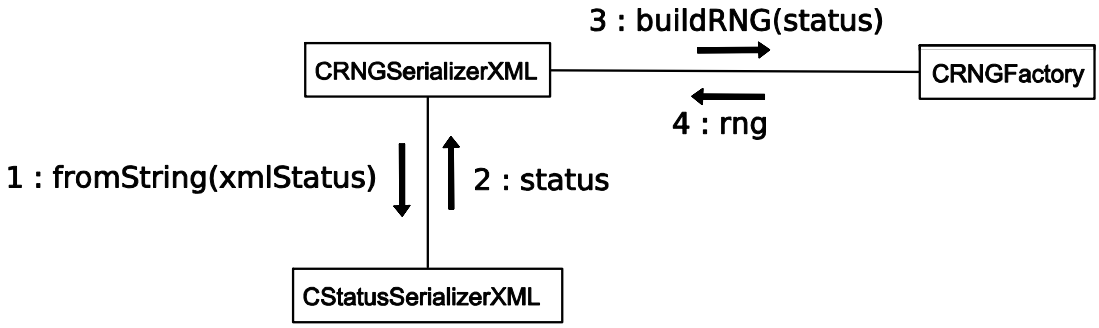


Figure 44 Diagramme UML de collaboration du paquetage « serializer » de DistRNG

La Figure 44 montre la collaboration entre les instances des classes du paquetage `serializer` de DistRNG pour la construction d'un générateur de nombres pseudo-aléatoires à partir d'un statut générique au format XML. La classe `CRNGSerializer` utilise

la classe `CStatusSerialiser` du progiciel DistTool pour la sérialisation-désérialisation de la partie statut des générateurs. Lors de l'opération de déserialisation, la classe `CRNGSerializer` fournit une chaîne de caractères contenant un statut au format XML à une instance de la classe `CStatusSerializer`. Cette dernière lui retourne un objet statut. La classe `CRNGFactory` est implémentée selon le patron de conception singleton ; l'application comporte donc une seule instance de cette classe. Cette instance permet d'associer un type donné à une classe fille de CRNG. La classe `CRNGSerializer` s'en sert donc pour construire un objet générateur de nombres pseudo-aléatoires à partir d'un objet statut.

Le Code 13 illustre l'instanciation d'un générateur avec DistRNG. Il faut noter que ce code est générique. Il ne dépend pas d'une classe de génération de nombres pseudo-aléatoires. Il n'est par conséquent pas nécessaire de fixer le type du générateur lors de l'implémentation d'un simulateur reposant sur DistRNG pour la partie génération de nombres pseudo-aléatoires. Ce choix est reporté à l'exécution. L'utilisateur pourra ainsi opter pour un générateur ou un autre sans modifier le code et sans recompilation, simplement en fournissant un statut approprié. Cette faculté est exceptionnelle pour étudier l'impact d'un algorithme de génération ou d'une méthode de parallélisation sur les résultats d'une simulation stochastique.

```
CRNGSerializerXML serialiser = new CRNGSerializerXML();
try {
  rng = serialiser.loadFile("statut.xml");
} catch (IOException e) {
  e.printStackTrace();
}
```

Code 13 Instanciation d'un générateur dans DistRNG

V.5 RNG parallèle

Une nouvelle partie de DistRNG est en cours d'implémentation afin de fournir une interface de génération de nombres pseudo-aléatoires en parallèle se rapprochant de SSJ et de RStream. L'objectif est d'utiliser les services de DistTools pour sélectionner des statuts indépendants dans la base de données de statuts et de les considérer comme des

flux et des sous-flux indépendants à la manière de SSJ et comme présenté dans (L'Ecuyer, Simard et Chen, et al. 2002).

DistRNG est une bibliothèque permettant d'implémenter des simulateurs stochastiques prêts pour une exécution en environnement distribué. L'autre logiciel de la suite Dist, DistMe, permet quand à lui la parallélisation de simulateurs basés ou non sur DistRNG. Le logiciel DistMe fait l'objet de la partie suivante.

VI DistMe

DistMe est né de la difficulté de distribuer des simulations stochastiques en un grand nombre de jobs sans commettre d'erreur et en restant rigoureux. Ce progiciel permet l'automatisation de cette tâche. Il est principalement conçu pour permettre à un développeur de distribuer une simulation stochastique de manière automatisée et rigoureuse, en accord avec l'état de l'art.

La Figure 45 représente la suite des tâches effectuées lors de la distribution d'une simulation avec le progiciel DistMe. Les tâches à la charge de l'utilisateur sont : la description des fichiers d'entrée de la simulation, la description des fichiers de sortie et la description de la commande de lancement d'un job de simulation. Toutes ces opérations constituent la description d'une simulation. Une fois ces étapes préliminaires effectuées, DistMe prend à sa charge la distribution de la simulation en jobs et la préparation de jobs prêts pour une exécution sur un environnement distribué.

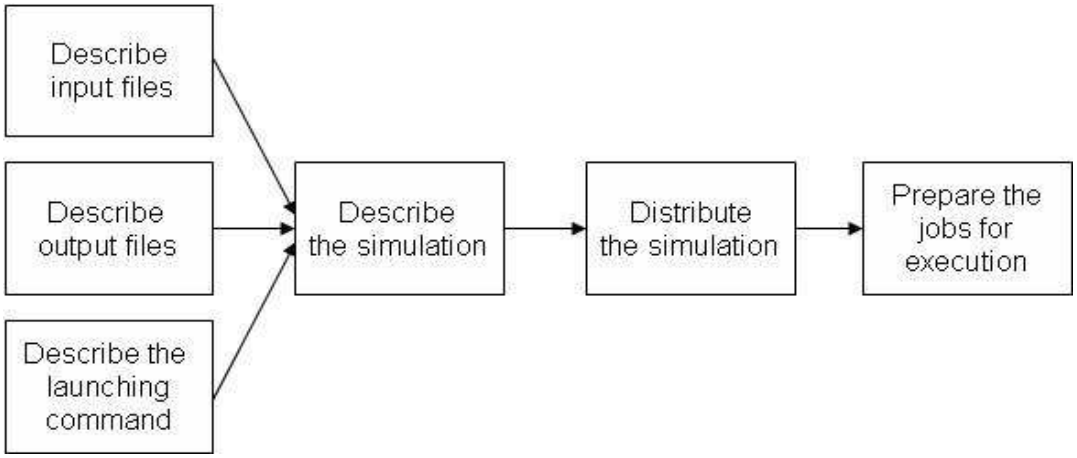


Figure 45 Suite de tâches pour la distribution d'une simulation stochastique avec DistMe

Cette partie expose dans un premier temps la tâche de description d'une simulation dans DistMe. Elle présente ensuite le paquetage `Processor` qui effectue la distribution automatique de la simulation. Enfin elle décrit le paquetage `Parameter` de DistMe qui permet la génération de fichiers uniques pour les différents jobs d'une même simulation distribuée ainsi que la distribution de plans d'expériences par exploration de paramètres.

VI.1 La description de simulations avec DistMe

Le paquetage Simulation de DistMe fournit à l'utilisateur les outils pour décrire la simulation qu'il souhaite distribuer. Le modèle d'un processus de simulation informatique est présenté dans la Figure 46. Ce modèle a servi de base à la conception de DistMe. Une simulation est une boite noire dont l'exécution est lancée par une commande de lancement. Elle utilise alors des fichiers d'entrée et génère des fichiers de sortie.

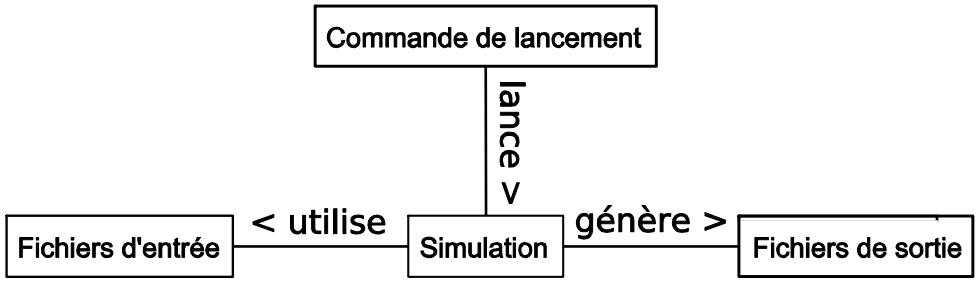


Figure 46 Modèle d'une simulation dans DistMe

Ce modèle est implémenté dans le paquetage `simulation` de DistMe. La Figure 47 le présente. La classe centrale, qui permet à l'utilisateur de modéliser la simulation qu'il souhaite distribuer, est `CSimulationDescription`. La simulation est représentée selon une approche boite noire. Ainsi, le modèle a été conçu pour correspondre à l'approche sac de travail (*bag of work*) utilisée en calcul distribué. L'utilisateur décrit chaque fichier d'entrée dont la simulation a besoin pour être exécutée ainsi que la commande de lancement pour la simulation.

Concernant les fichiers d'entrée, certains sont préexistants au moment du lancement de la distribution et d'autres sont générés lors du processus de distribution. Les fichiers déjà existants (ex. le fichier exécutable du simulateur sur le disque dur de la machine) sont décrits par des instances de la classe `CFile`. Les fichiers qui doivent comporter des

données spécifiques pour chaque job sont générés lors du processus de distribution de la simulation. Ils sont décrits par les métafichiers implémentés dans les classes dérivées de `CMetaFile`. A l'heure actuelle trois types de métafichiers sont disponibles :

- Les métafichiers statuts (i.e. `CStatusFile`) décrivent des fichiers statuts dans un format spécifique à une bibliothèque pour l'initialisation d'un générateur de nombres pseudo-aléatoires. Ils sont générés lors du processus de distribution de la simulation à partir de statuts sélectionnés dans un dépôt de statuts de DistTools. La génération des fichiers statuts est basée sur l'utilisation des paquetages `statusprovider` et `Serializer` de DistTools.

- Les métafichiers patrons ou « templates » (i.e. `CTemplateFile`) décrivent des fichiers textes générés à partir d'un patron dont une partie du contenu est substituée de façon unique pour chaque job lors du processus de distribution. Ces fichiers représentent par exemple des fichiers contenant des paramètres pour la simulation, comme le nom des fichiers de sortie ou le nom du fichier statut pour le générateur de nombres pseudo-aléatoires.

- Les métafichiers « scripts » (i.e. `CScript`) permettent de générer des fichiers de description de jobs pour différents environnements d'exécution. Ces fichiers servent à obtenir une simulation distribuée prête à être exécutée sur différents environnements de calcul distribué. Les classes héritant de la classe `CScript` génèrent des scripts de lancement en JDL (Job Description Language) pour gLite, en PBS (Portable Batch System) pour OpenPBS, ou en Python pour le logiciel de gestion d'exécution transversal à plusieurs environnements d'exécution distribuée : Ganga. Le support d'autres environnements d'exécution peut être obtenu en ajoutant des classes spécialisées, comme par exemple la possibilité de générer des projets g-Eclipse.

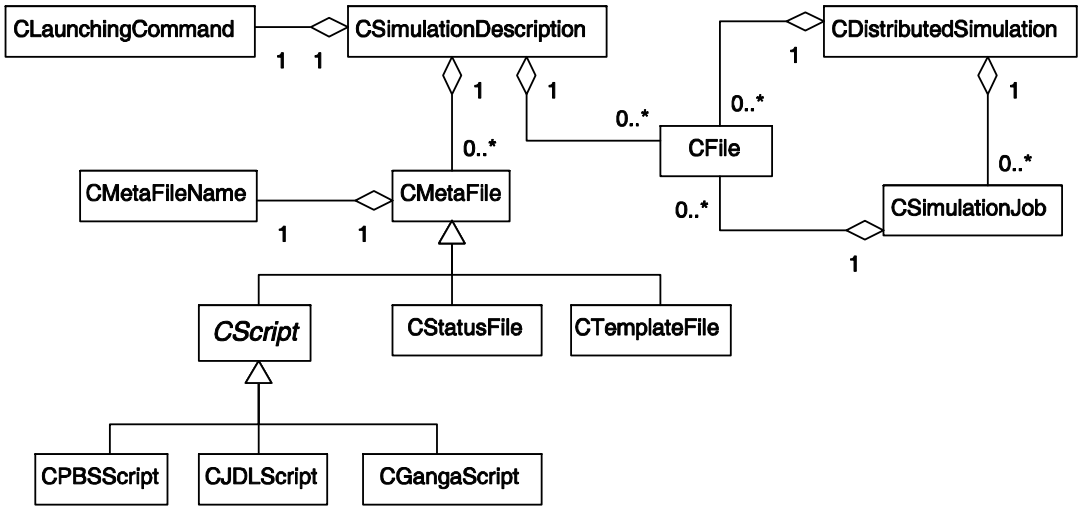


Figure 47 Diagramme UML de classe de la modélisation d'une simulation dans DistMe

Les métafichiers n'existent pas au moment où l'utilisateur décrit sa simulation. Ils n'ont par conséquent pas de nom. Cependant une instance de la classe `CMetaFileName` est systématiquement associée à une instance de `CMetaFile`. Celle-ci permet de passer le futur nom de ces fichiers en paramètre, pour paramétrer la commande de lancement d'un job ou pour replacer une partie du contenu du patron associé à un métafichier « template » par exemple. Tous les paramètres sont remplacés par leurs valeurs effectives lors du processus de distribution.

Une fois la phase de description terminée, l'application construit automatiquement une instance de `CDistributedSimulation`, représentant une version distribuée de la simulation de l'utilisateur. Elle est composée des fichiers communs nécessaires à l'exécution des jobs de la simulation distribuée. Le processus de distribution génère également des instances de `CSimulationJob` représentant chacune un job de la simulation distribuée. Ces instances sont composées de fichiers générés à partir des métafichiers et d'une commande de lancement en toute lettre. A la fin de la phase de distribution les métafichiers ont tous été transformés en fichiers représentés par des instances de la classe `CFile`.

VI.2 Le paquetage « Processor »

Dans DistMe, les simulations sont considérées comme des objets passifs de description. Le paquetage `Processor`, décrit dans la Figure 48, contient le code de

distribution et de préparation des simulations en vue de leur exécution en environnement distribué.

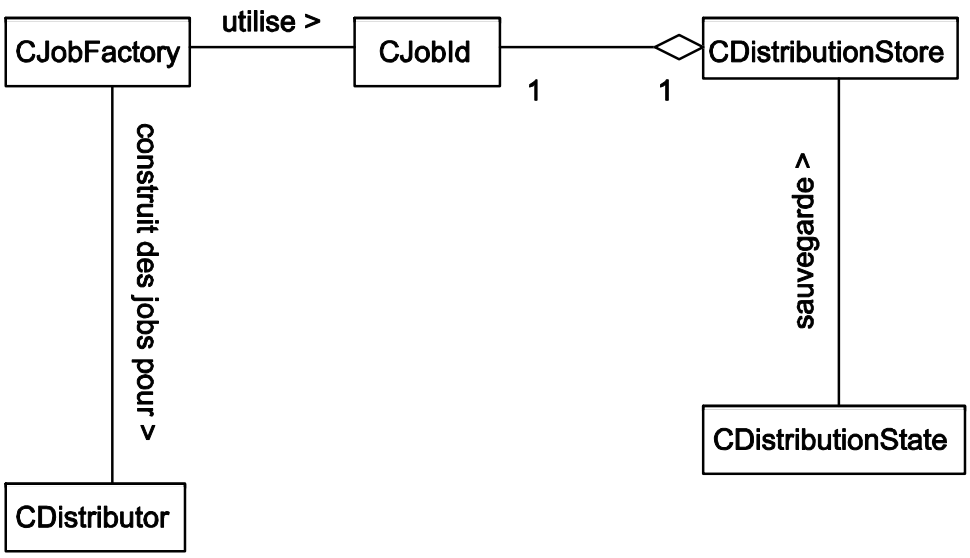


Figure 48 Diagramme UML de classe du paquetage « Processor » de DistMe

Ce paquetage permet de distribuer les simulations, de générer des fichiers à partir de métafichiers, d'explorer les paramètres, de générer les commande littérales de lancement des jobs ainsi que les fichiers de description des jobs pour l'environnement d'exécution cible. A l'heure actuelle, les langages de script supportés sont PBS (Portable Batch System) pour OpenPBS[47], JDL (Jobs Description Laguage) pour gLite[48] et Ganga[49].

La classe `CDistributor` est capable de générer une instance d'une simulation distribuée (i.e. de la classe `CDistributedSimulation`) à partir de sa description (i.e. une instance de la classe `CSimulationDescription`). La classe `CJobFactory` gère la construction des jobs. Les classes `CJobId`, `CDistributionStore` et `CDistributionState` permettent à l'utilisateur de sauvegarder l'état d'une distribution de simulation. L'utilisateur peut ainsi générer ultérieurement d'avantage de jobs pour une simulation distribuée ou de générer des jobs utilisant des statuts identiques à une simulation antérieurement distribuée pour des raisons de débogage.

Le progiciel de distribution DistMe permet la génération et la distribution de plan d'expériences. Cet aspect est décrit dans la partie sur le paquetage `Parameter` de DistMe.

[47] http://www.pbsgridworks.com
[48] http://glite.web.cern.ch/glite
[49] http://ganga.web.cern.ch/ganga

VI.3 Le paquetage « Parameter »

Le paquetage `Parameter` est un paquetage clef pour DistMe. Il permet de générer des fichiers et des commandes de lancement dissimilaires pour chaque job, du fait qu'un paramètre prendra des valeurs différentes pour chaque job. Ce paquetage permet non seulement une gestion souple des paramètres variables entre les jobs comme les noms de fichiers mais il permet de plus de générer des plans d'expériences pour faire de la conception expérimentale.

La conception expérimentale a été étudiée et exploitée depuis les années 30 par Sir Ronald Fisher dans le domaine de l'agriculture. Elle a été améliorée par George Box dans les années 50 pour être utilisée en chimie. Les ouvrages de référence dans ce domaine sont (Box, Hunter et Humter, Statistics for experimenters 1978) (Box et Draper, Empirical Model Building and Response Surfaces 1987). La communauté de la simulation trouva alors rapidement cette approche très prometteuse du fait que les modèles présentent souvent des centaines de facteurs contrôlables. La conception expérimentale en simulation a été largement étudiée par Kleijnen qui a fournit les ouvrage de références (Kleijnen, Statistical Tools for simulation practitioners 1987) (Kleijnen et Van Groenendaal, Simulation: A Statistical Perspective 1992).

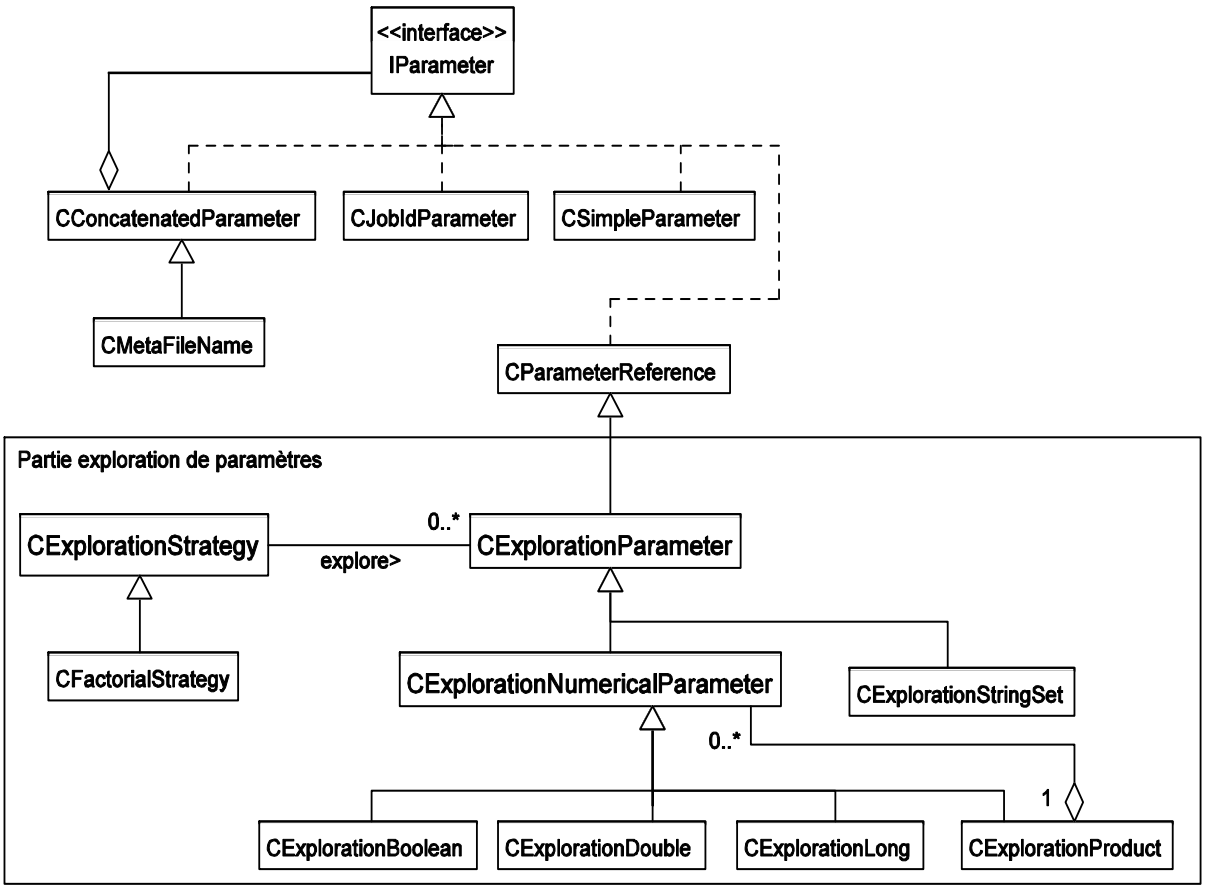


Figure 49 Diagramme de classe UML du paquetage « Parameter » de DistMe

La structure du paquetage `Parameter` est présentée Figure 49. L'interface `IParameter` est implémentée par toutes les classes de type paramètre. Elle impose l'implémentation de la méthode `getValue`. Celle-ci permet de récupérer la valeur d'un paramètre sous forme de chaîne de caractères au moment de la distribution.

La plus simple des classes « paramètre » est `CSimpleParameter`. Sa méthode `getValue` retourne toujours la même chaîne de caractères quelques soient les événements.

Dans DistMe, chaque job d'une simulation distribuée est identifié par un identifiant unique. La méthode `getValue` de la classe `CJobIdParameter` retourne une nouvelle valeur à chaque fois qu'un job est instancié au cours du processus de distribution. Par exemple, pour le 95ème job instancié lors de la distribution des réplications d'une simulation stochastique par DistMe, cette méthode renvoie la valeur « 0_95 ».

Nous avons vu précédemment l'utilité du paramètre `CMetaFileName`. Sa redéfinition de la méthode `getValue` retourne le nom d'un fichier. Ce nom de fichier change à chaque

nouveau job généré lors de la distribution de la simulation séquentielle en simulation distribuée. Un objet `CMetaFileName` sert par exemple à passer le nom d'un fichier statuts en argument de la commande de lancement de chaque job de la future simulation distribuée. Le fichier statuts est généré durant la phase de distribution ; il n'a pas encore de nom au moment de la description de la commande de lancement des jobs. L'utilisateur peut cependant utiliser le futur nom du fichier pour spécifier la commande de lancement des jobs. Pour ce faire, il utilise l'instance de la classe `CMetaFileName` associée à un objet `CStatusFile` pour paramétrer un objet `CLaunchingCommand`. A la fin du processus de distribution, la commande de lancement littérale pour un job de simulation distribuée intégrera le chemin vers le fichier statut généré pour ce job.

La classe `CConcatenatedParameter` est construite selon le patron de conception « composite ». Elle permet de concaténer des paramètres. La valeur de ce paramètre est la concaténation des valeurs littérales des paramètres qui le composent.

La deuxième partie du paquetage concerne l'exploration de plans d'expériences. Elle expose les sous-classes de `CExplorationParameter`. Ces paramètres peuvent être utilisés de manière similaire aux paramètres ordinaires (ex. utilisés pour décrire la commande de lancement des jobs). Ils ont l'avantage de permettre la description de plages de valeurs qui seront explorées selon une stratégie d'exploration. L'unique stratégie d'exploration implémentée à l'heure actuelle dans DistMe permet de mener une exploration de type « Full Factorial Design ». Elle explore l'intégralité des niveaux de des paramètres d'exploration choisis par l'expérimentateur. Par exemple, un paramètre `CExplorationStringSet` contient un ensemble de chaines de caractères à explorer. Si on définit un objet de type `CExplorationStringSet` contenant les niveaux « oui » et « non », la simulation sera distribuée deux fois et ce paramètre aura le niveau « oui » lors de la première distribution et « non » lors de la seconde. Les jobs issus de différentes distributions se verront fournir des statuts conduisant à la génération de séquences de nombres indépendantes entre elles.

Les paramètres du type `CExplorationNumericalParameter` permettent d'explorer des plages de nombres (ex. de 2,5 à 3,8 par pas de 0,1). Il est possible de décrire l'interdépendance de certains de ces paramètres en utilisant le paramètre d'exploration de type `CExplorationProduct`. La valeur d'un facteur de type `CExplorationProduct`

sera égale à tout moment au produit des niveaux des `CExplorationNumericalParameter` qui le compose et d'une constante.

Il est possible de concaténer les valeurs des paramètres d'exploration entre eux ou avec des paramètres simples en utilisant la classe `CConcatenatedParameter`.

VII Conclusion

La suite logicielle Dist propose des solutions concrètes aux problèmes liées à l'exécution de simulations stochastiques en environnement distribué. Elle propose tout d'abord une implémentation du concept de statuts génériques. Celle-ci permet de pallier à l'inadaptation des mécanismes de sauvegarde de l'état des générateurs de nombres pseudo-aléatoires pour des exécutions en environnement distribué des bibliothèques existantes : SSJ, JAPARA, CLHEP ou SPRNG.

Ainsi, les méta-informations proposées dans le concept de statut générique, et représentées sous formes de balise dans les fichiers statuts en XML, assurent une traçabilité par rapport à l'opération de parallélisation d'un générateur de nombres pseudo-aléatoires. Ces méta-informations sont concises (trois balises) et permettent cependant de décrire l'utilisation des séquences générées dans l'intégralité des cas de parallélisation décrit dans ce manuscrit (cf. chapitre I). Le risque d'erreur humaine dans l'utilisation des statuts générés est ainsi moindre et la rigueur accrue.

L'implémentation de la bibliothèque DistRNG permet de tirer le meilleur parti des statuts génériques en proposant :

- le choix de l'algorithme de génération de nombres pseudo-aléatoires au moment de l'exécution,
- la consultation et la mise à jour automatique des méta-informations sur les statuts permettant par exemple d'éviter à coup sûr les chevauchements lors de l'exécution distribuée d'un simulateur stochastique,
- plusieurs méthodes de parallélisation en accord avec l'état de l'art implémentées dans des classes exposant des interfaces simples d'utilisation,
- l'unique implémentation en java de l'algorithme de paramétrisation (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number

Generators 2000) pour le Mersenne Twister générique menant à la génération de séquences hautement indépendantes,

- une architecture extensible, la disponibilité d'un paquetage de formatage des nombres, la prise en charge systématique et transversale des opérations de sérialisation et désérialisation des statuts pour les générateurs au format XML.

La suite Dist propose de plus un deuxième logiciel nommé DistMe permettant la distribution non intrusive des simulations stochastiques selon une approche boite noire. L'expérimentateur qui utilise DistMe, décrit la simulation stochastique qu'il souhaite distribuer et le logiciel la prépare pour une exécution sur ferme de calcul ou grille de calcul.

DistMe fournit à l'utilisateur les fonctionnalités nécessaires pour générer :

- les fichiers d'entrée de chacun des jobs de simulation,
- un statut pour chaque job de simulation au format utilisé par le simulateur stochastique afin d'initialiser le générateur de nombres pseudo-aléatoires de manière à ce qu'il génère une séquence de nombres indépendantes de celles générées par les autres jobs,
- les fichiers de description des jobs pour l'environnement d'exécution cible, DistMe masque ainsi une partie de la complexité liée à l'utilisation d'environnements d'exécution distribués en générant des jobs prêts pour le lancement sur l'environnement cible.

DistMe permet la fourniture de statuts menant à la génération de séquences indépendantes pour chacun des jobs grâce à l'utilisation d'un dépôt contenant des statuts pour divers générateurs de nombres pseudo-aléatoires couplés à une stratégie de sélection choisie et paramétrée par l'utilisateur. DistMe supporte de plus l'exploration de plans d'expériences pour les simulations stochastiques.

DistMe et DistRNG présentent ainsi une solution générique, afin d'automatiser le travail de distribution des simulations stochastiques, et de permettre un maximum de rigueur tout en évitant les erreurs humaines. Ces logiciels proposent des techniques actuelles et en accord avec l'état de l'art pour la distribution de flux de nombres pseudo-aléatoires.

Nous allons dans la partie suivante montrer comment la suite Dist a été utilisée dans le cadre de réalisations scientifiques.

Chapitre 4 : Les applications

I Introduction

Le chapitre précédent présente la suite logicielle Dist. Celle-ci est composée des logiciels DistRNG et DistMe. DistMe permet la distribution automatique de simulations stochastiques sur des environnements d'exécution distribuée fondés sur le concept du sac de travail. La distribution d'une simulation est effectuée selon une approche boite noire afin de permettre de distribuer des simulateurs existants sans modifier leur code source. DistRNG est une bibliothèque de génération parallèle de nombres pseudo-aléatoires spécialement conçue pour la distribution de simulations stochastiques à grande échelle.

Ces logiciels ont été implémentés de manière itérative en s'appuyant sur des cas concrets d'utilisation. Ce chapitre présente tout d'abord l'« âge sombre » de la distribution des simulations stochastiques. Deux réalisations que nous avons menées avant de développer la suite Dist sont ainsi exposées et critiquées. Ces travaux nous ont permis de comprendre la nécessité d'outils adaptés pour la distribution de simulations stochastiques.

Les trois parties suivantes exposent des réalisations menées avec DistMe. La première est réalisée dans le but d'illustrer clairement le fonctionnement de DistMe. Les

deux suivantes correspondent à la distribution de deux simulateurs existants. Le premier est un simulateur utilisé en physique médicale, ce simulateur a été distribué sur deux fermes de calculs et la grille de calcul européenne EGEE. Le second est un simulateur pour la prévision de croissance de l'algue *Caulerpa taxifollia* en Méditerranée. DistMe a permis de réaliser l'exploration de certains paramètres de ce modèle et de distribuer les calculs sur une ferme de calcul.

Enfin, les deux dernières parties présentent DistRNG. L'avant dernière illustre les différentes fonctionnalités de cette bibliothèque à travers des extraits de code. La dernière présente une réalisation à grande échelle basée sur l'utilisation de DistRNG. Les résultats scientifiques de cette étude sont exposés et commentés.

II Les origines

En 2003, avant même de savoir que je ferai une thèse sur la distribution des simulations stochastique, un stage de deuxième année d'école d'ingénieur informatique m'a amené à paralléliser des simulations de Monte-Carlo sur grille de calcul. Cette partie présente ce travail et souligne, avec du recul, les faiblesses de celui-ci.

Dans une deuxième sous-partie je présente une étude de sensibilité à la qualité des séquences de nombres pseudo-aléatoires et aux corrélations inter-séquences d'une simulation distribuée dans le domaine de la physique médicale. Ce travail a permis de comprendre certaines limites des outils alors disponibles et ainsi de concevoir des solutions plus appropriées pour les études de sensibilité à la qualité des séquences nombres pseudo-aléatoires utilisés par des simulations séquentielles ou distribuées.

II.1 Parallélisation d'un simulateur en physique médicale

II.1.1 La génération des jobs de calcul

En 2003, nous avons travaillé sur la parallélisation de simulations de Monte-Carlo sur grille de calcul et la faisabilité d'un portail internet pour l'exécution distribuée de simulations en physique nucléaire. Les résultats de ce travail ont été publiés dans (Maigne, et al. 2004). Cette partie présente ce travail.

Les simulations de Monte-Carlo sont largement utilisées en physique médicale (Mackie 1990) (Rogers et Bielajew 1990) (Andreo 1991). La physique des particules ionisantes, utilisée pour les traitements médicaux est bien connue aujourd'hui.

Cependant, il est impossible de concevoir une expression analytique du transport des particules au travers de la matière. Du fait de la multitude et de la complexité des interactions au sein de la matière, il faut faire appel à des simulations de Monte-Carlo pour résoudre ce type problème de manière précise.

Les simulations de Monté Carlo en physique se basent sur la simulation du transport radiatif sachant la distribution de probabilité gouvernant chaque interaction des particules dans la matière. Différentes trajectoires ou historiques de particules peuvent ainsi être générées. Les simulations enregistrent ensuite les valeurs intéressantes pour un large nombre de trajectoires. Les trajectoires des particules émises lors de la simulation du transport radiatif sont indépendantes les unes des autres, leurs calculs peuvent ainsi être distribuées.

Dans le cas des applications de radiothérapie-brachythérapie, l'objectif est de calculer la distribution de doses adéquates pour un patient. La plupart des systèmes commerciaux nommés TPS (Treatment Planning Systems) utilisent des méthodes de calculs analytiques pour déterminer les distributions de doses. Les erreurs peuvent alors atteindre 10 à 20 pourcents. De tels codes sont très rapides (moins d'une minute d'exécution pour calculer une distribution de dose pour un traitement) et sont ainsi utilisables dans les centres médicaux. Bien que plus précises, les méthodes basées sur des simulations de Monte-Carlo ne peuvent à ce jour être utilisées dans un contexte clinique. La somme de calculs est top importante pour être exécutée sur une seule machine. Il y a ainsi un intérêt réel à distribuer les simulations de Monte-Carlo afin de produire des résultats très précis en physique médicales.

Le simulateur utilisé, GATE (Jan et al. 2004), est un simulateur pour la tomographie à émission. Il est écrit en C++ et comporte plus de 200 classes. Il est basé sur geant4 (Allison et al. 2006), une bibliothèque pour la simulation du passage de particules à travers la matière qui comprend plus de 2000 classes. Geant4 utilise pour la partie génération de nombres aléatoires, la bibliothèque CLHEP, conçue pour le calcul en physique des hautes énergies et regroupant 650 classes. CLHEP fournit notamment des algorithmes de génération de nombres pseudo-aléatoires. GATE utilise ainsi l'algorithme de génération de nombres pseudo-aléatoires par défaut de CLHEP, James Random (James 1990). Pour notre étude nous avons parallélisé ce générateur de nombres

pseudo-aléatoires par découpage en séquence. Ce choix a été fait dans le contexte suivant:

- Nous disposions d'un simulateur basé sur une bibliothèque de génération de nombres pseudo-aléatoires conçue pour la génération séquentielle de nombres pseudo-aléatoires et ne proposant pas de technique de parallélisation.
- Nous avions une méconnaissance des techniques de parallélisation de type indexation de séquences ou paramétrisation.
- Nous n'étions pas conscients de la faible qualité des séquences générées par le générateur James Random.

La première partie de l'étude visait donc à paralléliser ce générateur de nombres pseudo-aléatoires utilisé par le simulateur. Afin d'éviter les chevauchements possible lors de l'utilisation d'un générateur parallélisé par découpage en séquence, nous avons tout d'abord évalué la consommation en nombres pseudo-aléatoires de trois simulations de Monte-Carlo représentatives : une simulation de brachythérapie oculaire, une simulation de médecine nucléaire et une simulation de radiothérapie.

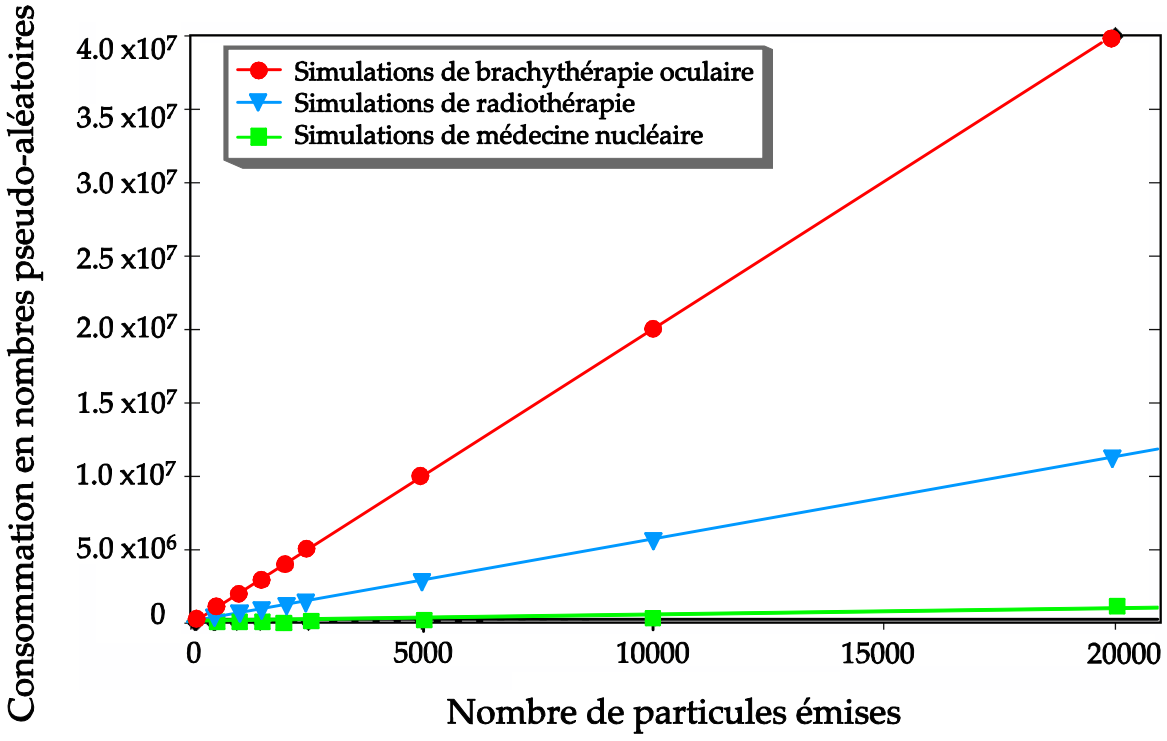


Figure 50 Consommation en nombres pseudo-aléatoires de différentes simulations de Monté Carlo de physique des particules

La Figure 50 montre la consommation en nombres pseudo-aléatoires pour un nombre de particules générées de plus en plus important pour chacune des trois simulations. La quantité de nombres pseudo-aléatoires consommée par le simulateur est entièrement dépendante du cas de simulation étudié et augmente de manière linéaire en fonction du nombre de particules générées. Afin de se placer dans le cas le plus large possible, nous avons pris comme consommation de référence la consommation la plus importante parmi les trois cas étudiés (la simulation en brachythérapie).

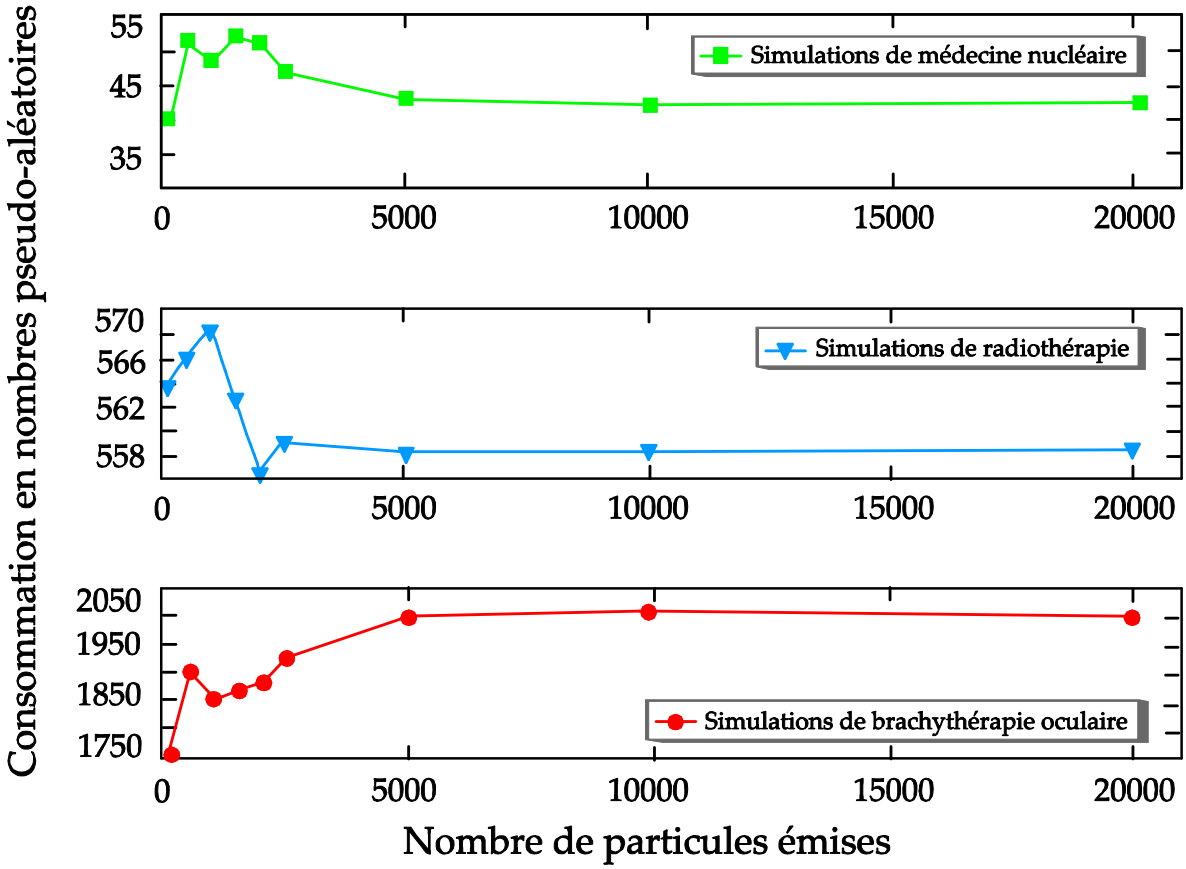


Figure 51 Consommation moyenne en nombres pseudo-aléatoires pour *n* particules émises dans trois cas de simulation en physique des particules

La Figure 51 présente la consommation moyenne en nombres pseudo-aléatoires par particule émise pour les trois cas de simulations précédemment évoquées. Cette figure montre des fluctuations importantes pour des petits jobs de simulations, qui simulent moins de 1000 particules. La consommation moyenne se stabilise après la simulation d'environ 10000 particules. Ceci nous à permis de dimensionner le nombre minimum de particules par job dans le cadre d'une simulation distribuée. En effet, en dessous de 10000 particules, le nombre de tirages de nombres pseudo-aléatoires peut varier significativement d'un job à l'autre ce qui réduit l'efficacité de la parallélisation par

découpage en séquence du fait de la nécessité de prendre en compte des marges d'erreur plus importantes dans l'estimation de la consommation des jobs de calcul. De plus cette étude nous apprend que dans le cadre de nos simulations l'estimation de la consommation en nombres pseudo-aléatoires doit se faire avec des simulations d'au moins 10000 particules et que 20000 suffisent pour avoir une bonne approximation de la consommation moyenne.

Nous avons ensuite parallélisé le générateur James Random par découpage en séquence pour des jobs de simulations simulant chacune 10 millions de particules dans le cas de la simulation la plus consommatrice (celle de brachythérapie), soit des séquences d'environ 20 milliards de nombres pseudo-aléatoires. Afin d'éviter tout chevauchement nous avons pris une marge de sécurité en générant des séquences de nombres espacés de 30 milliards de tirages. La génération de 200 statuts pour le générateur de nombres pseudo-aléatoires a nécessité environ 20 jours de calcul à pleine charge sur un Pentium 4 à 1,6 GHz.

Nous avons enfin écrit un code C++ spécifique à nos simulations afin de générer des jobs de simulation, comprenant la description de la simulation, l'initialisation du générateur de nombres pseudo-aléatoires, les scripts de lancements ainsi que les fichiers de description des jobs prêts pour l'exécution sur l'intergiciel EDG de la grille européenne qui s'appelait à l'époque Datagrid.

II.1.2 Les limites de ce travail

Ce travail représente un premier pas dans la parallélisation de simulation de Monte-Carlo sur grille de calcul. L'idée originelle était d'utiliser le mécanisme d'initialisation à partir d'un « seed », ou germe, implémenté dans la bibliothèque de génération de nombres pseudo-aléatoire CLHEP permettant d'obtenir des réplications indépendantes d'une simulation. Cependant, nous ne connaissions pas à l'époque les caractéristiques de ce mode de parallélisation. Nous avons utilisé un mode de parallélisation mieux maitrisé : le découpage en séquence. Il s'avère que ce choix était judicieux, le mécanisme d'initialisation à partir d'un germe de CLHEP implémente en effet une parallélisation par indexation de séquences à partir d'un générateur linéaire congruentiel de faible qualité.

Ce travail présente malgré tout plusieurs limites. Après 20 jours de calcul pour générer les statuts pour le générateur de nombres pseudo-aléatoires seulement 200 statuts étaient disponibles. Aujourd'hui l'organisation virtuelle pour le calcul biomédical

de la grille de calcul européenne EGEE regroupe plusieurs dizaines de milliers de machines permettant d'exécuter autant de jobs en parallèle et s'agrandit en permanence. Nous n'avions pas les moyens à l'époque d'utiliser cette puissance de calcul.

De plus, le générateur utilisé dans notre simulateur est James Random. Celui-ci n'avait pas été testé, ni avec des batteries de tests statistiques, ni en étudiant son impact sur nos résultats de simulation.

Enfin, le code de parallélisation de notre simulation et de génération des jobs développé pendant ce travail est spécifique :

- pour le simulateur GATE,
- pour la bibliothèque de génération de nombres pseudo-aléatoires, CLHEP,
- pour le générateur James Random,
- pour une parallélisation par découpage en séquence,
- pour la distribution sur la grille de calcul Datagrid,
- pour l'intergiciel EDG.

Le travail effectué pendant ma thèse - dont les parties III, IV et V de ce chapitre présentent des réalisations - a notamment permis de casser tous ces verrous technologiques pour la parallélisation automatique et générique de simulations stochastiques.

II.2 Test de sensibilité à la qualité du générateur de nombres pseudo-aléatoires

II.2.1 Protocole de test

Le générateur de nombres pseudo-aléatoires James Random utilisé par le simulateur GATE, est un générateur de faible qualité au regard des standards actuels. Il a été mis en défaut par la batterie de test « Crush » de TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007) en échouant à 60 tests sur 96. Ceci nous a mené à étudier l'impact de l'utilisation de James Random sur les résultats de simulations stochastiques en physique médicale. Pour ce faire, nous avons

établi un protocole de test en utilisant différents générateurs de nombres pseudo-aléatoires :

- un générateur de type linéaire congruentiel sur 32 bits de faible qualité,
- le générateur d'origine du simulateur, James Random (James 1990),
- le générateur Mesenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) qui passe la plupart des tests de la batterie « Crush » de testU01.

En plus de tester l'impact de la qualité intrinsèque des séquences de nombres pseudo-aléatoires utilisées, le protocole de test prévoyait de tester l'influence des corrélations inter-séquences sur les résultats de simulation à l'aide de séquences de nombres pseudo-aléatoires hautement corrélées en elles. Les corrélations inter-séquences ont étés introduites par chevauchement pour les deux générateurs James Random et Mersenne Twister.

II.2.2 Mise en place

Le logiciel GATE est programmé pour utiliser le générateur de nombres pseudo-aléatoires par défaut de la bibliothèque CLHEP (Lönnblad 1994). CLHEP n'a pas été conçue de manière à permettre le changement aisé du générateur de nombres pseudo-aléatoires par défaut et le nombre important (environ 650) des classes de CLHEP ne facilite pas cette opération.

Chacun des cas de test a ainsi nécessité la conception et le déploiement d'une version spécifique du simulateur basé sur une version de CLHEP différente. Le générateur par défaut de la bibliothèque CLHEP est instancié comme un attribut statique nommé `mainEngine` de la classe `HepRandom`. En modifiant cet attribut, il est possible de changer le type générateur par défaut pour n'importe quel autre générateur implémenté dans CLHEP. Pour chacun des cas de test nous avons ainsi dû modifier la classe `HepRandom` de CLHEP et recompiler la bibliothèque. De cette façon nous avons conçu trois versions de CLHEP utilisant par défaut :

- le générateur James Random,
- le générateur Mersenne Twister implémenté dans la class `MTwistEngine` de la bibliothèque,

- un générateur de type linéaire congruentiel, que nous avons implémenté.

L'ajout d'un générateur à la bibliothèque CLHEP n'est pas chose aisée. Nous avons ainsi dû ajouter une classe, concevoir un format de statut facilement lisible par CLHEP, modifier les fichiers `automake.ac`, `configure.in`, générer les fichiers `configure`, `makefile.in` et compilé la bibliothèque.

A l'issue de cette phase d'implémentation, nous disposions des trois versions de CLHEP, comprenant chacune un générateur de nombres pseudo-aléatoires par défaut correspondant à l'un de nos cas de test.

Pour chacune de ces versions, nous avons lancé la phase d'édition des liens avec une version précompilée du simulateur GATE. Nous avons ainsi obtenu trois versions de GATE, fonctionnant chacune avec un algorithme de génération de nombres pseudo-aléatoires.

Afin de permettre une exécution sur grille de calcul, les opérations de compilations et d'édition des liens ont été menées sur un ordinateur sous le système d'exploitation Scientific Linux 4, qui est le système utilisé sur les éléments de calcul de la grille EGEE.

II.2.3 Installation des versions de GATE sur la grille

L'installation d'un logiciel sur un élément de calcul de la grille EGEE utilise différents services. Ces services et leurs interactions sont exposés dans la Figure 52.

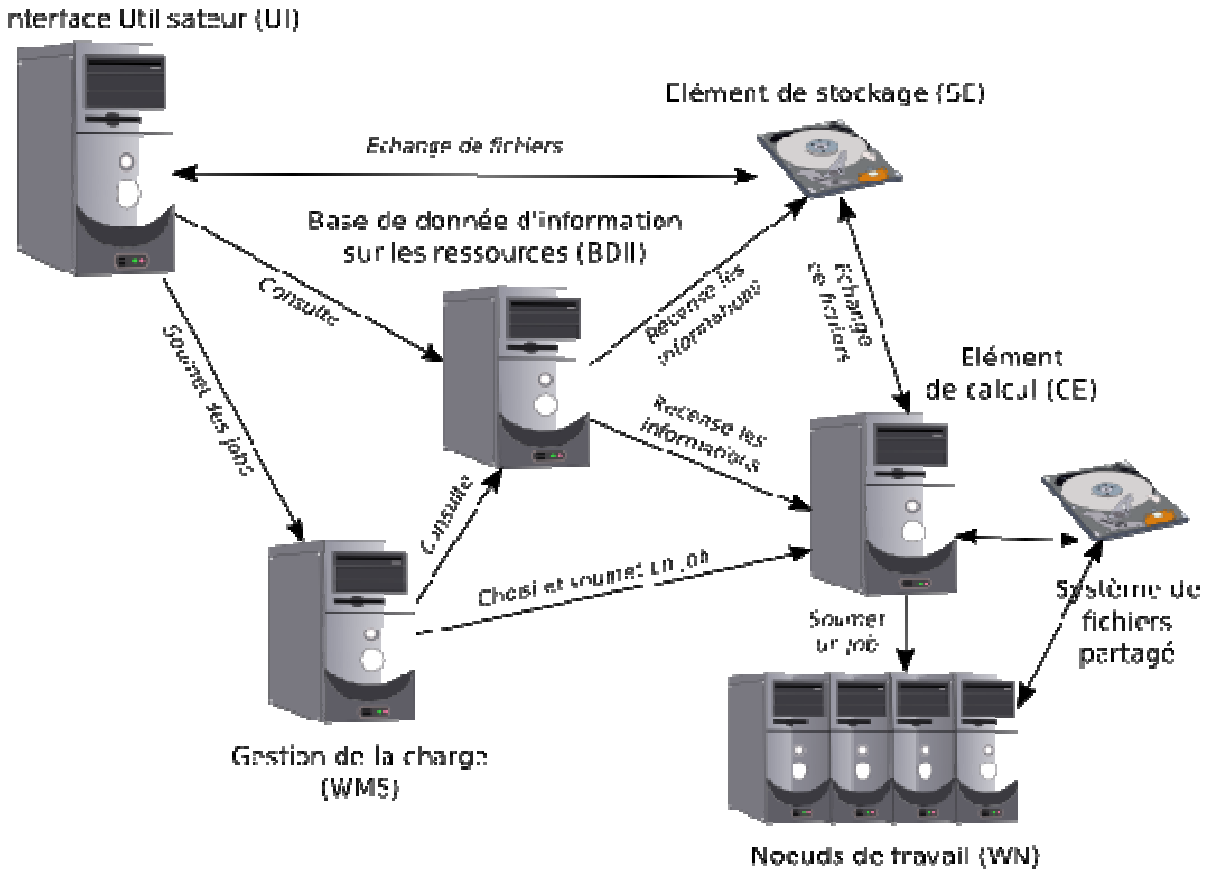


Figure 52 Schéma simplifié de l'architecture de la grille EGEE

En vue du déploiement de nos versions de GATE sur les éléments de calcul de l'organisation virtuelle « biomed » de la grille de calcul EGEE, nous les avons empaquetées dans des archives. Dans ces archives, nous avons ajouté des scripts d'initialisation de l'environnent ainsi que certaines bibliothèques nécessaires au bon fonctionnement de GATE comme `libXm.so.3`. Chaque archive occupe plus de 300 Mo d'espace disque.

Les archives ont ensuite été copiées sur un élément de stockage de la grille. La réalisation de cette opération nécessite de s'authentifier sur la grille en se connectant à une interface utilisateur de grille puis de générer un proxy d'authentification en tant que simple utilisateur (par opposition à gestionnaire de logiciel ou « software manager ») de l'organisation virtuelle « biomed ». L'initialisation du proxy est illustrée Code 14.

```
[reuillon@ouioui install]$ voms-proxy-init -voms biomed:/biomed/lcg1
Enter GRID pass phrase:
Your identity: /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon
Creating temporary proxy ..................................... Done
```

```
Contacting  cclcgvomsli01.in2p3.fr:15000
[/O=GRID-FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr] "biomed" Done
Creating proxy ................................ Done
Your proxy is valid until Mon May 26 03:05:52 2008
```

Code 14 Initialisation du proxy d'authentification sur la grille de calcul EGEE

Le choix du mode utilisateur s'effectue en passant un FQAN (Fully Qualified Attribute Name) qui permet de choisir un groupe et un rôle dans l'organisation virtuelle. Ici le FQAN `/biomed/lcg1` signifie le choix du groupe `lcg1`. Le rôle choisi est le rôle par défaut pour ce groupe : `role=NULL`. Comme l'illustre le Code 15, il est possible d'afficher les informations sur un proxy d'authentification avec la commande `voms-proxy-info`.

```
[reuillon@ouioui ~]$ voms-proxy-info -all
subject   : /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon/CN=proxy
issuer    : /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon
identity  : /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon
type      : proxy
strength  : 512 bits
path      : /tmp/x509up_u597
timeleft  : 11:59:58
=== VO biomed extension information ===
VO        : biomed
subject   : /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon
issuer    : /O=GRID-FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr
attribute : /biomed/lcg1/Role=NULL/Capability=NULL
attribute : /biomed/Role=NULL/Capability=NULL
timeleft  : 11:59:58
```

Code 15 Affichage des informations sur le proxy d'authentification précédemment créé

Les groupes et les rôles disponibles pour un utilisateur peuvent être affichés à l'aide de la commande `voms-proxy-list`. Le Code 16 montre un exemple d'utilisation de cette commande. Dans ce cas je peux prendre le rôle `NULL`, ou `lcgadmin` ou choisir le groupe `lcg1`.

```
[reuillon@ouioui ~]$ voms-proxy-list -voms biomed
Enter GRID pass phrase:
```

```
Your identity: /O=GRID-FR/C=FR/O=CNRS/OU=LIMOS/CN=Romain Reuillon
Creating temporary proxy ..................................... Done
Contacting  cclcgvomsli01.in2p3.fr:15000 [/O=GRID-FR/C=FR/O=CNRS/
                        OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr] "biomed" Done
Available attributes:
/biomed/Role=NULL/Capability=NULL
/biomed/Role=lcgadmin/Capability=NULL
/biomed/lcg1/Role=NULL/Capability=NULL
```

Code 16 Affichage des rôles et groupes disponibles pour un utilisateur

Une fois le proxy d'authentification initialisé, il faut sélectionner un élément de stockage disponible en vue de copier les archives d'installation sur la grille de calcul. La commande `lcg-infosites` dont l'utilisation est présentée au travers du Code 17 permet de lister des éléments de stockage disponibles et leurs capacités.

```
[reuillon@ouioui GATELCG]$ lcg-infosites --vo biomed se
Avail Space(Kb) Used Space(Kb)  Type    SEs
----------------------------------------------------------
1929044324      2751667740      n.a     scaise-2.scai.fraunhofer.de
256110000       1979244         n.a     se2.egee.cesga.es
2030000000      1135140         n.a     fornax-se.itwm.fhg.de
361550000       926705          n.a     se02.marie.hellasgrid.gr
…
2550000000      374756          n.a     egee-se.grid.niif.hu
1423536800      123433312       n.a     g03n05.pdc.kth.se
968640000       212229688       n.a     se01.kallisto.hellasgrid.gr
858730000       233151          n.a     SE.pakgrid.org.pk
```

Code 17 Exemple d'utilisation de la commande lcg-infosite

Une fois l'élément de stockage sélectionné la commande `lcg-cr` permet de copier l'archive du logiciel GATE sur un élément de stockage et l'enregistrement d'un nom de fichier logique, pour accéder au fichier dans le catalogue de fichiers de la grille. Son l'utilisation est présentée au travers du Code 18

```
[reuillon@ouioui GATELCG]$ lcg-cr --vo biomed file:/home/prof/reuillon/tmp/
GATES/GATELCG/GateLCG.tgz -d fornax-se.itwm.fhg.de
```

```
guid:e34eb631-3054-4635-b4d1-13e7c8b07b42
```

Code 18 Exemple d'utilisation de la commande lcg-cr

On obtient ainsi un `guid` qui est un identifiant unique du fichier sur la grille de calcul. On peut alors écrire le script d'installation. Celui-ci sera utilisé par les jobs d'installation pour déployer une version (correspondant à l'utilisation d'un générateur de nombres pseudo-aléatoires) de GATE sur les éléments calcul. L'exécution du script présenté dans le Code 19 par un élément de calcul de l'organisation virtuelle `biomed` de la grille EGEE permet d'installer le logiciel GATE sur cet élément de calcul. L'archive précédemment envoyée sur un élément de stockage de la grille est téléchargée par l'élément de calcul puis extraite dans le répertoire `VO_BIOMED_SW_DIR` qui correspond au répertoire d'installation de logiciels pour l'organisation virtuelle `biomed`.

```
#!/bin/bash

cd $VO_BIOMED_SW_DIR
mkdir $VO_BIOMED_SW_DIR/dist
rm -rf $VO_BIOMED_SW_DIR/dist/LCGGate
mkdir $VO_BIOMED_SW_DIR/dist/LCGGate
lcg-cp --vo biomed guid:e34eb631-3054-4635-b4d1-13e7c8b07b42
                            file:$VO_BIOMED_SW_DIR/dist/LCGGate/LCGGate.tgz
cd $VO_BIOMED_SW_DIR/dist/LCGGate
tar -xvzf LCGGate.tgz
rm LGCGate.tgz
```

Code 19 Script d'installation de GATE

Nous avons alors écrit un script de description de job par élément de calcul sur lequel nous souhaitions installer GATE en JDL (Job Decription Language). La partie `Requirements` du script JDL permet de choisir la machine destination du job. Cette fonctionnalité est très utile pour les jobs d'installation. Le Code 20 présente un exemple d'un JDL décrivant une tâche d'installation sur l'élément de calcul `clrlcgce01.in2p3.fr`.

```
executable="/bin/sh";
Arguments = "-x install.sh";
StdOutput = "install.out";
```

```
StdError = "install.err";
InputSandBox = {"install.sh"};
OutputSandbox = {"install.out","install.err"};
Requirements = (other.GlueCEInfoHostName ==
                                    " clrlcgce01.in2p3.fr");
```

Code 20 Jdl d'installation de GATE sur un élément de calcul

Une fois les jobs d'installation décrits il faut générer un nouveau proxy d'authentification en tant que gestionnaire de logiciel. Il est alors possible de lancer les jobs d'installation. Les jobs lancés en tant que gestionnaire de logiciel contrairement aux autres jobs sont exécutés directement sur les machines d'accès aux éléments de calcul (Computing Elements) et non pas sur les nœuds de travail (Worker Node).

Une fois les jobs terminés il faut valider l'installation sur chaque élément de calcul. Pour ce faire, on vérifie que le fichier contenant la sortie d'erreur standard du job d'installation correspondant est vide. On exécute ensuite un job de test sur l'élément de calcul. Les éléments de calcul de la grille EGGE ne sont pas parfaitement homogènes. La présence et les versions des librairies installées peuvent varier d'un élément de calcul à l'autre. Il est donc nécessaire de tester les installations afin d'éviter l'échec au moment de l'exécution de tous les jobs exécutés sur un élément de calcul comportant une installation défaillante. Une fois un élément de calcul testé, il faut le « marquer » en lui associant un « tag » à l'aide de la commande présentée dans le Code 21.

```
[reuillon@ouioui install]$ lcg-ManageVOTag -host clrlcgce01.in2p3.fr -vo
biomed --add -tag VO-biomed-LCGGate
The following tags are being added to
 //clrlcgce01.in2p3.fr/opt/edg/var/info/biomed/biomed.list:
                                                        VO-biomed-LCGGate
```

Code 21 Ajout d'un « tag » sur un élément de calcul

Une fois les éléments de calcul repérés par un « tag » il est possible de restreindre les éléments de calcul de destination des jobs GATE à ceux aptes à les recevoir. Il suffit d'ajouter la ligne présentée dans le Code 22 afin d'assurer l'exécution du job correspondant sur un élément de calcul sur lequel le tag `VO-biomed-LCGGate` est présent et donc la version de GATE basée sur le générateur linéaire congruentiel est installé.

```
other.GlueCEPolicyMaxCPUTime >= 600 && Member("VO-biomed-LCGGate"
                      ,other.GlueHostApplicationSoftwareRunTimeEnvironment)
```

Code 22 « Requirement » pour l'exécution d'un job sur un élément de calcul taggué avec « VO-biomed-LCGGate »

II.2.4 Automatisation de la procédure d'installation avec Ganga

La procédure d'installation que nous avons utilisée est fastidieuse. Nous l'avons ainsi partiellement automatisée en utilisant le logiciel Ganga présenté dans le chapitre 2 de ce manuscrit.

```
------------------------------ce.py---------------------------

CEs=['CE.pakgrid.org.pk',
'ComputingElement',
'ares02.cyf-kr.edu.pl',
…
'beagle14.ba.itb.cnr.it',
'cclcgceli04.in2p3.fr']

----------------------------install.py-------------------------

execfile('ce.py')

for ce in CEs:
  lcg=LCG()
  lcg.requirements.other = ['(other.GlueCEInfoHostName == "'+ce+'")']

  print lcg

  j = Job(backend=lcg)
  j.application = Executable(exe='/bin/bash',args=['install.sh'])
  j.inputsandbox=['install.sh']
  j.outputsandbox=['install.out', 'install.err']
  j.name=ce

for j in jobs :
  j.submit()
```

Code 23 Génération automatique des jobs d'installation en python avec ganga

Le Code 23 permet de générer et de soumettre automatiquement un job d'installation par élément de calcul. Ganga gère ensuite l'exécution des jobs et met à jour leurs statuts. Il est alors possible dans un deuxième temps de soumettre de la même manière les jobs de test et d'extraire la liste des jobs réussis afin de « tagger » les éléments de calcul correspondants.

II.2.5 Résultats scientifiques

Une fois les trois versions de GATE installées sur la grille, nous avons exécuté les simulations GATE utilisant les différents générateurs de nombres pseudo-aléatoires. Ces simulations terminées, nous avons compilé les résultats.

Des images résultats représentant une coupe de l'objet reconstruit par simulation sont présentées Figure 53. Le résultat idéal est présenté en haut à gauche. L'image obtenue en utilisant le générateur congruentiel linéaire, présentée en haut à droite de la figure est bruitée et inutilisable. Les deux images obtenues en utilisant les générateurs James Radom et Mersenne Twister, en bas de la figure, semblent de qualité équivalente.

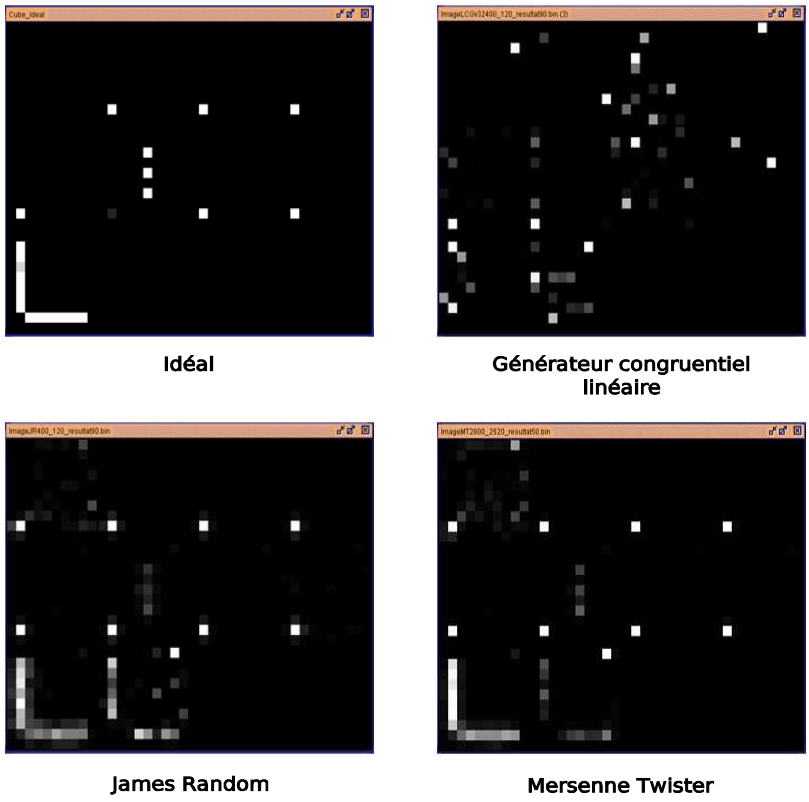


Figure 53 Résultats d'une simulation GATE utilisant différents générateurs de nombres pseudo-aléatoires

L'équivalence des deux images en terme de qualité a été vérifié dans un deuxième temps par des tests quantitatifs présentés Figure 54. Pour chacun des deux générateurs, un ratio mesurant la précision du résultat à été calculé. Plus ce ratio est proche de 1 plus le résultat est proche de celui attendu. Les courbes calculées pour les simulations effectuées en utilisant James Random et Mersenne Twister ne présentent qu'un écart très faible.

Pour les images reconstruites à partir des simulations utilisant des séquences volontairement corrélées, les résultats sont probants. La Figure 55 présente une comparaison des images reconstruites à partie des simulations utilisant des séquences de nombres pseudo-aléatoires faiblement corrélées à gauche et fortement corrélées à droite. Il est visible que les images reconstruites sont inutilisables. Les corrélations entre les séquences de nombres utilisées par chacun des jobs ont empêché la convergence de la simulation globale et biaisé les résultats.

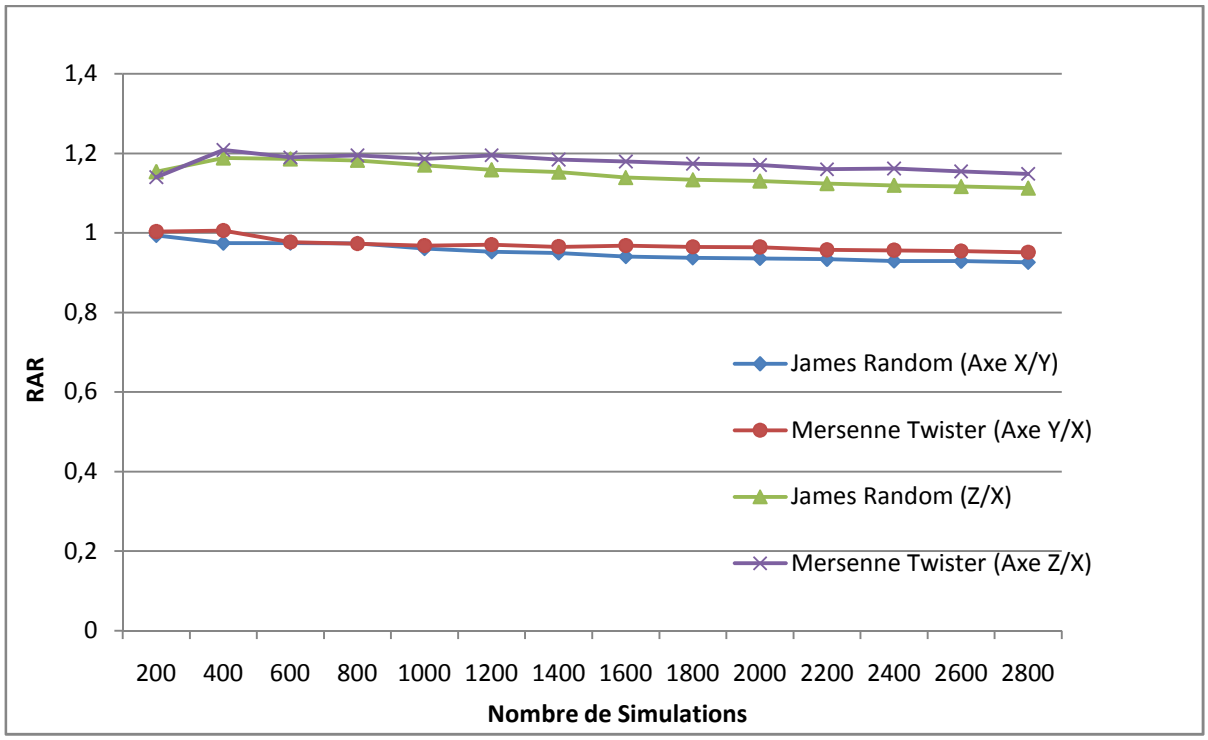


Figure 54 Mesure quantitative de l'impact de l'algorithme de génération de nombres pseudo-aléatoires sur les résultats de la simulation

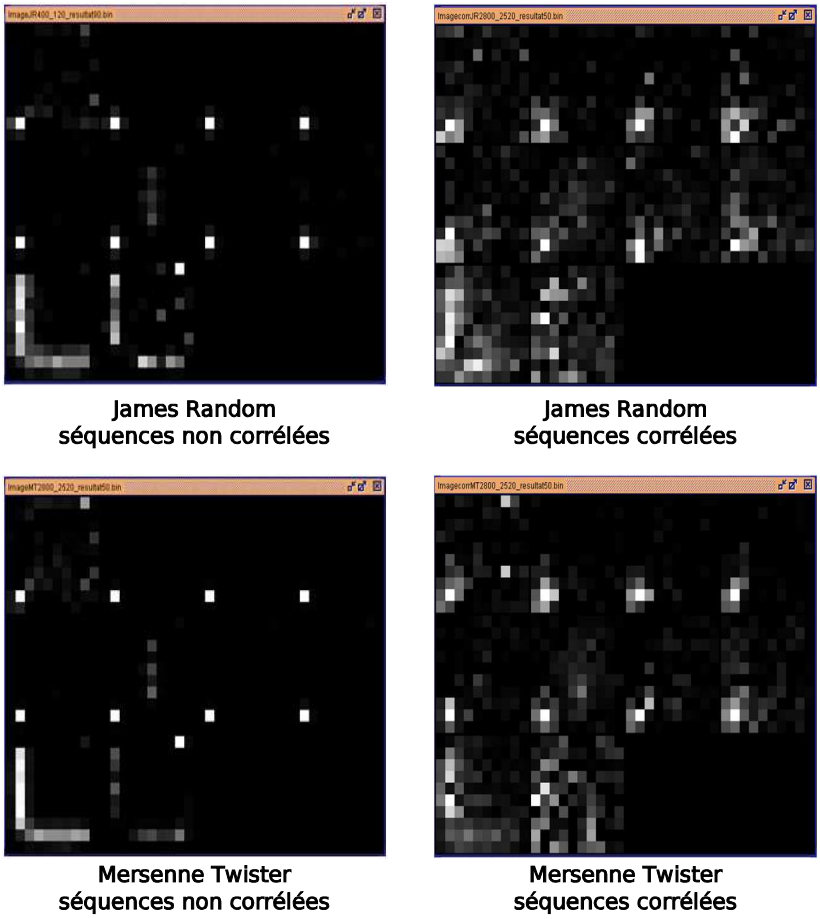


Figure 55 Effet des corrélations croisées sur les résultats d'une simulation de reconstruction avec le logiciel GATE

II.2.6 Les limites

L'étude présentée dans cette partie montre que, dans notre cas, les générateurs Mersenne Twister et James Random ne biaisent pas significativement les résultats de la simulation. Cette étude est importante et constitue le meilleur test pour mesurer l'impact d'un générateur de nombres pseudo-aléatoires sur des résultats de simulation (L'Ecuyer, 1998). Cependant, elle a nécessité un travail considérable car la bibliothèque de génération de nombres pseudo-aléatoires CLHEP est conçue selon une architecture peu adaptée à ce genre d'analyse.

A ma connaissance, aucune bibliothèque de génération de nombres pseudo-aléatoires ne prévoit la possibilité de choisir ou de changer l'algorithme de génération à l'exécution. Il a ainsi fallu pour chacun des algorithmes de génération de nombres pseudo-aléatoires, modifier le code source, recompiler la bibliothèque de génération et installer la nouvelle version du simulateur sur les éléments de calculs de la grille. Ces étapes préliminaires sont coûteuses en temps et demandent un effort important.

En outre, bien que les simulations basées sur GATE représentent pour la plupart des temps de calcul importants et nécessite d'être exécutées dans des environnements de calcul distribué, CLHEP est une bibliothèque de génération de nombres pseudo-aléatoires séquentielle. Hormis le mécanisme rudimentaire d'initialisation des générateurs avec un générateur linéaire congruentiel de nombres pseudo-aléatoires, CLHEP ne fournit pas d'implémentation des techniques de génération de séquences de nombres pseudo-aléatoires parallèles.

La partie VI de ce chapitre expose des réalisations faite en utilisant DistRNG. DistRNG permet de solutionner les problèmes rencontrés dans cette partie, et dans un cadre plus large de faciliter le test de sensibilité d'une simulation à la qualité des séquences de nombres générées par un algorithme de génération de nombres pseudo-aléatoires.

III Utilisation de DistMe pour la parallélisation d'une simulation stochastique

III.1 Implémentation de la simulation stochastique

Cette partie présente DistMe d'un point de vue utilisateur en prenant pour exemple la parallélisation d'une simulation stochastique d'un calcul de π sur une grille de calcul. DistMe utilise BeanShell[50], un langage de script dérivé de java. Les extraits de codes présentés dans cette partie sont du code BeanShell exécutable dans l'éditeur de script de DistMe dans sa version 3.0.

L'objectif de cette partie est d'illustrer comment il est possible de distribuer une simulation stochastique avec la suite DistMe. Pour ce faire, je prends tout d'abord un exemple simple de la distribution d'une simulation stochastique de calcul de π. Le Code 24 présente une implémentation de cette simulation basé sur l'utilisation du générateur de nombres pseudo-aléatoires Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) implémentée dans la bibliothèque de classes pour la physique des hautes énergies CLHEP (Lönnblad 1994) . Le programme de calcul de π accepte deux paramètres : le chemin vers un fichier statut pour le générateur Mersenne Twister 1997 au format CLHEP et le nom du fichier de sortie. Il initialise le générateur avec le

[50] http://www.beanshell.org/

statut et calcule une estimation de π en tirant 100 millions de points aléatoirement dans un carré unitaire. Il écrit ensuite le résultat dans le fichier de sortie. Dans le cadre de notre simulation distribuée, ce code constitue une réplication de la simulation globale.

```
#include <iostream>
#include "CLHEP/Random/MTwistEngine.h"

#define NB_PTS 100000000

using namespace CLHEP;
using namespace std;

int main (int argc, char** argv) {
      long inside = 0;
      HepRandomEngine* engine = new MTwistEngine();
      ostream  * os = &cout;

      if(argc >= 2) {
            engine->restoreStatus(argv[1]);
      }

      if(argc >= 3) {
            ofstream * of = new ofstream();
            of->open(argv[2]);
            os = of;
      }

      for(long i = 0; i < NB_PTS; i++) {
            double x = engine->flat();
            double y = engine->flat();
            if((x * x + y * y) < 1) inside ++;
      }

      *os << ((double) inside / NB_PTS) * 4 << endl;

      if(os != &cout) delete os;
      delete engine;
}
```

Code 24 Code de calcul en C++ de π par la méthode de Monté-Carlo

III.2 La génération de statuts pour le générateur de nombres pseudo-aléatoires

Afin de distribuer le calcul de π nous allons paralléliser le générateur de nombres pseudo-aléatoires. Il n'existe pas à l'heure actuelle de technique de division de cycle efficace qui permettrait d'avancer de plusieurs étapes de génération dans le cycle du Mersenne Twister 19937 en un temps de calcul raisonnable pour ce générateur. Il est donc inefficace de paralléliser ce générateur par saut de grenouille (« leap frog »). De plus, ce générateur est une version déjà paramétrée du générateur générique Mersenne Twister (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000), il n'est donc pas paramétrable. Reste à notre disposition l'indexage en séquence, qui consiste à initialiser le générateur à l'aide d'un autre générateur de nombres pseudo-aléatoires ou le découpage en séquence. Dans cet exemple nous allons opter pour le découpage en séquence.

La simulation précédente consomme 200 millions de nombres pseudo-aléatoires par réplication (2 tirages pour chaque point dans $\mathbb{R}^2$). Nous allons générer des statuts espacés de seulement 100 millions de tirages. Afin d'illustrer le principe de sélection rigoureuse et automatique des statuts nous confierons au logiciel DistMe le soin de sélectionner des statuts adaptés à notre simulation lors du processus de distribution. Le Code 25 implémente en C++ la génération des statuts correspondants à une parallélisation du générateur Mersenne Twister 19937 par découpage en séquence. La génération de 100 statuts nécessite environ 11 minutes d'exécution sur un Pentium 4 cadencé à 2,8 GHz.

```
#include <iostream>
#include "CLHEP/Random/MTwistEngine.h"

#define SPACING 100000000
#define NB_SEQUENCES 100

using namespace CLHEP;
using namespace std;

int main (int argc, char** argv) {
```

```
	HepRandomEngine* engine = new MTwistEngine();
	char genericStatusName[] = "statusMT_%ld.stat";
	char statusName[256];

	for(long nbSequence = 0 ; nbSequence < NB_SEQUENCES ; nbSequence++)
	{
		sprintf(statusName,genericStatusName,nbSequence);
		engine->saveStatus(statusName);
		for(long i = 0; i < SPACING; i++) {
			engine->flat();
		}
	}

	delete engine;
}
```

Code 25 Code C++ de génération de statuts pour le générateur Mersenne Twister 19937 de la bibliothèque CLHEP

III.3 L'initialisation de la base de données de DistMe

Dans cet exemple nous distribuons une simulation stochastique à l'aide du logiciel « DistMe ». Comme décrit dans le chapitre 3 de ce manuscrit ce logiciel utilise les mécanismes de gestion de statuts pour les générateurs de nombres pseudo-aléatoires implémentés dans la librairie « DistTools ». Ces mécanismes permettent la sélection rigoureuse de statuts dans un dépôt de statuts (voir partie « flux de nombres pseudo-aléatoires parallèles » du chapitre 3). Dans cet exemple nous utiliserons le dépôt de statuts local. Les statuts seront stockés dans une base de données sur le disque dur de la machine sur la laquelle « DistMe » est exécuté. Le dépôt de statuts local est implémenté dans la classe `CLocalStatusProvider`. Préalablement à son utilisation dans des scripts de distribution de simulations stochastiques, il est obligatoire d'insérer des statuts dans la base de données.

Les Code 26 et Code 27 présentent les opérations d'insertion des statuts dans la base de données. Avant d'être inséré dans la base, un statut sous forme de fichier ou de chaîne de caractères doit être transformé en un objet statut. C'est l'opération de désérialisation. La désérialisation de statuts à partir de différents formats de fichiers générés par les classes de CLHEP est implémentée dans DistTools. La classe `CStatusSerializerCLHEPMT19937v32` de DistTools implémente le code de

désérialisation des statuts générés par la classe `MTwistEngine` de CLHEP. Le Code 26 permet la déserialisation d'un fichier statuts au format CLHEP pour construire un objet « `CStatut` » de la bibliothèque « `DistTools` ». Ce dernier pourra alors être inséré dans la base de données. La classe « `CStatus` » est une implémentation du concept de statut générique présenté dans le chapitre 3. Elle contient deux parties : l'une servant à initialiser l'algorithme de génération de nombres pseudo-aléatoires et l'autre renseignant les métadonnées pour vérifier l'utilisation correcte du statut lors d'une exécution distribuée. Dans le paragraphe « La génération de statuts pour le générateur de nombres pseudo-aléatoires » de cet exemple, nous avons parallélisé le Mersenne Twister 19937 dans sa version 32 bits par découpage en séquence. Dans ce cadre, le nombre d'itérations du Mersenne Twister, entre la génération de deux statuts, est une métadonnée essentielle afin de garantir l'absence de chevauchement lors de l'utilisation des statuts. Les fichiers statuts générés par CLHEP ne contiennent pas de méta-informations hormis le nom du fichier. Nos statuts sont numérotés de 0 à 100. Dans notre exemple, le fichier « statusMT_0.stat » correspond au statut origine de distance zéro. DistTools fournit la classe `CFileNameExtractor`, qui permet de déduire la métadonnée « distance » à partir du nom du fichier statuts. Le tableau de chaînes de caractères `reject` présente une liste d'expressions régulières qui doivent être exclues du nom du fichier pour obtenir le numéro du fichier. Ce numéro est multiplié par un facteur constant de 100 millions de tirages. Les autres métadonnées, c'est-à-dire le nom du groupe de sous-séries indépendantes, et le nom de la sous-série, sont fixées à `MT_CLHEP_TUTO` et `NO_INIT`.

```
String [] reject = {"[a-z]","[A-Z]","\\.","_"};
CFileNameExtractor calculator = new CFileNameExtractor(reject,100000000L);
CStatusSerializer deserializer = new CStatusSerializerCLHEPMT19937v32(
                                 calculator , "NO_INIT" , "MT_CLHEP_TUTO" );
```

Code 26 Construction d'un sérialiser pour les statuts générés par la classe MTwistEngine de CLHEP

La classe `CLocalStatusProvider` contient une méthode `populate` qui permet d'insérer dans la base de données tous les statuts contenus dans un répertoire du disque dur. Le Code 27 présente l'appel à la méthode `populate`. Cette méthode liste tous les

fichiers du répertoire `dir` dont le nom correspond à l'expression régulière `filter`, puis les désérialise et les insère dans la base de données.

```
String dir = "/home/reuillon/install/CLHEP/status";
String filter =  "statusMT_[0-9]*\\.stat";
CLocalStatusProvider.GetInstance().populate( dir, deserializer, filter );
```

Code 27 Code d'insertion des statuts dans la base de données locale de DistMe

Cette opération d'insertion de statuts permet de disposer en local d'une base de statuts utilisable pour de futures distributions de simulations. Elle est pour le moment une étape obligatoire pour utiliser DistMe. Cependant un service web a été développé afin de centraliser la gestion des statuts et de permettre aux utilisateurs du monde entier de se servir de DistMe sans cette étape préparatoire. A l'heure où j'écris ces lignes, le service web est fonctionnel et attend d'être déployé. Une fois mis en service, celui-ci fournira des statuts pour les générateurs de nombres pseudo-aléatoires les plus courants. Les opérations de parallélisation des générateurs de nombres pseudo-aléatoires seront ainsi factorisées.

III.4 La distribution de la simulation

Cette partie décrit la distribution de la simulation stochastique de calcul de π avec DistMe. Le Code 28 est le début du code BeanShell de distribution de la simulation stochastique. Ce code initialise le distributeur de simulation `distributor` et un paramètre de type `CJobIdParameter`. Ce paramètre représente l'identifiant unique d'un job lors du processus de distribution. Il peut servir à définir un nom de fichier, par exemple `output1_30.out` correspond au nom du fichier de sortie du 30ème job de la simulation distribuée qui a pour identifiant `1_30`.

```
//Instanciation du distributeur
distributor = new CDistributor();
id = new CJobIdParameter(distributor);

//Répertoire de travail
String dir = "/home/reuillon/tmp/pi/";
```

Code 28 Début du code BeanShell de distribution de la simulation stochastique de calcul de π

Le Code 29 présente la suite du script de distribution. Il décrit les fichiers d'entrée commun à l'ensemble des jobs de simulation. Dans notre cas, seul l'exécutable `pi` est nécessaire au bon fonctionnement des jobs de simulation.

```
// Définition des fichiers d'entrée
inputFiles = new CFiles();
inputFiles.add(dir + "pi");
```

Code 29 Définition des fichiers d'entrée

Le Code 30 présente la définition des métafichiers d'entrée. Ces métafichiers sont générés par DistMe au moment de la distribution de la simulation. Ils n'existent pas encore au moment où l'utilisateur rédige le script de distribution. Dans notre cas, un fichier statut et le fichier qui représente un job seront générés. Les fichiers statuts sont extraits de la base de données locale selon une stratégie de découpage en séquence parmi les statuts pour le générateur Mersenne Twister 19937 version 32 bits. La distance minimale entre deux statuts est fixée à 200 millions d'itérations de l'algorithme. La base de données que nous avons initialisée précédemment contient des statuts espacés de 100 millions d'itération ; un statut sur deux sera donc sélectionné pour le processus de distribution. Quand aux scripts de définition des jobs, ils seront écrits en Python pour le logiciel de gestion d'exécution de job de calcul Ganga[51] présenté dans le chapitre 2.

```
//Définition du fichier statut
statusName = new CMetaFileName("status", id, ".stat");
hub = new CStatusHub( CLocalStatusProvider.GetInstance() );
hub.init( new CLargeSequenceSplitting( CRNGType.MT19937v32, 200000000L ) );
statusRef = new CStatusFile(statusName,
                            hub , new CStatusSerializerCLHEPMT19937v32()) ;

//Définition du fichier de lancement Ganga
scriptName = new CMetaFileName("job", id, ".py");
script = new CGangaScript(scriptName, distributor);
lcg = new CGangaLCG();
```

[51] http://ganga.web.cern.ch/ganga

```
script.setBackEnd(lcg);

//Définition des méta-fichiers d'entrée
inputMetaFiles = new CMetaFiles();
inputMetaFiles.add(statusRef);
inputMetaFiles.add(script);
```

Code 30 Définition des métafichiers d'entrée

Le Code 31 défini les fichiers de sortie des jobs. Dans notre cas, chaque job comprend un unique fichier de sortie contenant son estimation de π.

```
//Description des fichiers de sortie
output = new CMetaFileName("output",id,".out");
outputFiles = new CMetaFiles();
outputFiles.add(output);
```

Code 31 Définition des fichiers de sortie

Le Code 32 décrit la commande de lancement d'un job de simulation avec l'exécutable et ses arguments. Celle-ci est sous la forme : `pi fichier_statut fichier_de_sortie`.

```
//Description de la commande de lancement
launchingCmd = new CLaunchingCommand("pi");
launchingCmd.addArgument(statusName);
launchingCmd.addArgument(output);
```

Code 32 Définition de la commande de lancement

Le Code 33 présente les opérations de description et de distribution de la simulation. La simulation est distribuée en 50 jobs. Les fichiers sont écrits dans un répertoire du disque dur en local : `/home/reuillon/tmp/pi/jobs/`. En changeant de type de transfert, DistMe fournit la possibilité de transférer automatiquement les fichiers correspondant à une simulation distribuée sur une interface utilisateur ou sur le maître d'un cluster PBS via scp (Secure Copy).

```
simulation = new CSimulationDescription(inputFiles,
```

```
                                inputMetaFiles ,launchingCmd, outputFiles);

rm = new CTransfert(dir + "jobs/", new CLocal());
distributor.setTransfert(rm);

distributor.distribute(simulation, 50);
```

Code 33 Définition et distribution de la simulation

Après la distribution, pour chaque job sont générés deux fichiers : le fichier statut au format CLHEP et le fichier de description du job dans Ganga. Ce fichier de description est dans notre cas un script Python qui crée un job et décrit tout ce qui est nécessaire pour son exécution sur la grille de calcul EGEE avec Ganga. Le Code 34 présente un exemple de définition du 30ème job de notre simulation.

```
lcg=LCG();

j = Job(backend=lcg)
j.name = 'job1_30'
j.inputsandbox = ['status1_30.stat','job1_30.py','pi']
j.outputsandbox = ['output1_30.out']
j.application = Executable(exe='pi',args=
                                    ['status1_30.stat','output1_30.out'])

print "job1_30 instantiated."
```

Code 34 Contenu du fichier Pyhton job1_30.py de définition du job 30 pour le logiciel Ganga

III.5 L'exécution sur grille

Une fois les jobs générés, il faut les transférer sur une interface utilisateur de grille (User Interface) sur lequel Ganga est installé. Il suffit ensuite d'exécuter les scripts de définition des jobs dans Ganga et de lancer les jobs. La Figure 56 présente l'environnement Ganga gérant l'exécution des jobs sur la grille EGEE.

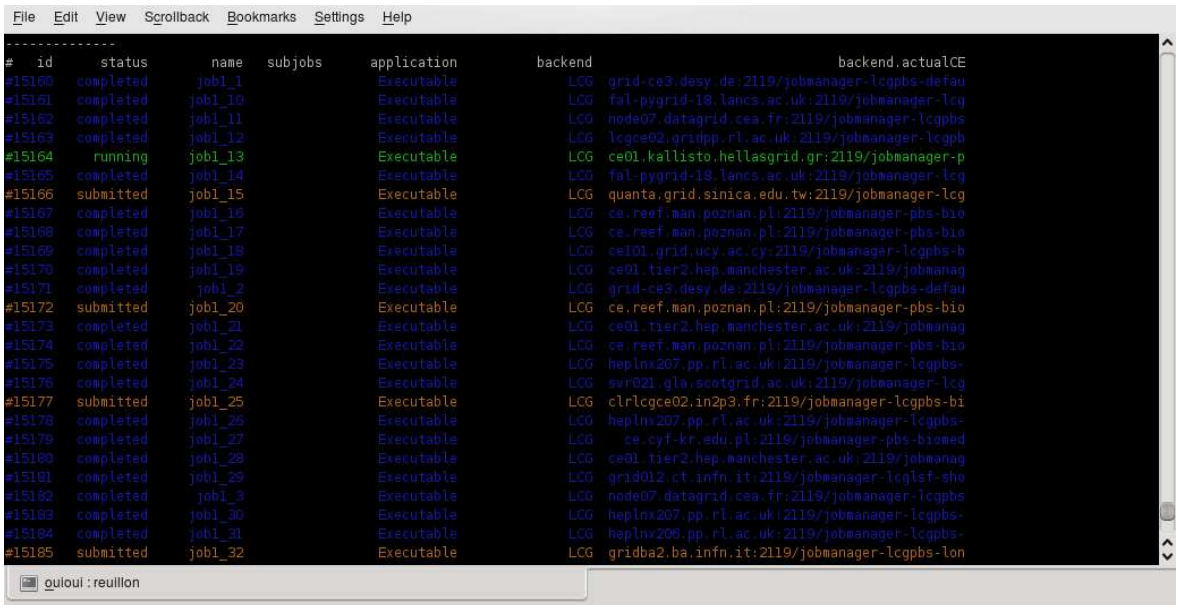


Figure 56 Capture d'écran du logiciel ganga de gestion d'exécution de jobs

Les jobs s'exécutent sur les ressources de l'organisation virtuelle « biomed » de la grille européenne EGEE. La Figure 57 présente la répartition de l'exécution des jobs sur les éléments de calcul selon le pays qui les hébergent. Par exemple, deux jobs ont été exécutés par un élément de calcul se trouvant sur l'île de Chypre.

Avec l'état actuel de la grille EGEE, les temps d'exécution des jobs sont très disparates. La Figure 58 présente graphiquement les temps de migration, d'attente et d'exécution des jobs sur la grille de calcul EGEE. Les temps de migration peuvent être importants du fait qu'ils incluent les tentatives d'exécution sur des éléments de calcul qui n'ont pas abouties. Dans la configuration que j'utilise, le nombre d'essais de soumission maximal pour un job est fixé à 3. Le job 50, par exemple, a certainement nécessité deux tentatives de soumissions avant d'être exécuté avec succès. Il a de ce fait attendu dans la file d'attente de trois fermes de calcul différentes. Dans le cas du job 48 il a été soumis rapidement à une ferme de calcul pour être exécuté. Malheureusement cette ferme de calcul était surchargée et non disponible pour exécuter le job dans un délai raisonnable.

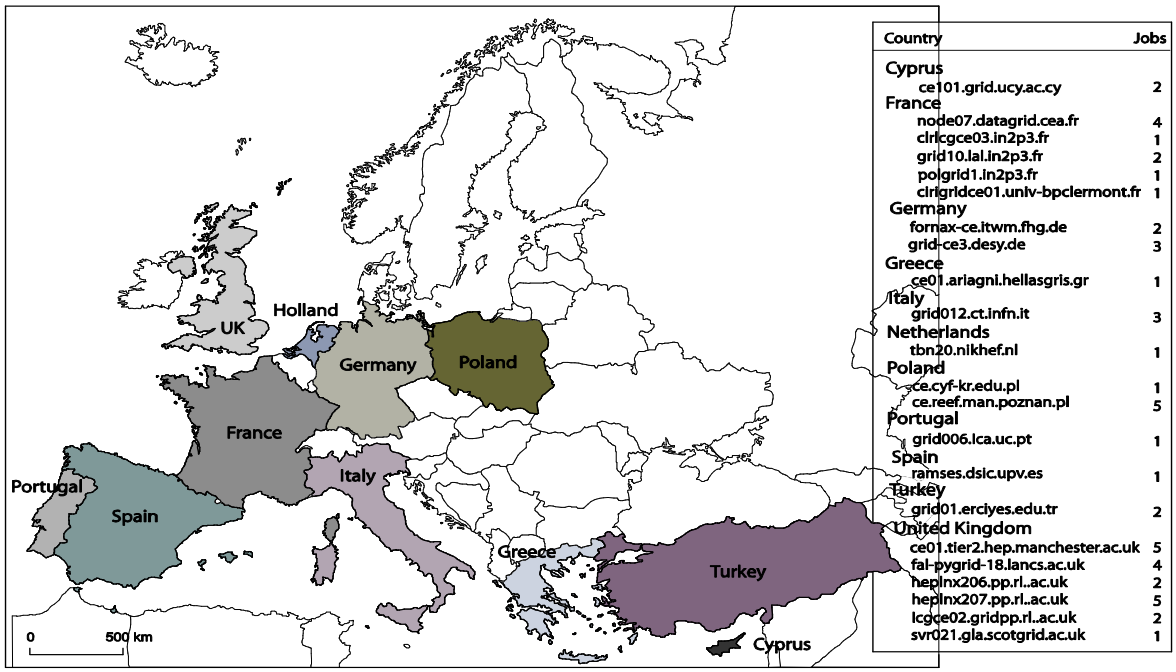


Figure 57 Répartition de l'exécution des jobs par pays

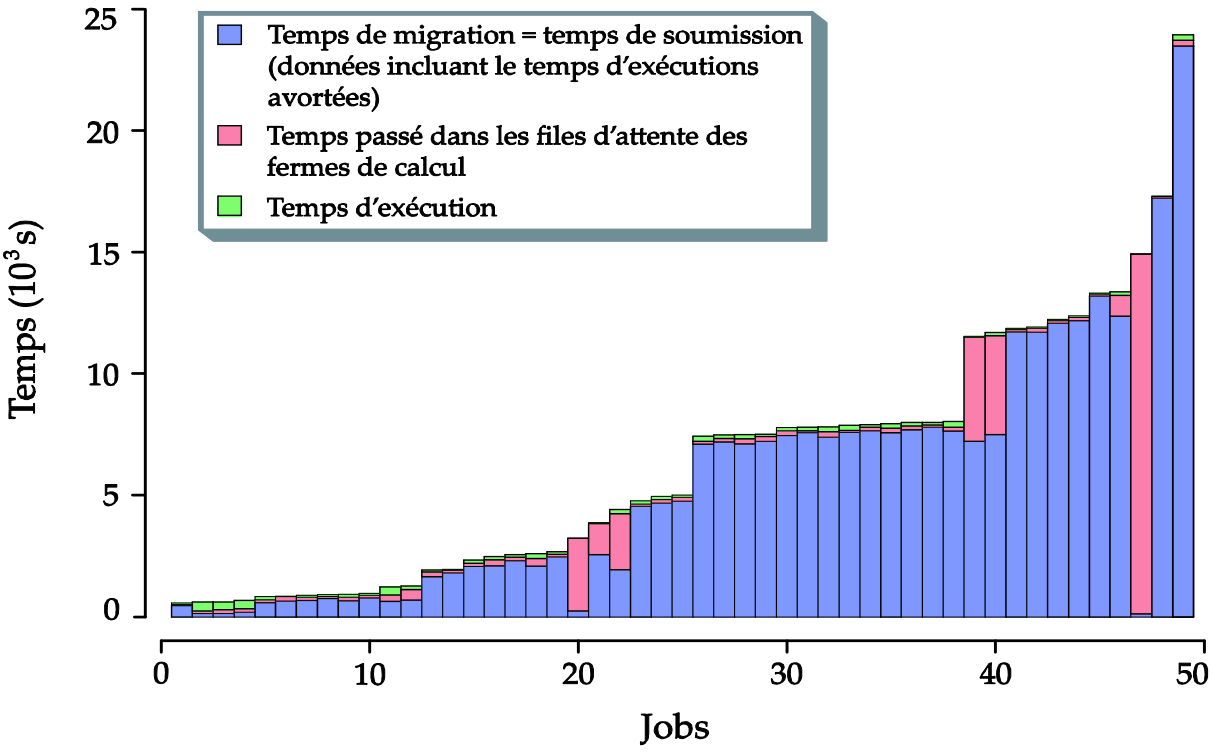


Figure 58 Temps d'exécution pour la simulation distribuée de calcul de π

Comme nous le voyons dans cet exemple, le temps total d'exécution d'un job sur la grille est égal au temps d'exécution ; plus le temps additionnel dû au mécanisme de migration et de soumission des jobs. Dans notre cas, le temps moyen additionnel dû à l'utilisation de la grille est de 128,77 minutes alors que le temps moyen d'exécution d'un

job est de 2 minutes 10 secondes. Il est bien sûr très mal venu de sous-traiter à une grille des petites simulations.

IV Distribution d'une simulation de physique médicale

L'exemple précédent permet d'illustrer la distribution d'une simulation stochastique avec DistMe. DistMe a été utilisé dans des cas concrets afin de réaliser la distribution d'une simulation stochastique de tomographie ainsi que pour la distribution d'une simulation environnementale. Cette partie présente ces deux travaux.

IV.1 La parallélisation de GATE

Dans la tomographie d'émission à simple photon simulée par ordinateur, le système matriciel qui transforme l'objet en 3 dimensions dans un ensemble de projection en 2 dimensions peut être calculé en utilisant des simulations de Monté Carlo. Cette approche est appelée F3DMC pour « Fully 3 Dimension Monte Carlo » (Lazaro, Breton et Buvat, Feasibility and value of fully 3D Monte Carlo reconstruction in Single Photon Emission Computed Tomography 2004). La simulation de Monté-Carlo permet de prendre en compte tous les phénomènes physiques qui ont lieu durant le processus d'acquisition, et ainsi d'obtenir une meilleure précision pour les images reconstruites par rapport aux autres méthodes de reconstruction (Lazaro, El Bitar, et al. 2005). Malheureusement, l'usage de F3DMC dans un contexte clinique semble impraticable du fait de la quantité de calcul qu'elle représente. Nous avons réalisé une reconstruction par la méthode F3DMC à partir d'acquisitions réalistes du fantôme présenté dans la Figure 59 et de la boîte à outils de simulation de Monté Carlo pour la tomographie par émission de positrons et pour la tomographie d'émission monophotonique GATE (Jan et al. 2004).

GATE a été conçu pour être flexible et très précis. Par conséquent, les simulations GATE sont très consommatrices en termes de ressources de calcul afin d'obtenir des résultats exploitables. Dans notre système simulé, le fantôme a été « voxélisé » en une matrice de taille 64x64x64. La précision de la reconstruction augmente avec le nombre de photons simulés par voxel (Qi et Huesman 2005). Dans notre cas, du fait de la taille de la matrice utilisée, les études préliminaires ont montré qu'il était nécessaire de calculer un grand nombre de trajectoires, de l'ordre de 74 milliards, afin d'obtenir une précision satisfaisante. L'augmentation de la définition du système, de sa complexité ou l'ajout d'une dimension temporelle représente chacun un facteur de croissance en

termes de puissance de calcul nécessaire. Seule l'utilisation de calculateurs massivement distribués type grille de calcul permet de réaliser de tels simulations (Maigne, et al. 2004).

Figure 59 Fantôme " Jaszczak"

Cette partie présente la parallélisation de l'application de la méthode de reconstruction d'image F3DMC en utilisant le simulateur GATE (pour plus d'information sur cette application voir (Breton et Buvat 2004) et (El Bitar, et al. 2006)). Le simulateur GATE a été initialement conçu pour utiliser l'algorithme de génération de nombres pseudo-aléatoires James Random (James 1990) implémenté en C++ dans la bibliothèque de classes pour la physique des hautes énergies (CLHEP). J'ai pu vérifier que celui-ci présentait de larges défauts au regard des tests statistiques actuels. En effet ce générateur ne passe que 36 tests sur 96 de la batterie de test « Crush » de TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007) (pour plus d'information sut TestU01 voir le chapitre 2 de ce manuscrit). Pour cette raison, j'ai remplacé l'utilisation de James Random dans GATE par celle du générateur Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) qui passe presque tous les tests de la même batterie « Crush ». Ce générateur est lui aussi implémenté dans CLHEP. Cela peut paraître surprenant, mais le changement du générateur pseudo-aléatoire de GATE n'est pas une opération aisée et nécessite la modification du code C++ et la recompilation d'une partie du logiciel GATE (plus de 80000 lignes de code).

Dans un deuxième temps, j'ai « parallélisé » le générateur de nombres pseudo-aléatoires Mersenne Twister 19937. Comme nous l'avons vu dans la première partie de ce chapitre, seules deux techniques de parallélisation sont utilisables pour ce

générateur : la technique des séquences indexées, et le découpage en séquence. J'ai choisi dans un premier temps de paralléliser le Mersenne Twister 19937 par découpage en séquence. Afin de garantir la contrainte de non-chevauchement des séquences de nombres pseudo-aléatoires consommées par les différents jobs ainsi qu'une durée des jobs compatible avec leurs exécutions sur grille, j'ai tout d'abord conçu un job de simulation d'environ 12 heures sur un nœud de calcul de puissance moyenne, pour l'organisation virtuelle « biomed » de la grille européenne EGEE. J'ai ensuite estimé sa consommation en termes de nombres pseudo-aléatoires à environ 12 milliards de tirages. J'ai enfin généré des fichiers statuts pour l'algorithme de génération Mersenne Twister 19937 espacés de 15 milliards de tirages chacun (nous avons utilisé une approche similaire dans (Maigne, et al. 2004)). La génération de 6000 statuts a nécessité 80 jours de calcul sur un processeur Intel Xéon 3 GHz. Ce calcul n'est pas aisément parallélisable et nécessite le déroulement du cycle de génération du « Mersenne Twister 19937 » pas à pas. Une fois les statuts générés ils ont été insérés dans la base de données de statuts de DistMe selon la méthode exposée dans le paragraphe III.3 de ce chapitre.

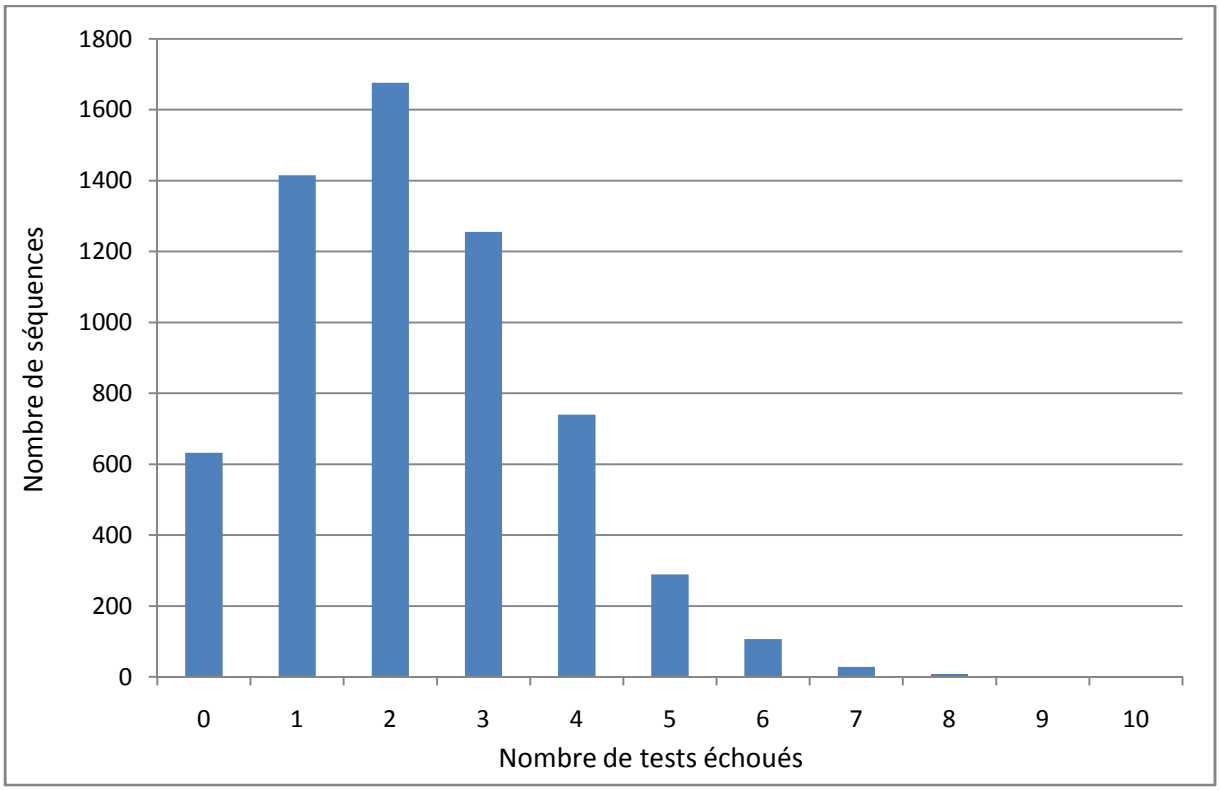


Figure 60 Nombre séquence échouant à n tests de la batterie Crush de TestU01

Nous avons ensuite testé individuellement chaque séquence de nombres pseudo-aléatoires générés à l'aide de la batterie « Crush » de TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007) pour en vérifier la qualité. La Figure 60 présente le résultat de l'exécution de la batterie de tests en totalisant le nombre de séquences échouant à *n* tests de la batterie. Seulement 2% des séquences échouent à plus de 5 tests de la batterie qui en comprend 96 au total. Les séquences utilisées pour la simulation sont donc de haute qualité au vue des standards actuels. Ce calcul a été effectué de manière distribuée. Il a nécessité 35 jours d'exécution sur une ferme de calcul de 14 bi-Xéon à 3 GHz à pleine charge. Ce même calcul exécuté en séquentiel sur un nœud de la ferme aurait nécessité 3 ans de calcul.

IV.2 Parallélisation de GATE avec DistMe

GATE permet de décrire le contexte de simulation sous forme de fichiers de macro. Ces fichiers comportent une suite de commandes qui permet entre autre de définir la géométrie de la simulation et les sources radioactives. Parmi les commandes disponibles dans GATE, la commande `/random/resetEngineFrom` permet d'initialiser le générateur de nombres pseudo-aléatoires avec un fichier statut. Le Code 35 est un extrait d'un fichier patron (template) utilisé par DistMe pour générer des macros GATE pour chaque job d'une simulation distribuée. Celui-ci permet d'initialiser le générateur de nombres pseudo-aléatoires avec un statut et définit différents noms de fichiers pour la sortie de chaque job de simulation.

```
/random/resetEngineFrom RNGSTATUS
/gate/output/root/setSaveRndmFlag 0
…
/gate/output/bin/setFileName BINOUTPUTFILE
…
/gate/output/root/source/setFileName ROOTOUTPUTFILE
…
```

Code 35 Extrait du fichier template permettant la génération de fichier de définition de simulation GATE pour la simulation distribuée

Ce fichier patron est utilisé par le script de distribution DistMe dont un extrait est présenté Code 36.

```
//Instanciation de l'identifiant pour les jobs
id = new CJobIdParameter(distributor);

//Création du méta-fichier statut
hub = new CStatusHub(CLocalStatusProvider.GetInstance());
hub.init(new CLargeSequenceSplitting(CRNGType.MT19937v32, 12000000000L));
statusName = new CMetaFileName("status",id,".stat");
statusRef = new CStatusFile(statusName,hub,
                                    new CStatusSerializerCLHEPMT19937v32());

//Création du la représentation du nom des fichiers de sortie
output = new CConcatenatedParameter();
output.add("output");
output.add(id);

//Définition du méta-fichier template pour la macro GATE
macroName = new CMetaFileName("macro",id,".mac");
macroRef = new CTemplateFile("/home/gate/Job.mac", macroName);
macroRef.addPattern("RNGSTATUS", statusName);
macroRef.addPattern("ROOTOUTPUTFILE",output);
macroRef.addPattern("BINOUTPUTFILE",output);

//Définition du méta-fichier JDL (Job Description Language)
script = new CJDLScript(distributor);
script.setRequirements("(Member(\"VO-biomed-GATE- … CPUTime>102))");
```

Code 36 Extrait du script de distribution DistMe d'une simulation GATE

A l'issue de l'exécution de ce script, chaque job de simulation comprendra :

- un statut pour le générateur Mersenne Twister 19937 parallélisé par découpage en séquence espacé d'au moins 12 milliards d'itérations de chacun des autres statuts,
- une macro GATE, initialisant le générateur aléatoire avec le statut, et générant des fichiers de sortie dont les noms sont distincts des autres jobs,
- un fichier JDL de description du job de calcul, permettant de lancer le job correspondant.

Un extrait du fichier JDL est présenté Code 37. Il contient la commande de lancement, son argument, les fichiers nécessaires pour l'exécution du job de simulation, ainsi que des informations pour que le job soit exécuté sur une unité d'exécution adaptée ou « Requirements ».

```
[
Executable ="Gate.sh";
Arguments = "macro0_10.mac";
InputSandbox ={ " macro0_10.mac", "status0_10.stat", … };
Requirements = (Member(\"VO-biomed-GATE- … CPUTime>102))");
…
]
```

Code 37 Extrait du fichier JDL généré par DistMe pour le lancement d'un job de simulation GATE

IV.3 L'Exécution de la simulation distribuée

Pour exécuter la simulation GATE distribuée, nous avions à notre disposition deux fermes de calcul des écoles d'ingénieur ISIMA et IFMA, composées de respectivement 14 et 28 bi-Xeon cadencés à 3 GHz et les ressources de l'organisation virtuelle « biomed » de la grille de calcul EGEE. L'utilisation de la grille EGEE nécessite l'installation préalable de GATE sur les éléments de calcul. Nous avons ainsi installé GATE sur 7 éléments de calcul regroupant 508 processeurs répartis dans 4 pays comme le montre la Figure 61.

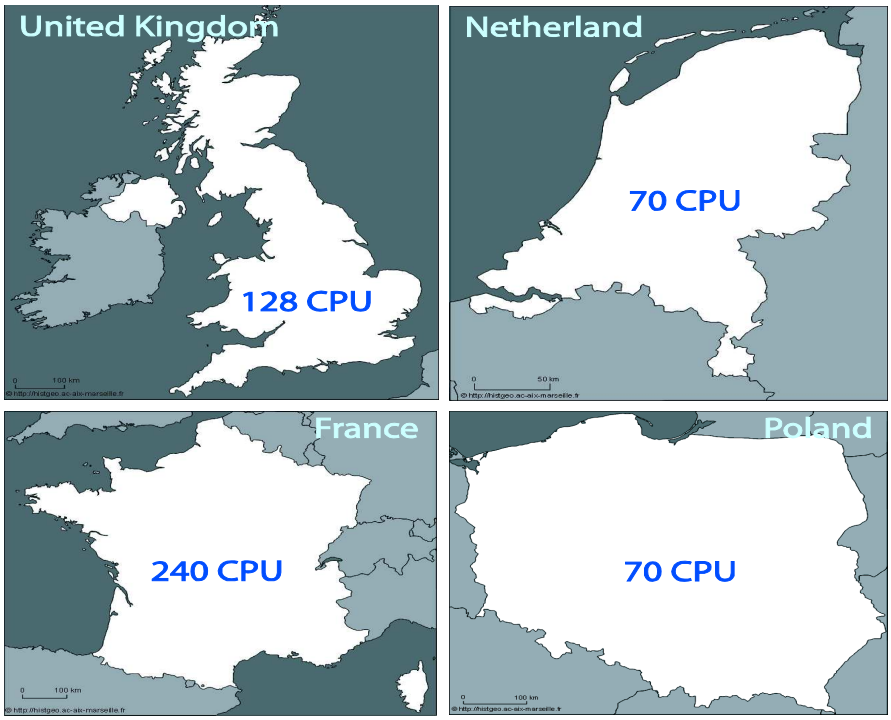


Figure 61 Répartition des processeurs disponibles pour l'exécution des jobs de simulation GATE

Les deux fermes de calcul ont exécuté 600 jobs et 2300 ont été exécutés sur la grille de calcul EGEE. Parmi les jobs exécutés sur la grille 1811 donnèrent des résultats utilisables (les autres jobs ont échouées). La Figure 62 expose la répartition des jobs de la simulation GATE parmi les éléments de calcul. La ferme de l'IFMA a exécuté 400 jobs, 200 se sont exécutés sur celle de l'ISIMA, 200 sur les nœuds de calcul de la grille en Pologne, 499 aux Pays-Bas, 922 en Angleterre et 190 en France. L'exécution de la simulation s'est effectuée en moins d'une semaine alors qu'elle aurait nécessité plus de trois ans de calcul sur une unité de calcul unique (Intel Xéon 3GHz). Le facteur de gain en temps de calcul pour cette exécution est donc de l'ordre de 170.

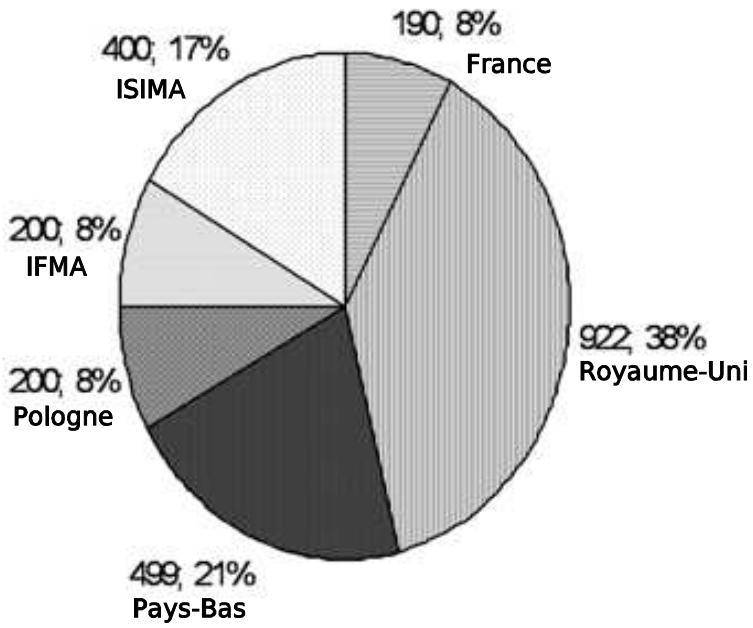


Figure 62: Répartition de l'exécution distribuée de la simulation GATE

La Figure 63 présente une étude de variance sur les résultats de la simulation. Sans entrer dans les détails du calcul de la variance, celle-ci fait apparaitre un état de convergence de la simulation après l'exécution de 2000 jobs. Nous avons alors rassemblé les fichiers de sortie des jobs, stockés sur des éléments de stockage (SE) de la grille de calcul et sur les systèmes de fichiers partagés des fermes. Dans notre cas chaque job de simulation génère deux fichiers binaires d'environ 10 méga-octets. Nous avons assemblés les fichiers afin de reconstruire le résultat tel qu'il aurait été calculé par une exécution séquentielle du simulateur. Cette opération a demandé 5 minutes de calcul pour regrouper les 30 giga-octets de données.

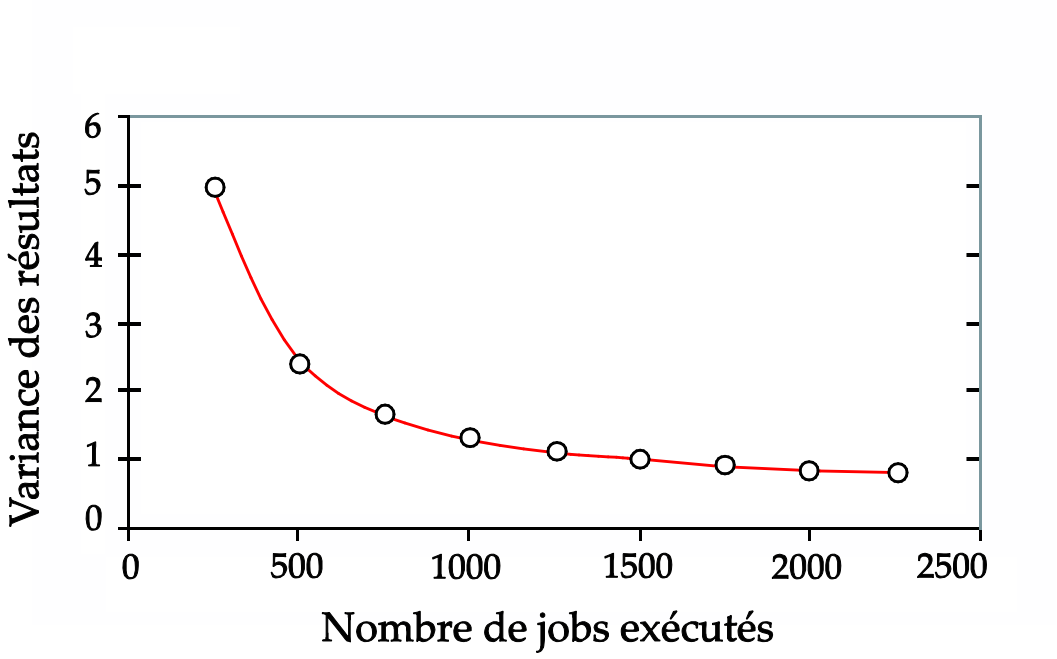


Figure 63 Etude de variance sur les résultats de sortie de la simulation distribuée en fonction du nombre de jobs exécutés

V Exploration de paramètres pour une simulation environnementale

La partie précédente présente la distribution d'une simulation GATE sur grille de calcul effectuée avec DistMe. Nous avons aussi utilisé DistMe pour l'exploration distribuée d'une simulation stochastique de croissance d'algues sur une ferme de calcul. Cette partie présente ce travail.

V.1 La simulation de croissance d'algues

DistMe permet également de réaliser des plans d'expériences élémentaires. A l'heure actuelle, le plan fractionnel complet (Kleijnen et Van Groenendaal, Simulation: A Statistical Perspective 1992) est implémenté et permet de bénéficier de la puissance de calcul d'environnements distribués pour explorer des modèles stochastiques. Cette partie présente l'exploration d'une application de simulation environnementale réalisée avec DistMe. Depuis le milieu des années 80, la *Caulerpa taxifolia* (Vahl) C. Agardh, une algue tropicale provenant d'aquariums a colonisé plusieurs milliers d'hectares sur la côte française et a été détectée à de nombreux endroits au nord de la Méditerranée, de la Croatie aux iles Baléares. Cette algue invasive a été également découverte en Turquie et sur la côte ouest des Etats-Unis (Jousson, et al. 2000) (Longepierre, et al. 2005). Une simple bouture suffit à induire la colonisation d'un lagon entier en moins de 10 ans. Afin de prévoir le développement de la *Caulerpa taxifolia* diverses techniques de

modélisation ont été testées et utilisées ces 10 dernières années (Hill, Coquillard et de Vaugelas, Discrete-Event Simulation of Alga Expansion 1997) (Hill, Coquillard et de Vaugelas, et al., An algorithmic Model for Invasive Species Application to Caulerpa taxifolia (Vahl) C, Agardh development in the North–Western Mediterranean Sea 1998) (Aussem et Hill 2000) (Hill, Coquillard et Aussem, et al. 2000), incluant la multi-modélisation d'un contrôle biologique utilisant des vers marins (Coquillard, et al. 2001). Parmi ces modèles, la simulation à événements discrets couplée avec un système d'information géographique est le meilleur outil pour simuler l'expansion spatiale de l'algue, principalement parce que cette approche autorise la modélisation d'interactions spatiales discrètes et stochastiques. Cependant, l'exécution de ce type de modèle requiert une grande puissance de calcul, en particulier lors de la simulation d'un grand nombre de simulations en vue d'obtenir des cartes de couverture spectrale en trois dimensions. Cette technique présentée à l'académie des sciences (Hill, Coquillard et de Vaugelas, Discrete-Event Simulation of Alga Expansion 1997) donne des cartes de probabilité de colonisation. Dans ce contexte, j'ai ainsi utilisé DistMe pour explorer différents scénarios et pour paralléliser chacun d'entre eux en utilisant une approche de type MRIP (Multiple Replications In Parallel).

V.2 Le modèle DistMe

Simct reste actuellement le simulateur le plus précis pour la prédiction de l'invasion de la *Caulerpa*. Cette partie présente un aperçu de la conception du script de distribution DistMe pour explorer plusieurs facteurs avec simct. Pour notre étude, le générateur d'origine de simct implémenté dans une macro C à été modifié pour utiliser l'algorithme de génération Mersenne Twister 19937 implémenté dans CLHEP supportant une initialisation à partir d'un fichier statut.

La Figure 64 montre un job de simulation simct avec l'approche boite noire proposée par DistMe. L'exécution d'un job simct nécessite :

- un fichier exécutable,
- un fichier contenant la carte du site géographique de simulation,
- un fichier décrivant le substrat et un fichier contenant des informations sur la position de l'algue au début de la simulation,

- un fichier de paramètres et un statut pour initialiser le générateur de nombres pseudo-aléatoires. Ces fichiers sont propres à chaque job.

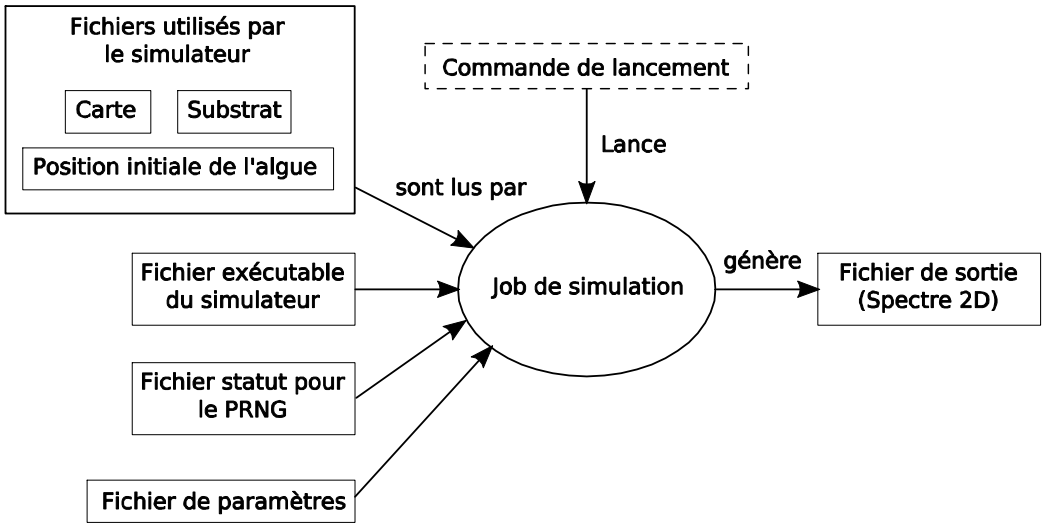


Figure 64 Modèle d'un job de simulation simct

Le Code 38 présente un extrait du script DistMe de distribution d'un plan d'expériences permettant d'explorer le modèle simct. Dans le premier paragraphe du script, on instancie un objet `CDistributor`. On construit ensuite un objet paramètre nommé « id » qui représente un identifiant unique pour chaque job. Dans le second paragraphe sont mentionnés les fichiers d'entrée nécessaires au bon fonctionnement du simulateur. Ils spécifient, la carte du site géographique de l'étude contenant les données bathymétriques, le substrat, et la position initiale des algues *Caulerpa*. Le troisième paragraphe décrit les fichiers statuts pour l'initialisation du générateur de nombres pseudo-aléatoires. Ces statuts sont sélectionnés dans la base de données et correspondent à une parallélisation du générateur par découpage en séquence. Les statuts sont espacés entre eux d'au moins 1 milliard de tirages, ce qui est très au dessus de la consommation en nombres aléatoires de chaque jobs de simulation. Chaque réplication de la simulation stochastique participant à l'exploration du modèle obtiendra ainsi une séquence de nombres pseudo-aléatoires indépendante de celles obtenues par toutes les autres.

```
/*- Paragraphe 1 -*/
distributor = new CDistributor();
id = new CJobIdParameter(distributor);
/*- Paragraphe 2 -*/
```

```
inputFiles = new CFiles();
inputFiles.add("/home/reuillon/tmp/simct/simct");
inputFiles.add("/home/reuillon/tmp/simct/mv.tca");
inputFiles.add("/home/reuillon/tmp/simct/mv.tsu");
inputFiles.add("/home/reuillon/tmp/simct/mv.tis");

/*- Paragraphe 3 -*/

statusName = new CMetaFileName("status", id, ".stat");
selection = new CLargeSequenceSplitting(CRNGType.MT19937v32, 1000000000L);
hub = new CStatusHub(CLocalStatusProvider.GetInstance());
hub.init(selection);
statusRef = new CStatusFile(statusName, hub ,
                                    new CStatusSerializerCLHEPMT19937v32());
```

Code 38 : Extrait du script DistMe pour la distribution de la simulation de croissance de la *Caulerpa taxifolia*

Le Code 39 présente la suite du script de distribution et d'exploration du modèle. Dans cette partie, je génère les fichiers d'entrée pour le simulateur. Chaque fichier d'entrée est généré à partir d'un fichier patron modifié de manière adéquate pour la classe `CTemplateFile`. Un fichier de paramètres contient le nom du fichier statuts pour initialiser le générateur de nombre pseudo-aléatoires ainsi qu'un ensemble de valeurs pour les paramètres d'entrée du modèle. Un extrait du patron est présenté à la fin du Code 39. Dans cette partie la génération du plan d'expérience est basée sur l'utilisation d'un ensemble de valeurs pour deux paramètres du modèle. Le premier paramètre est la croissance maximum d'un stolon par an.

Ce paramètre varie de 0,75 m à 1,5 m par pas de 0.05 m. 16 différentes valeurs sont ainsi générées pour ce paramètre. Le second paramètre représente la croissance en surface de l'algue. Ce facteur varie de 3 m à 6 m par pas de 0,2 m, ce qui correspond à 16 valeurs différentes. La combinaison des valeurs pour ces paramètres nous donnent un plan fractionnel complet comportant 256 expériences.

```
macroName = new CMetaFileName("mv", id, ".exp");
macroRef  = new CTemplateFile("/home/reuillon/tmp/simct/mv.exp",
                                                              macroName);
spectre = new CMetaFileName("mv", id, ".spe");
macroRef.addPattern("NOMSPECT", spectre);
```

```
longsto = new CExplorationDouble(0.75,1.5,0.05);     Ex : 0.75, 0.8, 0.85,
                                                     0.90, … 1.35, 1.40,
                                                     1.45, 1.50

croiss1 = new CExplorationDouble(3,6,0.2);

croiss2 = new CExplorationProduct();     Linear relations can be specified
between parameters

croiss2.addParam(croiss1);               For instance the biomass growth
croiss2.addParam(0.667);                 rate could be set at 2/3 of
                                    the surface growth rate

strategy = new CCompletePlan();          A complete plan strategy is chosen
strategy.explore(longsto);               and the parameters are registered
strategy.explore(croiss1);               for exploration.
strategy.explore(croiss2);

macroRef.addPattern("LONGSTO", longsto); Parameter values are replaced in
macroRef.addPattern("CROISS1",croiss1);  the parameter input file of the
macroRef.addPattern("CROISS2",croiss2);  simulator
```

Code 39: (Suite du) script DistMe utilisé pour paralléliser l'exploration du modèle d'invasion de la *Caulerpa taxifolia*

Afin de générer les fichiers d'expériences pour simct, DistMe utilise un fichier patron. Un extrait de ce fichier est présenté Code 40. Ce fichier comprend les trois facteurs explorés ainsi que des valeurs fixes pour les autres paramètres de simulation.

```
MODELE  cadre     Nom du site à étudier
FINICAU cadre     Nom du fichier des positions initiales de Caulerpe
SURFPX  0.0070    Surface occupée par un pixel sur ce site
SURFINI 0.01      Surface couverte par une pos. init. de Caulerpe (Deb Sim)
…
LGSTAN   LONGSTO   Rayon de distance parcourue par un stolon en 1 an (en m)
…
FACTSURF CROISS1   Taux de croissance annuel maxi pour 1 m2
FACTBIOM CROISS2   Taux de croissance annuel maxi pour 1 kg
…
```

Code 40 Extrait du fichier utilisé par DistMe pour générer les fichiers d'expériences pour simct

V.3 Exécution et résultats

D'un point de vu pratique, nous avons utilisé DistMe pour paralléliser l'exploration de paramètres sur la simulation de croissance d'algues. Nous avons généré des spectres en trois dimensions présentant la somme que des zones géographiques atteintes. Chacun des spectres est obtenu par le cumul de 1024 cartes en deux dimensions générées de manière distribuée. Elles correspondent chacune à l'exécution d'une réplication de la simulation. Le calcul de ces spectres représente 308 jours de calcul en séquentiel sur un processeur Intel Xéon à 3 GHz. Celui-ci a été réalisé en 11 jours sur un cluster de 14 bi-Xéons 3 GHz (soient 28 unités d'exécution).

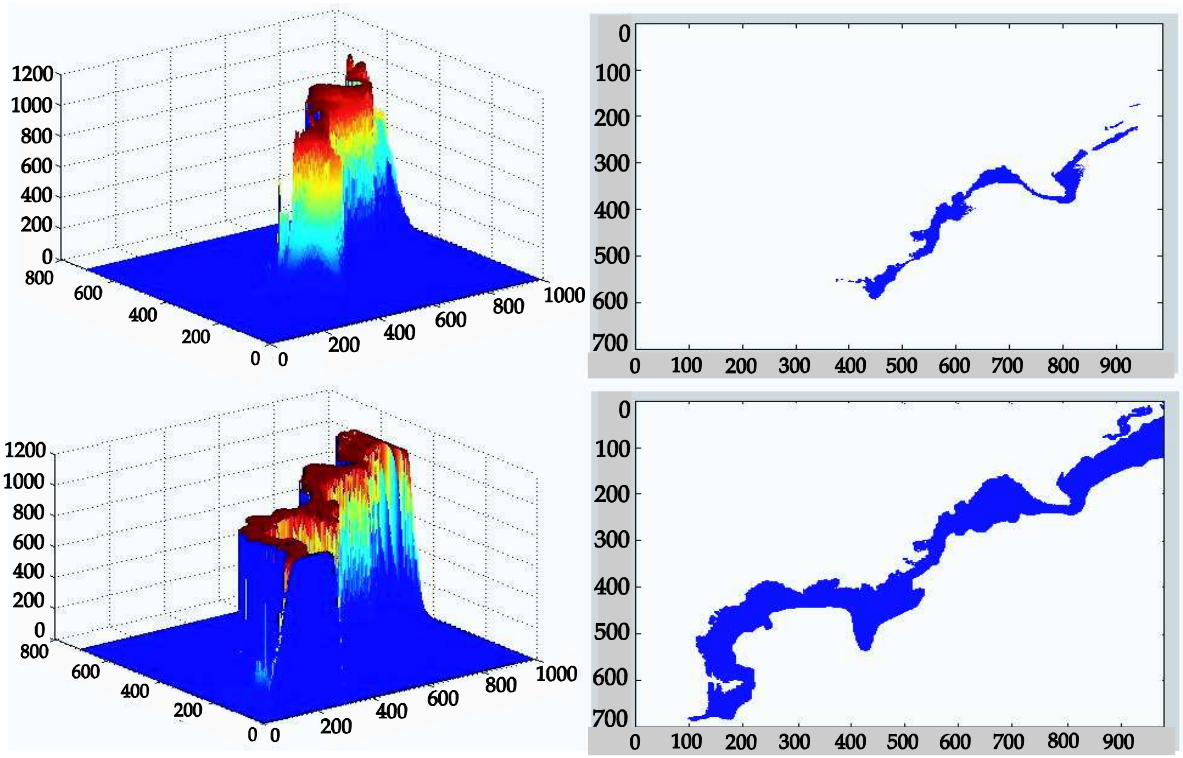


Figure 65: Spectres 3D de simulation de la répartition de la Caulerpa à gauche et carte des zone colonisées par l'algue en 2D à droite (1 pixel correspond approximativement à 360 m²)

La Figure 65 montre deux exemples parmi les résultats obtenus lors de la campagne d'exploration des paramètres. Les deux graphiques du haut présentent un scénario optimiste et ceux du bas un scénario pessimiste de la colonisation de l'algue pour la côte entre Menton et Villefranche-sur-Mer (12 années sont simulées). Les cartes en deux dimensions à gauche représentent les zones atteintes par l'algue. Le cumul de 1024 cartes, issues chacune des exécutions indépendantes du simulateur, permet d'obtenir des spectres en trois dimensions. Ces spectres sont présentés sur la gauche de la figure.

Les axes X et Y représentent la zone géographique discrétisée (environ 360 m^2 par pixel) et l'axe Z présente le nombre de réplication indépendantes de la simulation ou la colonisation sur un pixel (x,y) a été constatée.

VI Les caractéristique de DistRNG

DistMe permet la distribution de simulations stochastiques selon une approche MRIP ainsi qu'une exploration distribuée de plan d'expériences pour des simulateurs utilisant des bibliothèques de génération de nombres pseudo-aléatoires dont les générateurs peuvent être initialisés avec un statut. Comme évoqué auparavant, pour nous la notion de statut est centrale dans le processus de distribution des simulations stochastiques. Nous avons ainsi conçu un format de statut en XML permettant une description claire du contenu des statuts et étendant la notion de statut à une utilisation en environnements distribués. Nous avons, de plus, implémenté une bibliothèque appelée DistRNG, tirant le meilleur parti de cette nouvelle vision des statuts pour les générateurs de nombres pseudo-aléatoires. Cette partie décrit les fonctionnalités de DistRNG.

VI.1 Sélection du générateur de nombre pseudo-aléatoires au cours de l'exécution et l'utilisation des métadonnées

DistRNG est une bibliothèque libre en java conçue pour permettre le découplage entre la phase de conception d'un modèle de simulation stochastique et la phase de parallélisation de celui-ci en vu de son exécution sur un environnement distribué. DistRNG permet d'externaliser entièrement la tâche de distribution du modèle de la phase de conception. Cette bibliothèque est basée sur l'utilisation du concept de statuts génériques que nous avons proposé et utilise les métadonnées sur les séquences de nombres. DistRNG fournit ainsi les mécanismes nécessaires pour concevoir un modèle sans choisir ni paramétrer l'algorithme de génération de nombres pseudo-aléatoires. Ce choix est reporté à l'exécution au cours de laquelle l'algorithme de génération sera sélectionné, paramétré et initialisé à l'aide d'un statut. Cette fonctionnalité n'est à l'heure actuelle offerte dans aucune autre bibliothèque de génération de nombres pseudo-aléatoires.

Le Code 41 présente une utilisation de cette fonctionnalité dans un calcul d'intervalle de confiance pour une simulation stochastique de calcul de π avec différents générateurs de nombres pseudo-aléatoires. Dans cet exemple, des fichiers statuts génériques au

format XML (présentés dans le chapitre 3) sont placés dans un répertoire. Pour chaque fichier statut, un objet désérialiseur utilise la méta-information « désignation du générateur de nombres pseudo-aléatoires » du fichier statuts XML afin d'instancier l'algorithme de génération correspondant. Le générateur est ensuite initialisé en utilisant la partie « donnée » du fichier statut. L'objet de type `CRNG` ainsi initialisé permet d'une part, de générer des nombres pseudo-aléatoires, et d'autre part d'accéder aux méta-informations sur la séquence générée comme : la distance, la désignation du groupe et la désignation de la sous-série. Dans la suite du code, les méta-informations sont affichées. Enfin, un calcul de π est exécuté avec 25 réplications comprenant 10 millions de tirage de points pour chacune. A la fin de l'exécution la méta-information « distance » est affichée. Celle-ci est alors égale à la distance du statut initialement chargé augmentée du nombre effectif d'itérations de l'algorithme qui ont eut lieu pendant le calcul de π. Dans ce code, l'algorithme de génération n'est pas fixé, ce dernier est choisi à l'exécution en fonction des statuts présents dans un répertoire. De plus, les méta-informations sont consultables et tenues à jour. Ceci permet par exemple à l'expérimentateur de choisir et de changer le générateur de nombres pseudo-aléatoires utilisé par tous jobs d'une simulation dans un script de distribution DistMe. L'utilisation de générateurs différents pour des réplications d'une même simulation doit cependant être évitée en l'état des connaissances actuelles.

```
public static void main(String[] args) {
  File dir = new File("/home/reuillon/tmp/DistRNGTest");
  File statusDir = new File(dir,"status");

  CRNGSerializerXML serializer = new CRNGSerializerXML();
  CPi pi = new CPi();

  for(File status : statusDir.listFiles()) {
    try {
      CRNG<?> rng = serializer.loadFile(status);
      System.out.println(rng.getStatus().getGroup());
      System.out.println(rng.getStatus().getSubSerie());
      System.out.println(rng.getStatus().getDistance());
      pi.initRNG(rng);
      pi.intervalConf(25, 10000000, new PrintStream(
                                  new File(dir, status.getName()+ ".res")));
```

```
        System.out.println(rng.getStatus().getDistance());
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
```

Code 41 Exemple de choix de l'algorithme de génération lors de l'exécution

Les résultats de l'exécution du Code 41 sont présentés Figure 66 sous forme graphique. La courbe en bleu foncé au milieu de la figure représente la moyenne des valeurs de π après *n* réplications. L'aire en bleu clair représente l'intervalle de confiance. Lors de l'exécution de l'exemple, j'ai utilisé quatre statuts pour quatre algorithmes de génération de nombres pseudo-aléatoires : James Random (James 1990), un algorithme de type linéaire congruentiel, le Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) et le Mersenne Twister générique. D'après les indicateurs statistiques présentés pour ce cas d'étude, il semble que dans ce cas l'algorithme James Random et le Mersenne Twister générique présentent la meilleure vitesse de convergence.

Le nombre d'itérations de l'algorithme de génération comptabilisé par la méta-information `distance` varie en fonction de l'algorithme utilisé. Le générateur Mersenne Twister 19937 dans sa version 32 bits manipule des entiers sur 32 bits. Une itération du générateur correspond donc à la génération d'un entier. Dans certaines bibliothèques comme CLHEP, la génération d'un nombre double précision en utilisant cet algorithme utilise le tirage d'un seul entier. Afin de générer un nombre double précision aléatoire sur les 52 bits de sa mantisse, dans DistRNG le tirage d'un nombre double précision en utilisant l'algorithme Mersenne Twister 19937 nécessite la génération de deux entiers. La méta-information « distance » est ainsi incrémentée de deux à chaque tirage d'un nombre double précision avec le générateur Mersenne Twister 19937. Dans l'exemple précédent, 10 millions de points sont générés pour chacune des 25 réplications. Un point est un couple de nombres en double précision, il correspond donc à la génération de 4 entiers avec l'algorithme Mersenne Twister. Pour Mersenne Twister la distance finale est augmentée de 1 milliard par rapport à la distance initiale alors que pour l'algorithme

James Random, qui génère nativement des nombres en double précision, l'augmentation de distance n'est que de 500 millions de tirages.

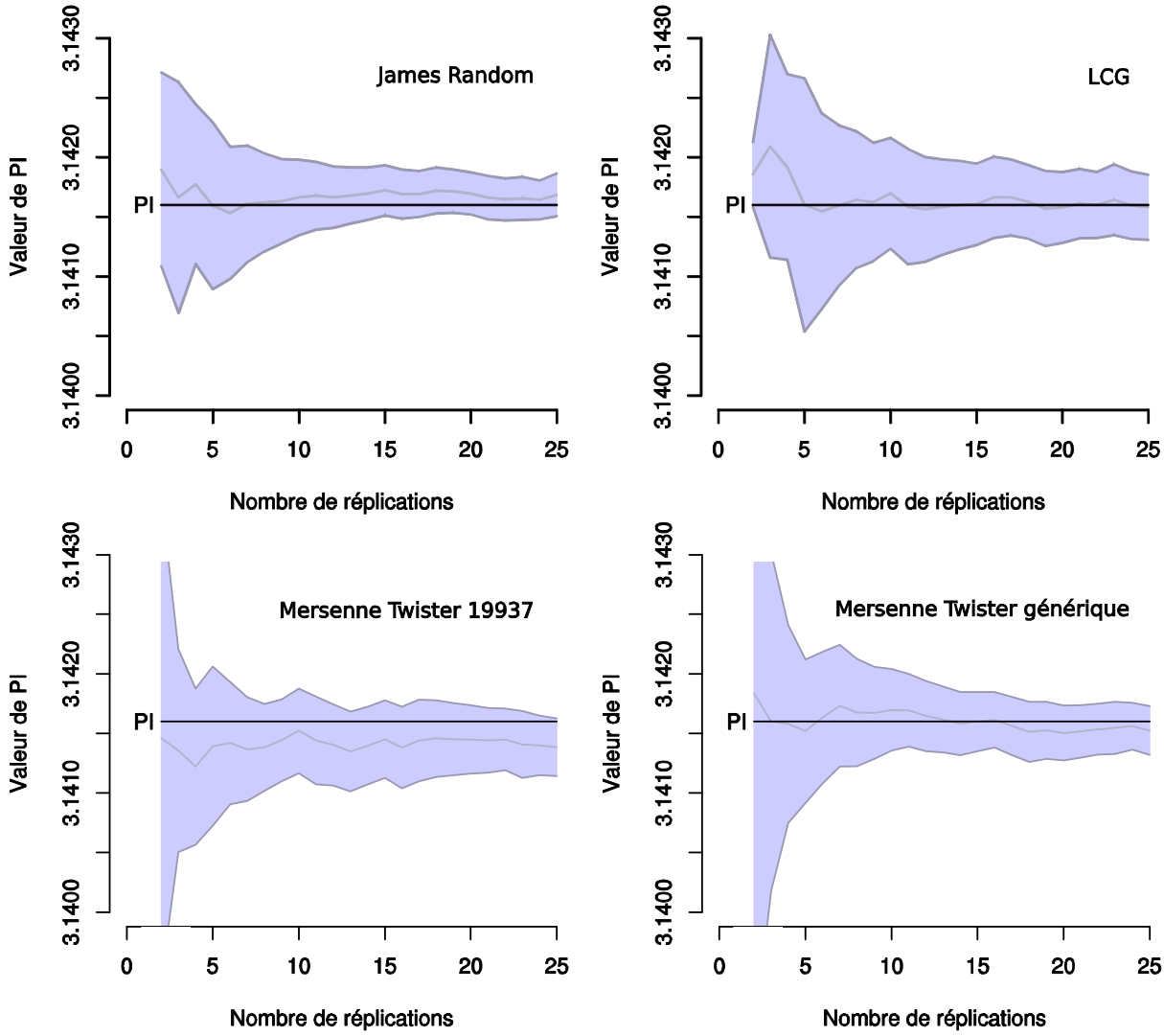


Figure 66 Calcul d'un intervalle de confiance lors de l'exécution d'un calcul de π par la méthode de Monte-Carlo avec quatre générateurs de nombres pseudo-aléatoires différents et 25 réplications

VI.2 Parallélisation des générateurs de nombres pseudo-aléatoires

VI.2.1 Le découpage en séquence

Toujours avec l'objectif de découpler les tâches de distribution d'une simulation stochastique du développement d'un modèle stochastique, DistRNG fournit une implémentation de techniques pour la parallélisation de générateurs de nombres pseudo-aléatoires en accord avec l'état de l'art.

La parallélisation d'un générateur de nombres pseudo-aléatoires par découpage en séquence permet l'utilisation lors de l'exécution distribuée des sous-séquences du cycle global de génération espacées d'un nombre connu de tirages. Le Code 42 permet de

générer 100 séquences parallèles pour le générateur de nombres pseudo-aléatoires Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997). Pour chaque séquence l'état du générateur est sauvegardé dans un statut au format XML. Le statut suivant sera espacé d'un 1 milliard de tirages. La classe `CSequenceSplitting` implémente la technique de découpage en séquence. Elle utilise la méthode `jump` de la classe `CRNG` pour dérouler le cycle du générateur. Pour certains générateurs, cette méthode peut être surchargée afin de coupler la technique de parallélisation par découpage en séquence avec une technique de division de cycles (Mascagni et Srinivasan, Parameterizing parallel multiplicative lagged-Fibonacci generators 2004).

```
public static void main(String[] args) {
  CStatusSerializerXML serializer = new CStatusSerializerXML();
  File dir = new File("/home/reuillon/tmp/MT");
  CMT19937v32 rng = new CMT19937v32();
  CSequenceSplitting<CMT19937v32> splitter =
                    new CSequenceSplitting<CMT19937v32>(rng, 10000000L);
  for(int i = 0; i < 100; i++) {
    try {
      CMT19937v32 splittedRng = splitter.generateRNG();
      serializer.saveFile(splittedRng.getStatus(),
                                        new File(dir, "MT" + i + ".xml"));
    } catch (CNoRNGAvailableException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Code 42 Parallélisation du générateur de nombres pseudo-aléatoires Mersenne Twister par découpage en séquence

VI.2.2 La paramétrisation du Mersenne Twister générique

DistRNG est la seule bibliothèque de classes en java à implémenter la parallélisation par paramétrisation pour l'algorithme Mersenne Twister (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000). Cet algorithme a été porté à partir de la version en langage C implémentée par M. Matsumoto vers une

version orientée objet en java. L'utilisation de la classe implémentant le générateur de paramètres est exposée dans le Code 43. L'algorithme prend en paramètre la longueur pour les mots (31 ou 32 bits) et la période p du générateur cible sous la forme d'un nombre premier de Mersenne : $2^p - 1$ (les choix possibles pour p sont : 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497). Ici ces paramètres se retrouvent sous la forme d'arguments du constructeur de la classe `CMTv32Parametrizer`. Pour créer des générateurs de type Mersenne Twister générant des séquences hautement indépendantes (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000) il est alors possible d'appeler la méthode `generateRNG`. Du fait des limitations de l'algorithme en C de Matsumoto, il est possible d'instancier au maximum 656536 générateurs paramétrés indépendants.

```
CMTv32Parametrizer param = new CMTv32Parametrizer(32, 521);
CMTv32 mt = param.generateRNG();
```

Code 43 Instanciation d'un paramétreur pour l'algorithme générique Mersenne Twister

VI.2.3 L'indexation de séquences

Grâce à la grille de calcul européenne EGEE, il est aujourd'hui possible d'exécuter de manière distribuée des calculs équivalents à plusieurs centaines d'années de calculs sur un seul processeur (Jacq, N. et al. 2007). Afin de réaliser de telles quantités de calcul, il est nécessaire d'exécuter plusieurs centaines de milliers de jobs. Dans le cadre de l'exécution d'une simulation de Monte-Carlo on doit alors disposer d'autant de statuts pour initialiser le générateur de nombres pseudo-aléatoires de chaque job.

Partant du constat de ce besoin bien réel et à la demande de la communauté des physiciens médicaux, j'ai généré 1 million de statuts pour le générateur Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) menant à la génération de séquences indépendantes en utilisant la méthode de l'indexation de séquences. Afin de minimiser les corrélations dans les statuts générés, j'ai indexé le générateur Mersenne Twister avec un générateur de qualité cryptographique basé sur la

fonction de hachage « SHA-1 » conçue par la NSA (National Security Agency) et implémenté dans la bibliothèque java GNUCrypto[52].

```
public static void main(String[] args) {
  MDGenerator rnd = new MDGenerator();
  rnd.init(new HashMap());
  CGnuCryptoRNG grng = new CGnuCryptoRNG(rnd);

  CMT19937v32 rng = new CMT19937v32();
  rng.getStatus().setDistance(0);
  rng.getStatus().setSubSerie("");
  rng.getStatus().setGroup("IndexSequence_SHA1_Joe");

  CIndexSequence is = new CIndexSequence(grng, rng);

  CRNGSerializerXML serial = new CRNGSerializerXML();
  CStatusSerializerCLHEPMT19937v32 serialCLHEP =
                                  new CStatusSerializerCLHEPMT19937v32();

  File dir = new File("/home/reuillon/tmp/Joe/statuses");
  File dirCLHEP = new File("/home/reuillon/tmp/Joe/statusesCLHEP");

  for(int i = 0; i < 1000000; i++) {
    try {
      CRNG isrng = is.generateRNG();
      String name = "MT" + isrng.getStatus().getSubSerie();
      serial.saveFile(isrng, new File(dir, name + ".xml"));
      serialCLHEP.saveFile(isrng.getStatus(),
                                  new File(dirCLHEP, name + ".stat" ) );
    } catch (CNoRNGAvailableException e) {
      e.printStackTrace();
      break;
    } catch (IOException e) {
      e.printStackTrace();
      break;
    }
  }
}
```

[52] http://www.gnu.org/software/gnu-crypto/

Code 44 Génération de 1 million de statuts en XML et au format CLHEP pour l'algorithme Mersenne Twister 19937 par indexation de séquence à l'aide d'un générateur de qualité cryptographique

Le Code 44 permet de réaliser ce travail. Tout d'abord, le générateur cryptographique SHA-1 de la bibliothèque GNUCrypto est instancié et encapsulé dans un objet DistRNG. Ensuite, on initialise un générateur patron qui sera cloné puis modifié afin d'obtenir un nouveau générateur par indexation de séquences. Enfin, les 1 million de statuts sont générés et sauvegardés en deux formats différents : XML et CLHEP. Après 30 heures de calcul sur un Pentium 4 à 2,8 GHz, j'ai obtenu les 1 million de statuts occupant 30 Go d'espace disque.

VI.3 Vitesse de génération et choix d'implémentations

Une des caractéristiques importante pour un bon générateur de nombres pseudo-aléatoires est sa vitesse de génération. Le Tableau 8 présente une comparaison entre DistRNG, la bibliothèque java SSJ (L'Ecuyer et Buist, Simulation in Java with SSJ 2005) et la bibliothèque C++ CLHEP (Lönnblad 1994). La vitesse de génération de 1 milliard de nombres pseudo-aléatoires est comparée pour les deux algorithmes sur un AMD Athlon 3000+ (à 1,8 GHz) :

- Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) sur 32 bits,
- James Random (James 1990).

Tableau 8 Comparaison des vitesses de tirage pour 1 milliard de nombres entre différentes implémentations pour 3 algorithmes de génération de nombres pseudo-aléatoires

Générateur	CLHEP	SSJ	DistRNG sans mise à jour de la distance	DistRNG avec mise à jour de la distance
Mersenne Twister 32 bits	45s (1 itération par double)	51s (1 itération par double)	79s (1 itération par double) 182s (2 itérations par double)	109s (1 itération par double) 242s (2 itérations par double)
James Random	40s (un entier tiré par double)	NA	79s	106s

Comme le montre la comparaison de la vitesse de tirage des générateurs implémentés en java dans DistRNG et dans SSJ avec celle de leurs implémentations en C++ dans CLHEP : le choix de java comme langage d'implémentation ne permet pas d'obtenir une vitesse de génération optimale.

Au lancement du projet DistRNG, une étude de faisabilité a été menée pour l'implémentation de DistRNG en C++. Le C++ ne comprend pas d'API standard pour l'introspection, de ce fait, les bibliothèques de sérialisation en C++ comme Boost[53] ne permettent pas la flexibilité d'une bibliothèque java comme XStream[54]. L'introspection avec Boost est intrusive par rapport au code métier et nécessite l'implémentation d'une portion code spécifique dans chaque classe sérialisée.

L'utilisation de java permet de bénéficier non seulement d'une grande flexibilité dans l'utilisation de DistRNG, mais aussi de simplifier son extensibilité. Par exemple, lors de l'intégration d'un nouvel algorithme de génération de nombres pseudo-aléatoires dans l'architecture de DistRNG un format pour les fichiers statuts XML est disponible sans aucun travail d'implémentation de la part du développeur. De plus, le nouveau générateur est automatiquement utilisable dans DistMe pour la parallélisation de simulations stochastiques.

Le Tableau 8 montre de plus, que pour l'algorithme Mersenne Twister, l'implémentation de DistRNG est 50% plus lente que celle de SSJ (L'Ecuyer et Buist,

[53] http://www.boost.org/
[54] http://xstream.codehaus.org/

Simulation in Java with SSJ 2005). En effet plusieurs particularités de DistRNG expliquent cet écart :

- le statut est un objet séparé de l'objet générateur de nombres pseudo-aléatoires l'accès à cet objet prend un peu de temps,
- un test supplémentaire est effectué à chaque itération pour savoir si la distance doit être mise à jour ou pas,
- le formatage des nombres est délégué à des objets spécialisés.

La bibliothèque DistRNG en est actuellement à ses premières versions et un effort d'optimisation pourra très certainement réduire cet écart. La mise à jour de la méta-information distance, lors de l'exécution d'une simulation, ralentit inéluctablement la génération des nombres pseudo-aléatoires en ajoutant le temps d'une incrémentation sur un entier long à chaque itération du générateur de nombres pseudo-aléatoires. Elle peut être désactivée, cependant DistRNG est la seule bibliothèque qui permette d'éviter à coup sûr la corrélation par recouvrement de séquences en vérifiant qu'une distance maximum n'est jamais dépassée.

VII Test en parallèle de 65536 séquences

Nous avons vu dans la partie précédente que DistRNG implémente l'algorithme de génération parallèle de nombres pseudo-aléatoires Mersenne Twister générique (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000). Cet algorithme permet de générer des séquences de nombres pseudo-aléatoires hautement indépendantes. Le générateur de nombres pseudo-aléatoires Mersenne Twister générique est utilisable conjointement avec un algorithme de création dynamique (ou de paramétrisation) (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000). L'algorithme de paramétrisation, dans sa version actuelle, permet la génération de 65636 séquences indépendantes. Les deux algorithmes sont implémentés dans DistRNG et nous les avons mis en œuvre afin de tester les 65536 séquences indépendantes de nombres générées par l'algorithme Mersenne Twister générique sur la grille de calcul EGEE.

VII.1 Implémentation de l'application de test et génération des statuts

Pour tester les séquences de nombre pseudo-aléatoires générées par le Mersenne Twister générique, nous avons utilisé TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007), la bibliothèque de test la plus rigoureuse pour les générateurs de nombres pseudo-aléatoires à l'heure actuelle. Cette bibliothèque est présentée dans le chapitre 2 de ce manuscrit.

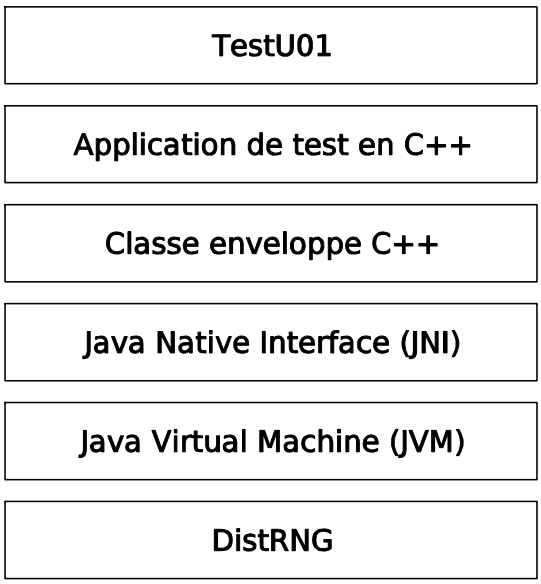


Figure 67 Architecture de l'application de test

DistRNG est implémenté en java alors que test TestU01 est implémenté en C. Afin de faire communiquer ces deux bibliothèques nous avons conçu l'application dont l'architecture élémentaire est présentée Figure 67. Au cœur de l'application, une classe enveloppe (wrapper) en C++ s'appuie sur l'interface de programmation JNI (Java Native Interface) pour construire des objets de DistRNG et appeler leurs méthodes. Celle-ci permet d'utiliser efficacement la bibliothèque DistRNG dans des programmes C++ grâce à des mécanismes de mise en mémoire tampon. L'application de test est ainsi programmée en C++. Elle utilise d'une part la classe enveloppe et d'autre part la libraire C TestU01.

Une fois l'application implémentée, nous avons généré 65636 statuts permettant d'initialiser des Mersenne Twister d'entiers de 32 bits de période 2^{521}-1. La génération des statuts avec la version en java de l'algorithme de création dynamique implémentée dans DistRNG a nécessité environ 18 jours de calcul sur un processeur Intel Pentium 4 à 3 GHz.

VII.2 Exécution

Une fois l'application de test mise sur pied, nous l'avons installée sur une quarantaine d'éléments de calcul appartenant à l'organisation virtuelle « biomed » de la grille de calcul EGEE. Puis nous avons exécuté le test des 65636 séquences.

Les jobs de test ont été générés grâce à l'application de distribution des simulations stochastiques DistMe. Pour ce faire, nous avons inséré les 65536 statuts générés dans la base de données de DistMe. Puis, nous avons généré un job par test. Les jobs ont été exécutés à l'aide du gestionnaire d'exécution ganga (présenté dans le chapitre 2 de ce manuscrit).

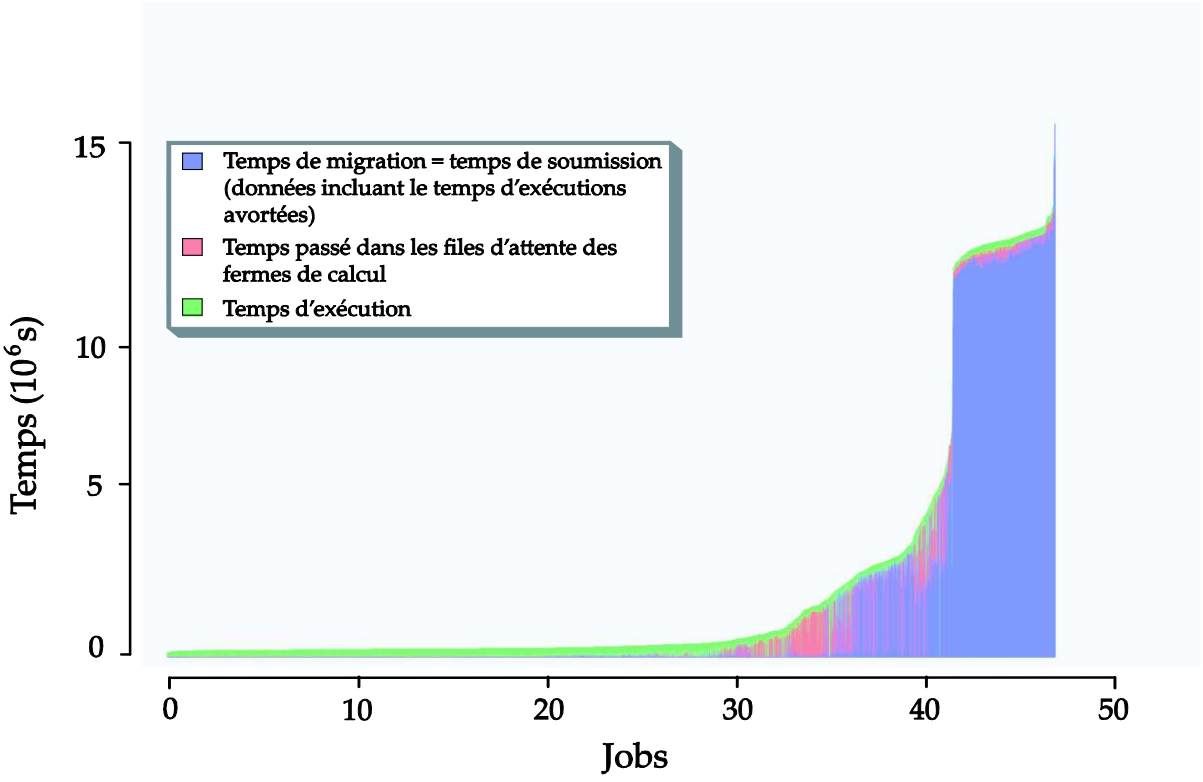


Figure 68 Représentation graphique des temps de migration, d'attente et d'exécution des jobs de test des séquences

Il a fallu un mois afin d'obtenir une exécution avec succès pour chacun des 65536 jobs. Au terme de l'exécution, nous avons pu récupérer 23302 fichiers de résumé d'exécution. Les autres étaient indisponibles au moment de la requête. La synthèse du contenu de ces fichiers est présentée dans la Figure 68. En bleu, figurent les temps de migration entre les différents éléments de grille. Ce dernier atteint un maximum à 13 jours et 22 heures. En rouge et en vert figurent respectivement les temps d'attente dans la queue de l'élément de calcul et le temps d'exécution. Le temps moyen d'exécution d'un

job de test pour une séquence est de 6h30, le temps moyen d'attente est de 5h45, le temps moyen de migration de 45h50. Même si le temps de migration moyen et très conséquent par rapport au temps d'exécution, le parallélisme massif des jobs m'a permis d'exécuter tous les tests, ce qui correspond à 49 années de calcul pour un seul processeur, en moins d'un mois.

VII.2.1 Résultats scientifiques

Les 2,25 billards de nombres générés par l'algorithme Mersenne Twister ont permis de tester la qualité statistique intrinsèque de chacune des séquences. Parmi les 65636 séquences testées aucune d'entre elle n'a échoué à plus de huit tests de la batterie « Crush » de TestU01.

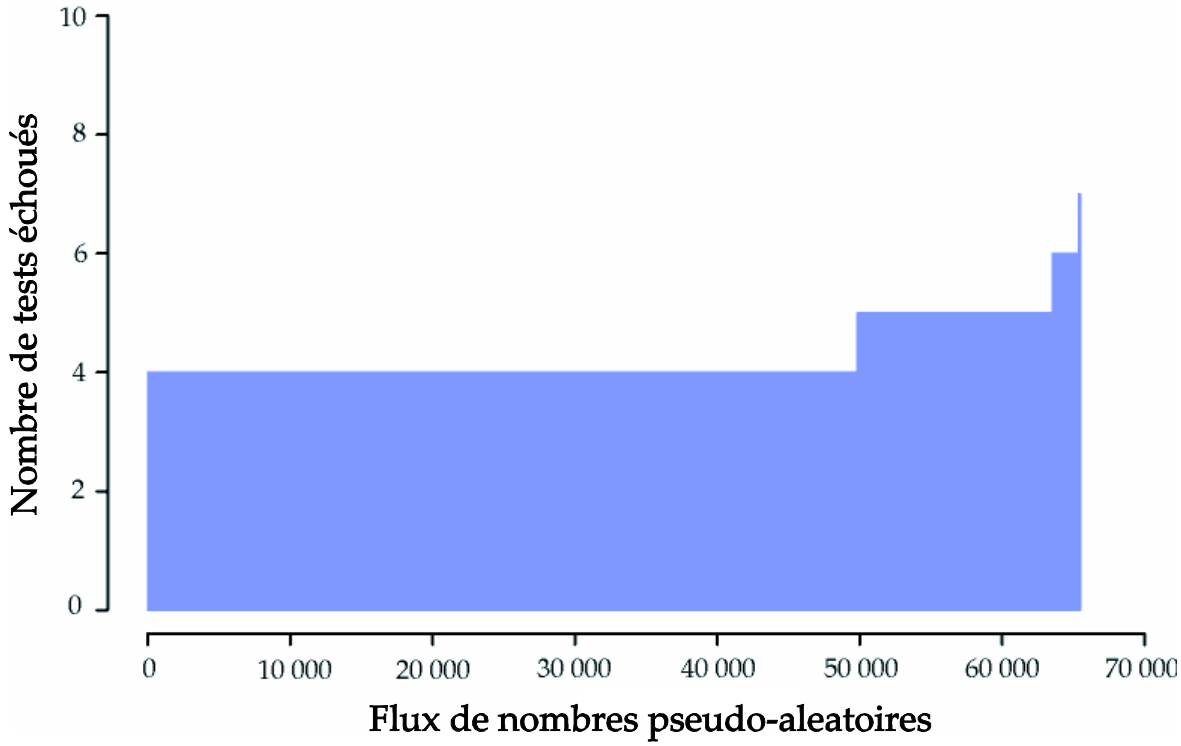


Figure 69 Représentation graphique du nombre de tests échoués par chaque séquence testée

Comme le montre la Figure 69, la majorité des séquences a échoué à 4 tests seulement. La batterie « Crush » de TestU01 ne met pas en évidence des défauts significatifs dans les séquences générées. Ces séquences sont donc au regard des standards actuels utilisables pour des simulations de Monte-Carlo.

La Figure 70 présente de manière graphique les nombres de séquences échouant à un test donné de la batterie « Crush » de la bibliothèque TestU01. Parmi les 96 tests, 4 ont échoué systématiquement pour toutes les séquences. Ces tests sont les tests

numéros 60, 61, 70 et 71 de la batterie de test. Les tests 70 et 71 sont des tests de complexité linéaire sur la séquence de bits générée. Ce défaut est connu a priori pour les générateurs de type Mersenne Twister. Les tests 60 et 61 correspondent à une implémentation du test « Matrix Rank » de la bibliothèque de test DieHard[55] avec deux jeux de paramètres différents. Il semble ainsi que dans notre cas, compte tenu des paramètres utilisés pour l'algorithme de paramétrisation, les corrélations internes du Mersenne Twister générique le rendent inutilisable dans certain cas de simulations.

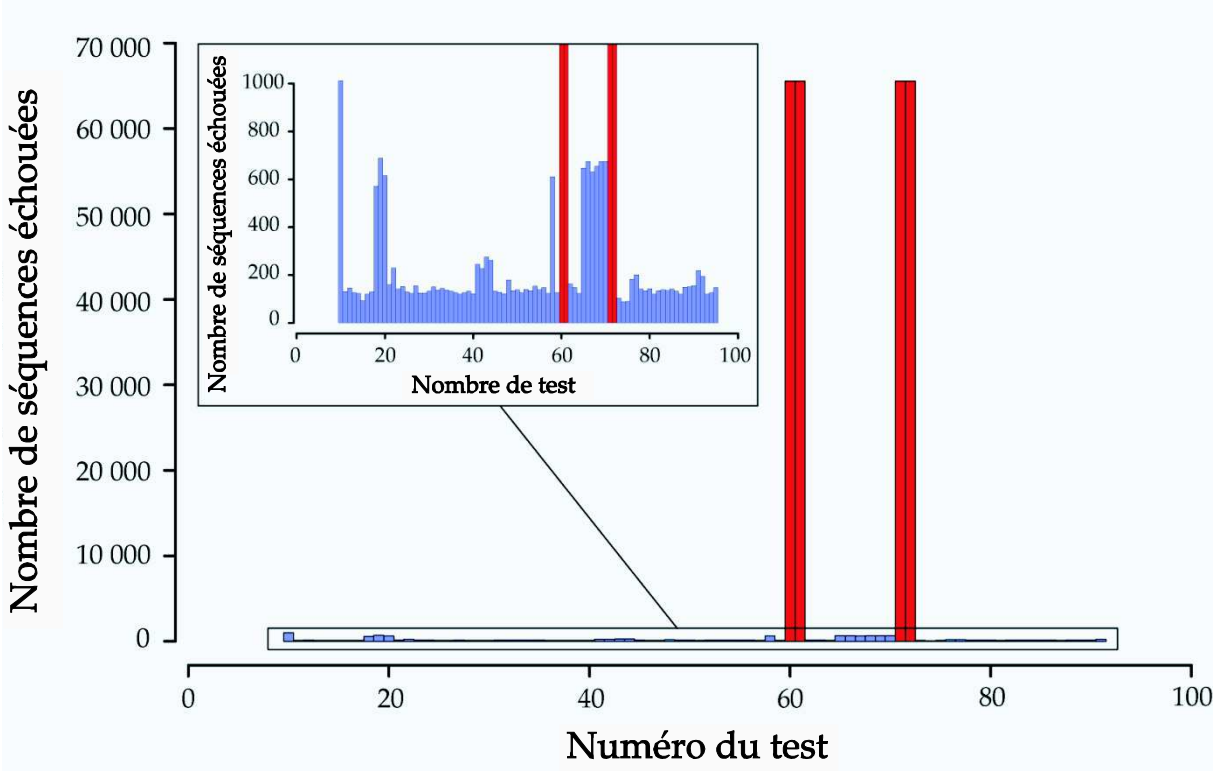


Figure 70 Représentation graphique du nombre de séquence échouant à un test donné de la batterie Crush de TestU01

Ce travail a servi à tester la qualité intrinsèque des séquences de nombres pseudo-aléatoires générées par l'algorithme Mersenne Twister paramètrique. Il a de plus permis d'utiliser pour la première fois la récente bibliothèque DistRNG pour la production de résultats scientifiques. Enfin, il a permis de valider l'implémentation en java dans DistRNG du générateur testé et de son algorithme de création dynamique.

[55] http://stat.fsu.edu/pub/diehard/

VIII Conclusion

Ce chapitre a présenté tout d'abord deux études qui ont motivé la réalisation des outils DistMe et DistRNG. Il expose ensuite des réalisations basées sur l'utilisation de ces deux logiciels.

Pour DistMe :

1. L'utilisation de DistMe dans des cas pratiques met en avant la capacité inégalée de ce logiciel à automatiser de manière rigoureuse la parallélisation de simulations stochastiques en produisant des dizaines de milliers de jobs prêt à être exécutés sur un environnement distribué.
2. La distribution du simulateur GATE sur deux fermes de calcul et sur la grille EGEE montre la polyvalence de DistMe par rapport aux environnements de calcul basés sur le paradigme du sac de travail.
3. La parallélisation d'application implémentée dans des langages différents (C, C++ et Java) et utilisant différentes bibliothèques de génération de nombres pseudo-aléatoires (CLHEP pour GATE et simct, DistRNG pour le test de 65636 séquences du Mersenne Twister générique) montre l'intérêt de l'approche boite noire utilisée par DistMe pour la distribution de simulateurs stochastiques existants, programmés dans différents langages et utilisant diverses bibliothèques de génération de nombres pseudo-aléatoires.
4. Les utilisations de DistMe pour, d'une part, une distribution du simulateur GATE basée sur un découpage en séquence du générateur Mersenne Twister 19937 et pour, d'autre part, la distribution du test des 65636 séquences indépendantes générées par paramétrisation du Mersenne Twister générique montre l'efficacité de l'utilisation des métadonnées des statuts génériques dans DistMe pour des parallélisations de simulations stochastiques en exploitant différents générateurs parallélisés selon diverses techniques.
5. L'exécution de 49 ans de calcul par un seul expérimentateur pour le test des 65536 séquences du générateur Mersenne Twister générique montre qu'associé à Ganga, DistMe est un outil approprié pour l'exploitation à grande échelle de la puissance des grilles de calcul.

Ce chapitre illustre aussi les caractéristiques exceptionnelles de DistRNG, le deuxième logiciel de la suite Dist. En effet, DistRNG propose les fondations pour la conception d'une bibliothèque de génération de nombres pseudo-aléatoires et présente des avantages significatifs par rapport aux autres bibliothèques de génération de nombres pseudo-aléatoires parallèles disponibles à ce jour.

Pour DistRNG :

1. L'exécution sur grille de calcul du test des 65536 séquences parallèles du Mersenne Twister générique illustre la possibilité d'utiliser DistRNG pour la distribution de simulations stochastiques à grande échelle qui permet l'exploitation de la puissance d'environnements largement distribués comme les grilles de calcul,
2. La génération pour la communauté des physiciens des particules d'un million de statuts issus de la parallélisation de l'algorithme Mersenne Twister 19937 en version 32 bits par indexation de séquences avec des générateurs de qualité cryptographique, montre que DistRNG correspond à un besoin réel de disposer d'implémentation de méthodes de parallélisation rigoureuses. La disponibilité de plusieurs méthodes de parallélisation dont la paramétrisation par création dynamique de Mersenne Twister répond à ce besoin.
3. L'exemple joué de l'étude de convergence de la simulation de Monte-Carlo de calcul de π en fonction de l'algorithme de génération de nombres pseudo-aléatoires utilisé, montre la faculté de déléguer le choix et l'initialisation de l'algorithme à l'exécution. Cette fonctionnalité est exceptionnelle et facilite l'analyse de sensibilité d'une simulation stochastique distribuée ou non à la qualité d'un générateur de nombres pseudo-aléatoires parallèle ou non.

Bien que DistRNG et DistMe soient deux outils encore en développement, ce chapitre montre qu'ils répondent mieux que tous les autres outils actuellement disponibles au problème de la distribution de simulations stochastiques à grande échelle. DistRNG et DistMe représentent ainsi des contributions significatives au domaine de la parallélisation des simulations stochastiques.

Conclusion générale

L'avènement des processeurs multi-cores, la disponibilité de fermes de calcul fédérant la puissance d'unités d'exécution de plus en plus nombreuses, la possibilité d'utiliser des environnements largement distribués extensifs comme les grilles de calcul et les plateformes de calcul pair à pair permettent aujourd'hui d'exécuter des simulations stochastiques correspondant à de très importants volumes de calcul. L'achèvement de ce type de calcul massif prendrait plusieurs dizaines d'années sur un seul calculateur séquentiel. Si les simulations stochastiques qui demandent l'exécution d'un nombre important de réplications indépendantes sont par conception naturellement parallèles, les mauvaises pratiques constatées dans certains domaines de recherche ainsi que les outils proposant l'utilisation de générateurs obsolètes et avec des défaillances facilement mises en évidence représentent un risque bien réel de produire et de publier des résultats erronés. Ce constat, évoqué dans de nombreuses publications (Pawlikowski, Jeong et Lee, On Credibility of Simulation Studies of Telecommunication Networks 2002), (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002), (Pawlikowski, Do Not Trust All Simulation Studies of Telecommunication Networks 2003), (Entacher et Hechenleitner, Pitfalls when using parallel streams in OMNET++ simulations 2003) (Heinrich, Detecting a Bad Random Number Generator 2004) et (Heinrich, TRandom PitFalls 2006), souligne le

besoin pour la communauté scientifique de disposer d'outils adaptés à la production de résultats de simulations stochastiques les plus fiables possibles. Les travaux réalisés pendant ce doctorat ont permis de constater l'absence ou l'insuffisance de tels outils, de proposer des méthodes novatrices et de les implémenter dans des outils logiciels en accord avec l'état de l'art dans le domaine de la distribution des réplications des simulations stochastiques.

Par exemple, les faiblesses statistiques constatées dans le générateur James Random utilisé par le simulateur GATE, lui même utilisé en physique médicale ont été présentées à la communauté de physique lors du symposium NSS (Nuclear Science Symposium and Medical Imaging Conference) en 2006 à San Diego (CA, USA) puis lors de la réunion « OpenGATE collaboration meeting » en mai 2008. A l'issue de ces communications, la communauté OpenGATE a choisi de changer le générateur de nombres pseudo-aléatoires utilisé dans GATE depuis les premières versions en 2001 pour le générateur Mersenne Twister 19937. Des constats similaires avaient été établis pour des outils de simulations très répandus comme NS-2 (Hechenleitner et Entacher, On Shortcomings of the ns-2 Random Number Generator 2002), OMNeT++ (Entacher et Hechenleitner, Pitfalls when using parallel streams in OMNET++ simulations 2003) et même pour des outils plus récent comme le logiciel d'analyse de données ROOT (Heinrich, TRandom PitFalls 2006) utilisé par les chercheurs en physique médicale.

Même si il existe des générateurs de nombres pseudo-aléatoires plus récents que le Mersenne Twister 19937 (Matsumoto et Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator 1997) et parfois plus performants comme les générateurs WELL (L'Ecuyer et Panneton, Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparison 2005) (Panneton, L'Ecuyer et Matsumoto 2006) et SFMT (Saito et Matsumoto 2008), nous avons effectué le choix d'utiliser ce générateur pour l'ensemble des travaux et sur la durée de cette thèse. Pour cette raison, j'ai fourni un million de statuts pour ce générateur de nombres pseudo-aléatoires parallélisé par indexation de séquences à l'aide d'un générateur de qualité cryptographique. Ce travail a été réalisé grâce à l'utilisation de la bibliothèque DistRNG implémentée pendant mon doctorat. La communauté de physique médicale peut, grâce à ce travail, exploiter la puissance des dizaines de milliers de processeurs de l'organisation virtuelle biomédicale de la grille de

calcul européenne EGEE pour exécuter des simulations GATE. Sans ces travaux, il était impossible de travailler rigoureusement à une telle échelle.

La bibliothèque, DistRNG, de génération parallèle de nombres pseudo-aléatoires que j'ai proposée, est libre, implémentée en Java, et disponible sur la forge logicielle sourceforge[56]. Celle-ci implémente des méthodes de parallélisation en accord avec l'état de l'art. Elle est par exemple la seule bibliothèque Java à fournir un générateur de nombres pseudo-aléatoires parallèle utilisant la création dynamique de générateurs séquentiels Mersenne Twister proposée par Makoto Matsumoto.

Le fondement de DistRNG est le concept novateur de statuts génériques. Ceux-ci contiennent l'état des variables utilisées par les algorithmes de génération de nombres pseudo-aléatoires. Ces statuts contiennent en outre des méta-informations sur l'algorithme de génération et sur l'utilisation de la séquence de nombres pseudo-aléatoires dans un contexte distribué. DistRNG implémente les classes permettant la manipulation, la consultation et la mise à jour automatique des méta-informations lors de l'exécution d'un algorithme de génération de nombres pseudo-aléatoires ou de parallélisation d'un des générateurs séquentiel de la bibliothèque.

DistRNG fournit de surcroit des classes de sérialisation transversale des statuts génériques depuis et vers un format XML lisible par l'homme et documenté. Elle tire ainsi le meilleur parti de ces statuts. Elle est par exemple la seule bibliothèque permettant d'éviter à coup sûr les corrélations croisées par chevauchement en fournissant à tout moment le nombre d'itérations de l'algorithme de génération de nombres pseudo-aléatoires utilisés par une simulation.

L'utilisation de statuts génériques dans DistRNG confère à cette bibliothèque une autre fonctionnalité exceptionnelle permettant le report à l'exécution du choix de l'algorithme de génération parallèle de nombres pseudo-aléatoires utilisé par un simulateur stochastique. Il est de ce fait possible de découpler entièrement la tâche de distribution d'un simulateur stochastique du développement de celui-ci. Cette fonctionnalité permet de plus de simplifier la mise en œuvre de tests de sensibilité des résultats d'une simulation stochastique à la qualité d'un générateur de nombres pseudo-aléatoires séquentiel et parallèle. Ce test représente la seule méthode fiable disponible à

[56] http://sourceforge.net/projects/distme (DistRNG est implémenté comme un sous projet du projet DistMe)

l'heure actuelle pour s'assurer de l'absence de biais dans les résultats d'une simulation stochastique.

DistRNG à été conçu pour être extensible et pour favoriser la factorisation du code. L'ajout d'algorithmes de génération de nombres pseudo-aléatoires dans cette bibliothèque implique l'implémentation de la classe représentant un statut pour cet algorithme. Un format de statuts en XML est alors automatiquement proposé au développeur. Cet algorithme est de plus automatiquement supporté par l'autre logiciel proposé dans ce manuscrit : DistMe.

Le logiciel DistMe permet la parallélisation automatique et rigoureuse des réplications des simulateurs stochastiques selon une approche boite-noire et génère des jobs prêts pour l'exécution sur des environnements de calcul distribués supportant la notion de sac de travail (ferme de calcul et grille de calcul). L'approche boite-noire permet de s'affranchir des limites liées au langage d'implémentation du simulateur, à l'algorithme de générations de nombres pseudo-aléatoires utilisé par celui-ci ainsi qu'à l'environnement d'exécution. Ces limites contraignent l'utilisation des autres logiciels de distribution de simulations stochastiques comme Akaroa (Pawlikowski et Yau, AKAROA: a Package for Automatic Generation and Process Control of Parallel Stochastic Simulation 1993), ASDA (Bruschi, et al. 2004) ou PMCD (Parallel Monte Carlo Driver) (Mendes et Pereira, Parallel Monte Carlo Driver (PMCD)-a software package for Monte Carlo simulations in parallel 2003). DistMe génère pendant le processus de distribution de la simulation : les fichiers statuts dans un format spécifique au générateur de nombres pseudo-aléatoires du simulateur, et tous les fichiers nécessaires au lancement des jobs de simulation pour l'environnement d'exécution distribué choisi par l'utilisateur.

Dans ce manuscrit j'ai présenté des méthodes de parallélisation des générateurs basées sur l'utilisation de statuts préalablement générés associés à des méta-informations génériques sur leur utilisation en environnement distribué. L'implémentation de ces méthodes dans DistMe permet la mise en œuvre rigoureuse et à grande échelle de toutes les techniques de parallélisation pour les générateurs de nombres pseudo-aléatoires présentées dans ce manuscrit. En effet, DistMe permet la distribution rigoureuse et automatique de simulations stochastiques à partir de l'extraction de statuts d'une base de données contenant : des statuts pour divers

générateurs de nombres pseudo-aléatoires et des méta-données. Les stratégies d'extraction implémentés dans DistMe permet le choix de la technique de parallélisation utilisée et réduisent significativement les risques de corrélations inter-séquences liés à des erreurs humaines ou à des chevauchements inter-séquences.

Enfin DistMe propose un support pour l'exploration distribuée de plan d'expériences pour les simulations stochastiques. Celui-ci a été utilisé afin d'explorer les paramètres d'un modèle de croissance de l'algue *Caulerpa* en méditerranée.

La parallélisation du simulateur GATE avec le logiciel DistMe pour la distribution rigoureuse de simulations stochastiques, a fait l'objet d'une publication internationale dans une revue scientifique de physique nucléaire (R. Reuillon, D. Hill, et al., Rigorous distribution of stochastic simulations using the DistMe toolkit 2008a). Différents aspects présentant les objectifs (Reuillon et Hill, DistMe: A Generic Toolkit for Stochastic Simulation Distribution 2006) ainsi que la conception et la mise en œuvre (R. Reuillon, D. Hill, et al., A Java Based Toolbox for the Distribution of Parallel Monte Carlo Simulations. Application to Nuclear Medicine Using the GATE Simulation Package 2007) (R. Reuillon, D. Hill, et al. 2008b) de ce logiciel ont également été publiés dans les actes de conférences internationales.

Les deux outils DistRNG et DistMe implémentés durant mon travail de thèse permettent de répondre à tous les objectifs fixés en introduction de ce manuscrit (Voir Introduction Générale page 10):

- « *de paralléliser les générateurs de nombres pseudo-aléatoires existants avec diverses méthodes issues de l'état de l'art,* » : DistRNG permet de paralléliser plusieurs algorithmes de génération de nombres pseudo-aléatoires selon diverses techniques en accord avec l'état de l'art (voir partie V.3 du Chapitre 3) ; DistMe permet le stockage, et la sélection selon diverses stratégies paramétrables, de statuts « indépendants » dans une base de données en utilisant des métadonnées génériques (voir partie IV.2 du Chapitre 3).
- « *de faciliter la production de résultats de simulations stochastiques basés sur différents générateurs de nombres pseudo-aléatoires parallèles afin de les comparer,* » : DistRNG permet de retarder le choix de l'algorithme de génération de nombres pseudo-aléatoires à l'exécution (voir partie VI.1 du Chapitre 4).

- « *de permettre la génération automatique de dizaines de milliers de jobs de simulation prêts pour une exécution en environnement distribué et correspondants à la distribution des réplications d'une simulation stochastique,* » : DistMe à permis la génération de 65536 jobs pour la grille de calcul EGEE pour tester le générateur de nombres pseudo-aléatoires Mersenne Twister générique (Matsumoto et Nishimura, Dynamic Creation of Pseudorandom Number Generators 2000) (voir partie VII du Chapitre 4).
- « *de permettre l'utilisation de simulateurs existants (sans modification du code) et le support des générateurs de nombres pseudo-aléatoires déjà implémentés dans ces simulateurs,* » : DistMe est basé sur une approche boîte noire. Il permet de ce fait la distribution des réplications de n'importe quel simulateur stochastique de manière non-intrusive. Le simulateur « GATE » (Jan et al. 2004) en physique médicale et celui de croissance d'algues « simct » (Hill, Coquillard et de Vaugelas, et al., An algorithmic Model for Invasive Species Application to Caulerpa taxifolia (Vahl) C, Agardh development in the North–Western Mediterranean Sea 1998) ont été distribués (voir partie IV et V du Chapitre 4).
- « *de supporter l'utilisation d'environnements d'exécution existants comme les gestionnaires d'exécution par lot pour les fermes de calcul et les intergiciels de grille.* » : DistMe supporte la génération automatique de jobs prêts à être exécutés sur ferme de calcul pour le gestionnaire d'exécutions par lots OpenPBS, sur grille de calcul pour l'intergiciel gLite et pour le gestionnaire transversal d'exécution Ganga (Egede, et al. 2005) (voir partie VI.3 du Chapitre 3).

L'utilisation couplée des deux outils que j'ai développé et du logiciel de gestion d'exécution en environnement distribué Ganga (Egede, et al. 2005) a notamment permis l'achèvement de plusieurs simulations stochastiques parallèles sur ferme et grille de calcul. La plus importante somme de calcul réalisée pendant ce travail de thèse représente 49 années de calcul. Elle a été menée à bien par une seule personne sur la grille de calcul européenne EGEE (ces travaux ont été présenté au EGEE User Forum de 2007 sous forme de poster).

Si DistRNG et DistMe sont utilisables dans leurs versions actuelles. Plusieurs axes peuvent être explorés pour améliorer ces outils. Tout d'abord DistRNG propose une vision nouvelle de la génération parallèle de nombres pseudo-aléatoires à grande échelle. Cependant certains points peuvent être améliorés, notamment le support d'algorithmes de générations de nombres pseudo-aléatoires récents comme la version 64 bits du générateur Mersenne Twister 19937[57], les générateurs de haute qualité avec des techniques de division de cycle rapide WELL (L'Ecuyer et Panneton, Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparison 2005) (Panneton, L'Ecuyer et Matsumoto 2006) et du générateur Mersenne Twister optimisé pour architectures de processeur moderne SFMT (Saito et Matsumoto 2008).

L'implémentation de la génération de lois de probabilité dans DistRNG permettrait de simplifier son utilisation pour l'implémentation de simulations stochastiques. Une collaboration avec les travaux de Pierre L'Ecuyer sur SSJ nous intéresserait particulièrement.

Enfin, l'intégration d'une bibliothèque de tests statistiques au sein de la suite Dist permettrait le test systématique de la qualité des algorithmes de génération de nombres pseudo-aléatoires implémentés dans DistRNG. Les bases d'une bibliothèque appelée DistTest, dédiée au test séquentiel et parallèle des différents générateurs de nombres pseudo-aléatoires de DistRNG à été conçue. Le code source est disponible sur sourceforge, cependant, faute de temps, aucune version utilisable de cette bibliothèque n'est disponible à l'heure actuelle. Cette bibliothèque pourrait intégrer les techniques de la suite TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007).

Le projet DistMe est pleinement opérationnel et sa version 3.0[58] propose une approche fonctionnelle pour la distribution de simulations stochastiques. Le travail effectué dans le cadre du développement du projet DistMe fait actuellement l'objet d'un transfert technologique vers la plateforme logicielle du Cemagref baptisée SimExplorer[59] conçue pour l'exploration automatisée de modèles.

Ce travail de transfert étend grandement les capacités du logiciel DistMe et permet en plus de la distribution automatique des simulations stochastiques, la distribution de

[57] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html
[58] http://www.isima.fr/~reuillon/files/software/DistMe/3.0/distme.jnlp
[59] http://www.simexplorer.org/

plans d'expériences sur fermes et sur grilles de calcul à travers une interface graphique évoluée, basée sur l'environnement de développement NetBean[60].

SimExplorer utilise les fonctionnalités de g-Eclipse (Wolniewicz, et al. 2007) pour exécuter de manière transparente pour l'expérimentateur des simulations distribuées sur machines multiprocesseur et diverses fermes et grilles de calcul. Des stratégies d'exécution, de resoumissions et de sur-soumissions de jobs y sont implémentées afin de réduire le temps global de simulation.

De plus SimExplorer utilise un dépôt de statuts génériques sous forme de service web proposant des statuts pour plusieurs des principaux générateurs de nombres pseudo-aléatoires utilisés à l'heure actuelle. Ce service web fournit la possibilité de sélectionner les statuts selon des stratégies paramétrables par l'utilisateur.

Il sera ainsi possible de rendre disponible à travers ce dépôt public de statuts, les résultats à des tests statistiques séquentiels et parallèles des séquences de nombres pseudo-aléatoires proposées. Ce travail permettra l'implémentation de stratégies de sélection prenant en compte les résultats à des tests statistiques des séquences en vue de leur sélection. Par exemple un expérimentateur pourra choisir de sélectionner seulement les statuts échouant à moins de 5 tests de la batterie de tests statistiques TestU01 (L'Ecuyer et Simard, TestU01: A C Library for Empirical Testing of Random Number Generators 2007). Le double travail de parallélisation des générateurs de nombres pseudo-aléatoires et des tests des séquences de nombres sera ainsi factorisé et rendu disponible à un niveau mondial à travers internet.

[60] http://www.netbeans.org/

Travaux cités

Ackermann, J. «Parallel random number generator for inexpensive configurable hardware cells.» *Computer Physics Communications*, 2001: 293-302.

Afflerbach, L., et H. Grothe. «The lattice structure of pseudo-random vectors generated by matrix generators.» *Journal of Computational and Applied Mathematics*, 1988: 127-131.

Alexandrov, D., M. Ibel, K.E. Schauser, et C.J. Scheiman. «SuperWeb: Towards a global web-based parallel computing infrastructure.» *Proceedings of the 11 th IEEE International Parallel Processing Symposium (IPPS).* Geneva, 1997. 100-106.

Allison et al. «Geant4 Developments and Applications.» *IEEE Transactions on Nuclear Science*, 2006: 270-278.

Anderson, D.P. «BOINC: A System for Public-Resource Computing and Storage.» *Proceedings of the 5th IEEE/ACM International Workshop on the Grid Computing.* 2004. 4-10.

Anderson, D.P. et al. «SETI@home: An Experiment in Public-Resource Computing.» *Communication of the ACM*, 2002: 56-61.

Anderson, D.P., et G. Fedak. «The Computational and Storage Potential of Volunteer Computing.» *Prceedings of the IEEE/ACM International Symposium on Cluster Computing and the Grid.* IEEE Press, 2006. 73- 80.

Anderson, D.P., et J. Kubiatowicz. «The worldwild computer.» *Sci. Am.*, 2002: 40-47.

Andreo, P. «Monte Carlo techniques in medical radiation physics.» *Phys. Med. Biol.*, 1991: 861-920 .

Antoch, J., J.M. Deshouillérs, et G.P. Purnaba. «Revisiting the pseudorandom number generator ran1 from the NUMERICAL RECIPES.» *Computational Statistics & Data Analysis*, 1998: 487-495.

Aumage, O., L. Eyraud, et R. Namyst. «Efficient inter-device data-forwarding in the madeleine communication library.» *Proceedings of the 15th Intl. Parallel and Distributed Processing Symposium, 10th Heterogeneous ComputingWorkshop, HCW 2001.* San Francisco, 2001. 853-863.

Aussem, A., et D.R.C. Hill. «Neural networks metamodelling for the prediction of Caulerpa taxifolia development in the Mediterranean sea.» *Neurocomputing Letters*, 2000: 71-78.

Badal, A., et J. Sempau. «A package of Linux scripts for the parallelization of Monte Carlo simulations.» *Computer Physics Communications*, 2006: 440-450.

Baduel, L., et al. «Programming, Composing, Deploying for the Grid.» Dans *GRID COMPUTING: Software Environments and Tools*. Springer-Verlag, 2006.

Balci, O. «Verification, validation, and testing.» Dans *The Handbook of Simulation*, de J. Banks, 335-393. New York, NY: John Wiley & Sons, 1998.

Banks, J. *Handbook of simulation: Principles, Methodology, Advances, Applications, and Practice.* Wiley-Interscience, 1998.

Barabasi, A. et al. «Parasitic computing.» *Nature*, 2001: 894-897.

Barak, A., et A. Litman. «MOS: a Multicomputer Distributed Operating System.» *Software – Practice and Experience*, 1985: 725–737.

Barak, A., et R. Wheeler. «MOSIX: An Integrated Multiprocessor UNIX.» *Proceedings of the Winter 1989 USENIX Conference.* 1989. 101–112.

Baratloo, A., M. Karaul, Z. Kedem, et P. Wyckoff. «Charlotte: Metacomputing on the Web.» *Future Generation Computer Systems*, 1999: 559-570.

Barbero, M., F. Jouault, J. Gray, et J. Bézivin. «A Practical Approach to Model Extension.» *Proceedings of the Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007.* Haifa, Israel, 2007. 32-42.

Bégin, M.E. *An EGEE comparative study: Grids and Clouds Evolution or Revolution?* CERN, 2008, 1-33.

Beiriger, J., W. Johnson, H. Bivens, S. Humphreys, et R. Rhea. «Constructing the ASCI Grid.» *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing.* 2000. 193–199.

Bézevin, J. «On the Unification Power of Models.» *Software and System Modeling*, 2005: 171-188.

Bézivin, J. «In Search of a Basic Principle for Model Driven Engineering.» *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, 2004: 21-24.

Bézivin, J., M. Barbero, et F. Jouault. «On the Applicability Scope of Model Driven Engineering (Draft Version).» *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007).* 2007.

Blum, M., et S. Micali. «How to generate cryptographically strong sequences of pseudo-random bits.» *SIAM J. Comput.*, 1984: 850-864.

Box, G., et N.R. Draper. *Empirical Model Building and Response Surfaces.* New York: Wiley, 1987.

Box, G., S. Hunter, et W.G. Humter. *Statistics for experimenters.* New York: Wiley, 1978.

Brecht, T., H. Sandhu, M. Shan, et J. Talbot. «ParaWeb: Towards World-Wide Supercomputing.» *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications.* 1996. 181-188.

Breton, V., et I. Buvat. «Feasibility and value of fully 3D Monte Carlo reconstruction in Single Photon Emission Computed Tomography.» *Nucl. Instr. and Meth. Phys. Res. A*, 2004: 195-200.

Brun, R., et F. Rademakers. «ROOT: An object oriented data analysis framework.» *Nucl. Instrum. Methods A*, 1997: 81-86.

Brune, M.A., G.E. Fagg, et M.M. Resch. «Message-passing environments for metacomputing.» *Future Generation Computer Systems*, 1999: 699-712.

Brunett, S. et al. «Application Experiences with the Globus Toolkit.» *Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing.* 1998. 81-89.

Bruschi, S.M., R.H.C. Santana, M.J. Santana, et T.S. Aiza. «An Automatic Distributed Simulation Environment.» *Proceedings of the 2004 Winter Simulation Conference.* 2004. 378-385.

Bryant, R.E. «Simulation of packet communication architecture computer systems.» Massachusetts Institute of Technology, 1977.

Cai, W, Z. Yuan, M.Y.H. Low, et S.J. Turner. «Federate migration in HLA-based simulation.» *Future Generation Computer Systems*, 2005: 87-95.

Calvin, J., A. Dikens, B. Gaines, P. Metzger, D. Miller, et D. Owen. «The SIMNET Virtual World Architecture.» *Proceedings of VRAIS'93.* 1993. 450-455.

Cappello, F., et al. «Computing on large-scale distributed systems: Xtrem Web architecture, programming models, security, tests and convergence with grid.» *Future Gener. Comput. Syst.*, 2005: 417-437.

Cappello, P., B. Christiansen, M. Ionescu, M. Neary, K. Schauser, et D. Wu. «Javelin: Internet-based parallel computing using Java.» *Concurrency: Practice and Experience*, 1997: 1139-1160.

Caromel, D. «Toward a Method of Object-Oriented Concurrent Programming.» *Communication of the ACM*, 1993: 90-102.

Caromel, D., A. di Costanzo, et C. Mathieu. «Peer-to-peer for computational grids: mixing clusters and desktop machines.» *Parallel Computing*, 2007: 275-288.

Caron, E., A. Chis, F. Desprez, et A. Su. «Design of plug-in schedulers for a GridRPC environment.» *Future Generation Computer Systems*, 2008: 46-57.

Castro, M. «Real Random Numbers Without Additional Hardware.» *Proceedings of the AUUG 96 and Asia Pacific World Wide Web Conference.* Melbourne, 1996. 8.

Chaitin, G.J. «On the Length of Programs for Computing Finite Binary Sequences.» *Journal of the ACM*, 1966: 547-569.

Chandy, K.M., et J. Misra. «Asynchronous distributed simulation via a sequence of parallel computations.» *Communication of the ACM*, 1981: 198-205.

—. «Distributed Simulation: A case study in design and verification of distributed programs.» *IEEE Transaction on Software Engineering*, 1979: 440-452.

Chang, M.W., W. Lindstrom, A.J. Olson, et R.K. Belew. «Analysis of HIV Wild-Type and Mutant Structures via in Silico Docking against Diverse Ligand Libraries.» *J. Chem. Inf. Mode*, 2007: 1258 - 1262.

Chen, D., G.K. Theodoropoulos, S.J. Turner, W. Cai, R. Minson, et Y. Zhang. «Large scale agent-based simulation on the grid.» *Future Generation Computer Systems*, 2008: In Press.

Cheon, S., C. Seo, S. Park, et B. Zeigler. «Design and implementation of distributed DEVS simulation in a peer to peer network system.» *Proceeding of the Advanced Simulation Technologies Conference — Design, Analysis, and Simulation of Distributed Systems Symposium, Arlington, VA.* Arlington, VA, 2004.

Chunlin, L., et L. Layuan. «Integrate software agents and CORBA in computational grid.» *Computer Standard and Interfacess*, 2003: 357-371 .

Clauss, C., M. Pöppe, et T. Bemmerl. «Optimising MPI applications for heterogeneous coupled clusters with MetaMPICH.» *Proceedings of the IEEE International Conference on Parallel Computing in Electrical Engineering.* Dresden, Germany, 2004. 7 - 12.

Coddington, P. D. «Analysis of Random Number Generators Using Monte Carlo Simulation.» *Int. J. Mod. Phys. C*, 1994: 547-560.

Coddington, P. D. «Random number generator for parallel computers.» NHSE Review, 2nd issue, Northeast Parallel Architecture Center, 1996.

Coddington, P.D. «Tests of random number generators using Ising model simulations.» *Int. J. Mod. Phys. C*, 1996b: 295-303.

Coddington, P.D., et A.J. Newell. «JAPARA – A Java Parallel Random Number Library for High-Performance Computing.» *Proceeding of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04) - Workshop 5.* 2004. 156-166.

Coddington, P.D., et S.H. Ko. «Techniques for Empirical Testing of Parallel Random Number Generators.» *Proceedings of the 12th international Conference on Supercomputing, ICS '98.* Melbourne, Australia: ACM Press, New York, 1998. 282-288.

Coddington, P.D., J.A. Mathew, et K.A. Hawick. «Interfaces and Implementations of Random Number Generators for Java Grande Applications.» *Proceedings of High Performance Computing and Networks (HPCN).* Amsterdam, NL, 1999. 873-883.

Compagner, A. «Operational conditions for random-number generation.» *Physical Review E*, 1995: 5654-5645.

Coquillard, P., T. Thibaut, D.R.C. Hill, J. Gueugnot, C. Mazel, et Y. Coquillard. «Modelling and simulation of the mollusc Ascoglossa Elysia subornata Verril population dynamic: Application to the potential biocontrol of Caulerpa taxifolia (Valhl) C. Agarh growth in the Mediterranean Sea.» *Ecological Modeling*, 2001: 1-16.

Dagum, L., et R. Menon. «OpenMP: an industry standard API for shared-memory programming.» *Computational Science and Engineering, IEEE*, 1998: 46-55.

De'Matteis, A., et S. Pagnutti. «Controlling correlations in parallel Monte Carlo.» *Parallel Computing*, 1995: 73–84.

—. «Long-range correlations in linear and non-linear random number generators.» *Parallel Computing*, 1990: 207-210.

—. «Parallelisation of random number generators and long-range correlations.» *Parallel Computing*, 1990b: 155-164.

DeMatteis, A., et S. Pagnutti. «A class of parallel random number generators.» *Parallel Comput.*, 1990c: 193-198.

DeMatteis, A., J. Eichenauer-Herrmann, et H. Grothe. «Computation of critical distances withing multiplicative congruential pseudorandom number sequences.» *J. Comp. App. Math.*, 1992: 49-55.

Denis, A., Pérez C., T. Priol, et A. Ribes. «Parallel CORBA Objects for Programming Computational Grids.» *Distributed Systems Online*, 2003.

Dikaiakos, M. «Grid Benchmarking: Vision, Challenges, and Current Status.» *Concurrency and Computation: Practice and Experience*, 2007: 89-105.

Drira, K., A. Martelli, et T. Villemur. *Cooperative Environments for Distributed Systems EngineeringThe Distributed System Engeeniring.* Springer, 2001.

Drutarovsky, M., et P. Galajda. «A Robust Chaos-Based True Random Number Generator Embedded in Reconfigurable Switched-Capacitor Hardware.» *Proceeding of the 17th International Conference Radioelektronika.* Brno, Czech Republic, 2007.

Durst, M.J. «Testing parallel random number generators.» *In: Computing Science and Statistics: Proceedings of the XXth Symposium on the Interface.* 1988. 228-231.

Egede, U., et al. «Ganga user interface for job definition and management.» *Proc. Fourth International Workshop on Frontier Science: New Frontiers in Subnuclear Physics.* Milan, Italie, 2005. 367-374.

Ehrig, K., C. Ermel, et S. Hansgen. «Generation of visual editors as eclipse plug-ins.» *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* 2005. 134 -143.

Eichenauer-Herrmann, J. «Statistical Independence of a New Class of Inversive Congruential Pseudorandom Numbers.» *Mathematics of Computation*, 1993: 375-384.

Eichenauer-Herrmann, J., et H. Grothe. «A remark on long-range correlations in multiplicative congruential pseudo random number generators.» *Numer. Math.*, 1989: 609-611.

El Bitar, Z., D. Lazaro, V. Breton, D.R.C. Hill, et I. Buvat. «Fully 3D Monte Carlo image reconstruction in SPECT using functional regions.» *Nucl. Instr. Meth. Phys. Res.*, 2006: 399-403.

Entacher, K., A. Uhl, et S. Wegenkitt. «Parallel Random Number Generation: Long-Range Correlations Among Multiple Processors.» *Lecture Notes in Computer Science*, 1999: 107-116.

Entacher, K., et B Hechenleitner. «Pitfalls when using parallel streams in OMNET++ simulations.» *Proceedings of Inter-domain Performance and Simulation (IPS) Workshop.* Salzburg, Austria, 2003. 11-20.

Ewing, G., K. Pawlikowski, et D. McNickle. «Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation.» *Proc. European Simulation Multiconference ESM'99.* Warsaw, 1999. 175-181.

Fagg, G.E., K.S. London, et J. Dongarra. «Mpi connect managing heterogeneous mpi applications ineroperation and process control.» *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Springer-Verlag, 1998. 93–96.

Favre, J.M. «Towards a Basic Theory to Model Model Driven Engineering, .» *Proceedings of the WiSME 2004.* Lisbon, Portugal, 2004.

Fedak, G. et al. «Xtrem Web: A generic global computing system.» *Proceedings of the First International Workshop on Global and Peer-to-Peer Computing on Large Scale Distributed Systems at the First IEEE/ACM International Symposium on Cluster Computing and the Grid.* IEEE Press, 2001. 582-588.

Ferrenberg, A.M., D.P. Landau, et Y.J. Wong. «Monte Carlo simulations: Hidden errors from "good" random number generators.» *Phys. Rev. Let.*, 1992: 3382-3384.

—. «Monte Carlo Simulation: Hidden errors from “good” random number generator.» *Physics Review Letter*, 1992: 3382-3384.

Ferscha, A., et S.K. Tripathi. *Parallel and distributed simulation of discrete event systems.* University of Maryland at College Park, 1994, 51.

Filk, T., et M.M. Fredenhagen. «Long range correlations in random number generators and their influence on Monte Carlo simulations.» *Physics Letters B*, 1985: 125-130.

Fishman, G.S. *Monte Carlo: Concepts, Algorithms, and Applications.* New York: Springer, 1995.

Fishman, G.S., et L.R. Moore. «An exhaustive analysis of multiplicative congruential generators with modulus M=2^31-1.» *SIAM J. Sci. Stat. Comput.*, 1986: 24-45.

Fishwick, P.A. *Simulation Model Design and Execution.* Prentice Hall, 1995.

Foster, I., C. Kesselman, et S. Tuecke. «The Anatomy of the Grid.» *International Journal of High Performance Computing*, 2001: 200-222.

Foster, I., et C. Kesselman. «The Globus project: a status report.» *Future Generation Computer Systems*, 1999: 607-621.

—. *The Grid: Blueprint for a New Computing Infrastructure 2nd Edition.* Morgan Kaufmann, 2004.

—. *The Grids: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, 1999.

Frattolillo, F. «Running large-scale applications on cluster grids.» *International Journal of High Performance Computing Applications*, 2005: 157–172.

Fujimoto, R.M. «Parallel discrete event simulation.» *Communication of the ACM*, 1990: 30-53.

Gamma, E., R. Helm, R. Johnson, et J.M. Vlissides. «Design Patterns: Elements of Reusable Object-Oriented Software.» 1995.

Geist, A., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, et V. Sunderam. *PVMParallel Virtual Machine : a Users' Guide and Tutorial for Networked Parallel Computing.* MIT press, 1994.

Ghormley, D.P., D. Petrou, S.H. Rodrigues, A.M. Vahdat, et T.E. Anderson. «Glunix: A Global Layer Unix for a Network of Workstations.» *Software Practice and Experience*, 1998: 929-961.

Gleick, J. *La théorie du chaos: Vers une nouvelle science.* Paris: Albin Michel, 1989.

Gonnet, G. «Repeating Time Test for U(0,1) Random Number Generators.» Informatik, ETH, Zurich, 2003.

Grassberger, P. «On correlations in 'good' random number generators.» *Physics Letter A*, 1993: 43-46.

Greenfield, J., et K. Short. «Software factories assembling applications with pattern, model, framework and tools.» *Proceedings of OOPSLA.* Anaheim (CA, USA), 2003.

Gregoretti, F., G. Laccetti, A. Murli, G. Oliva, et U. Scafuri. «MGF: A grid-enabled MPI library.» *Future Generation Computer Systems*, 2008: 158-165.

Hamming, R.W. *Coding and Information Theory.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

Hechenleitner, B. «Defects in Random Number Routines of Well-Known Network Simulators and Appropriate Improvements.» Salzburg, Austria, 2004.

Hechenleitner, B., et K. Entacher. «On Shortcomings of the ns-2 Random Number Generator.» *Communication Networks and Distributed Systems Modeling and Simulation*, 2002.

Heinrich, J. «Detecting a Bad Random Number Generator.» University of Pennsylvania, 2004.

Heinrich, J. «TRandom PitFalls.» University of Pennsylvania, 2006.

Heinzlreiter, P., et D. Kranzlmüller. «Visualization Services on the Grid: The Grid Visualization Kernel.» *Parallel Processing Letters*, 2003: 135-148.

Hellekalek, P. «Don't trust parallel Monte Carlo.» *Proceedings of the twelfth workshop on Parallel and distributed simulation.* Alberta, Canada, 1998. 82–89.

—. «Good random number generators are (not so) easy to find.» *Mathematics and Computer in Simpulation*, 1998b: 485–505.

Herr, W., E. McIntosh, et F. Schmidt. «Large scale beam-beam simulations for the CERN LHC using distributed computing.» *Proceedings of EPAC.* Edinburgh, Scotland, 2006. 526-528.

Hill, D.R.C. «A Design Pattern for Object-Oriented Distributed Simulation of Large Scale Ecosystems.» *Proceedings of the 1997 Summer Simulation Conference.* Arlington, Virginie, 1996. 945-950.

Hill, D.R.C., P. Coquillard, A. Aussem, J. de Vaugelas, T. Thibaut, et A. Meineisz. «Modeling the Ultimate Seaweed.» *Simulation*, 2000: 126-134.

Hill, D.R.C., P. Coquillard, et J. de Vaugelas. «Discrete-Event Simulation of Alga Expansion.» *Simulation* , 1997: 269-277.

Hill, D.R.C., P. Coquillard, J. de Vaugelas, et A. Meinesz. «An algorithmic Model for Invasive Species Application to Caulerpa taxifolia (Vahl) C, Agardh development in the North–Western Mediterranean Sea.» *Ecological Modeling*, 1998: 251-265.

Holmes, V. «Parallel algorithms on multiple processor architectures.» Ph. D dissertation, Computer Science Department, University of Texas, Austin, 1978.

Hong, G.P., et T.G. Kim. «A Framework for Verifying Discrete Event Models whithin a DEVS-based System Development Methodology.» *Transactions of The Society for Computer Simulation*, 1996: 19-34.

Howell, F., et R. McNab. «simjava: a discrete event simulation package for Java with applications in computer systems modelling.» *Proceeding s of the First Conference on Web-based Modelling and Simulation.* San Diego, CA, 1998.

Hu, Y.C., H. Lu, A.L. Cox, et W. Zwaenepoel. «OpenMP for Networks of SMPs.» *Journal of Parallel and Distributed Computing*, 2000: 1512-1530.

Hwang, M.H. «Identifying equivalence of DEVSs: Language approach.» Édité par A. Bruzzone et M. Itmi. *Proceedings of 2003 Summer Computer Simulation Conference.* Montreal, Canada, 2003. 319–324.

—. «Tutorial: Verification of Real-time System Based on Schedule-Preserved DEVS.» *Proceedings of 2005 DEVS Symposium.* San Diego, CA, 2005.

Imamura, T., Y. Tsujita, H. Koide, et , H. Takemiya. «An architecture of stampi: Mpi library on a cluster of parallel computers.» *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Springer-Verlag, 2000. 200–207.

Jacq, N. et al. «Virtual screening on Large Cale Grids.» *Parallel Computing*, 2007: 289-301.

James, F. «A review of pseudorandom number generators.» *Computer Physics Communications*, 1990: 329-344.

Jan et al. «GATE: a simulation toolkit for PET and SPECT.» *Phys. Med. Biol.*, 2004: 4543-4561.

Jefferson, D.R. «Virtual Time.» *ACM Transaction, Programming Language System*, 1985: 404-425.

Jézéquel, J.M., S. Gérard, C. Mraidha, et B. Baudry. «Le génie logiciel et l'IDM : une approche unificatrice par les modèles.» Dans *L'ingénierie dirigée par les modèles*, de J.M. Favre, J. Estublier et M. Blay-Fornarino. 2006.

Johnston, W.E., D. Gannon, et B. Nitzberg. «Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid.» *Proceedings of the The 8th IEEE International Symposium on High Performance Distributed Computing.* 1999. 197-204.

Jousson, O., et al. «Invasive alga reaches California.» *Nature*, 2000: 157-158.

Karonis, N.T., B. Toonen, et I. Foster. «MPICH-G2:next term A Grid-enabled implementation of the Message Passing Interface.» *Journal of Parallel and Distributed Computing*, 2003: 551-563.

Kendall, M.G., et B.B. Smith. *Tables of Random Sampling Numbers.* Cambridge, England: Cambridge University Press, 1939.

Kielmann, T., H.E. Bal, J. Maassen, R. van Nieuwpoort, R.F.H. Hofman, et K. Verstoep. «Programming environments for high-performance grid computing: The albatross project.» *Future Generation Comput. Syst.*, 2002: 1113–1125.

Kim, T.G., S.M. Cho, et W.B. Lee. «Framework for systems development: Unified specification for logical analysis, performance development: Unified specification for logical analysis, performance.» Chap. 8 dans *Discrete Event Modeling and Simulation Technologies*, 131–166. New-York: Springer-Verlag, 2001.

Kleijnen, J.P.C. *Statistical Tools for simulation practitioners.* New-York: Dekker, 1987.

Kleijnen, J.P.C., et W. Van Groenendaal. *Simulation: A Statistical Perspective.* Chichester: John Wiley, 1992.

Knop, F., E. Mascarenhas, V. Rego, et V. Sunderam. «An Introduction to Fault Tolerant Parallel Simulation with EcliPSe.» *Proceedings of the Winter Simulation Conference.* 1994.

Knop, F., et V. Rego. «Parallel Cluster Labeling on a Network of Workstations.» *Proceedinds of the 13th Brazilian Symposium on Computer Networks.* 1995.

Knuth, D. E. *The art of computer programming, Seminumerical Algorithms.* 3rd. Vol. 2. Addision-Wesley, 1997.

—. *The art of computer programming, Vol. 2 Seminumerical Algorithms 3rd Edition.* Addision-Wesley, 1997.

Knuth, D. *The Art of Computer Programming: Seminumerical Algorithms.* Vol. 2. Addison-Wesley, 1969.

Koenker, R. *Lecture 5: "Monte Carlo I: Generating random variables".* 2007. http://www.econ.uiuc.edu/~roger/courses/476/lectures/L5.pdf.

Kolmogorov, A.N. «Three Approaches to the Quantitative Definition of Information.» *Problems of Information Transmission*, 1965: 1-7.

Kopela, E. et al. «SETI@home - Massively distributed computing for seti.» *Computing in Science & Engineering*, 2001: 78-83.

Kupczyk, M., R. Lichwała, N. Meyer, B. Palak, M. Płóciennik, et P. Wolniewicz. «"Applications on demand" as the exploitation of the Migrating Desktop.» *Future Generation Computer Systems*, 2005: 37-44.

L'Ecuyer, P. «Combined Multiple Recursive Generators.» *Operations Research*, 1996: 816–822.

L'Ecuyer, P.. «Good Parameter and Implementations for Combined Multiple Recursive Random Number Generators.» *Operations Research*, 1999: 159–164.

L'Ecuyer, P., F. Blouin, et R. Couture. «A Search for Good Multiple Recursive Generators.» *ACM Trans. on Modeling and Computer Simulation*, 1993: 87-98 .

L'Ecuyer, P., R. Simard, E.J. Chen, et W.D. Kelton. «An object-oriented random-number package with many long streams and substreams.» *Operations Research*, 2002: 1073-1075.

Laforenza, D. «Grid programming: some indications where we are headed.» *Parallel Computing*, 2002: 1733-1752.

Lamport, L. «Time, clocks and the ordering of events in a distributed system.» 21, n° 7 (1978): 558-565.

Landis, J.R., et A.R. Feinstein. «An empirical comparison of random numbers acquired by computer-generation and from the Rand tables.» 1973: 322-326.

Laszewski, G. von. «The Grid-Idea and Its Evolution.» *Information Technology*, 2005: 319-329.

Laszewski, G. von, J. Gawor, S. Krishnan, et K. Jackson. «Commodity Grid Kits Middleware for Building Grid Computing Environments.» *Grid Computing: Making the Global Infrastructure a Reality*, 2003: 639-656.

Laszewski, G. von, M. Hategan, et D. Kodeboyina. «Work Coordination for Grid Computing.» *Grid Technologies: Emerging from Distributed Architectures to Virtual Organizations*, 2006: 309-329.

Laure, E., S.M. Fisher, A. Frohner, et C. Grandi. «Programming the Grid using gLite.» *Computational Methods in Science and Technology*, 2006: 33–45.

Law, A.M., et W.D. Kelton. *Simulation Modeling & Analysis.* Édité par E.M. Munson et M. Luhrs. 1991.

Lazaro, D., V. Breton, et I. Buvat. «Feasibility and value of fully 3D Monte Carlo reconstruction in Single Photon Emission Computed Tomography.» *Nucl. Instr. and Meth. Phys. Res. A*, 2004: 195 - 200.

Lazaro, D., Z. El Bitar, V. Breton, D.R.C. Hill, et I. Buvat. «Fully 3D Monte Carlo reconstruction in SPECT: a feasibility study.» *Phys Med Biol*, 2005: 3739-3754.

L'Ecuyer, P. «Efficient and Portable Combined Random Number Generators.» *Communications of the ACM*, 1988: 742-749.

—. «Random Number Generators and Empirical Tests.» *Lecture Notes in Statistics*, 1998: 124-138.

—. «A table of linear congruential generators of different sizes and good lattice structure.» *Mathematics of Computation*, 1999: 249-260.

L'Ecuyer, P., et E. Buist. «Simulation in Java with SSJ.» *Proceedings of the 2005 Winter Simulation Conference.* 2005. 611-620.

L'Ecuyer, P., et F. Panneton. «Fast Random Number Generators Based on Linear Recurrences Modulo 2: Overview and Comparison.» *Proceedings of the 2005 Winter Simulation Conference.* 2005. 110-119.

L'Ecuyer, P., et J. Leydold. «rstream: Streams of random numbers for stochastic simulation.» *R News*, 2005: 16-20.

—. «rstream: Streams of random numbers for stochastic simulation.» *R News*, 2005b: 16-20.

L'Ecuyer, P., et R. Simard. «TestU01: A C Library for Empirical Testing of Random Number Generators.» *ACM Transactions on Mathematical Software*, 2007: Article 22.

L'Ecuyer, P., F. Blouin, et R. Couture. «A search for good multiple recursive generators.» *ACM Transaction on Modeling And Computer Simulation*, 1993.

L'Ecuyer, P., R. Simard, E.J. Chen, et W.D. Kelton. «An object-oriented random-number package with many long streams and substreams.» *Operations Research*, 2002: 1073-1075.

Ledeczi, R., et al. «Composing domain-specific design environments.» *Computer*, 2001: 44 - 51.

Lehmer, D.H. «Mathematical methods in large-scale computing units.» *Proceedings of 2nd Symposium on Large-Scale Digital Calculating Machinery.* Cambridge, MA, 1949. 141-146.

Lenica, A., F. Ogel, F. Peschanski, et J.P. Briot. «Agent-based grid resource management.» *Proceeding of the International Conference on Computational Science (ICCS'2006) .* Reading, United-Kingdom, 2006.

Leroudier, J. *La simulation à événements discrets.* Editions Hommes et Techniques, 1980.

Levy, P. *Stochastic Processes and Brownian Motion.* Paris: Gauthier Villars, 1965.

Lewis, P., A. Goodman, et J. Miller. «A pseudo-random umber generator for the System/360.» *IBM Syst. J.*, 1969: 136-146.

Li, Y., et M. Mascagni. «Analysis of Large-Scale Grid-Based Monte Carlo Applications.» *International Journal of High Performance Computing Applications*, 2003: 369-382.

Lian, C.C., F. Tang, P. Issac, et A. Krishnan. «GEL: Grid execution language.» *Journal of Parallel and Distributed Computing*, 2005: 857-869.

Liang, Y., et P.A. Withlock. «A new empirical test for parallel pseudo-random number generators.» *Mathematics and Computers in Simulation*, 2001: 149–158.

Lin, Y.B., et P.A. Fishwick. «Asynchronous parallel discrete event simulation.» *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 1996: 397-412.

Litzkow, M., et M. Solomon. «Supporting Checkpointing and Process Migration outside the UNIX Kernel.» *Proceedings of the USENIX Winter Conference.* 1992. 283–290.

Litzkow, M., M. Livny, et M. Mutka. «Condor - A Hunter of Idle Workstations.» *Proceedings of the 8th International Conference on Distributed Computing Systems.* 1988. 104–111.

Livny, M. «High-throughput resource management.» Édité par Foster et Kesselman. *In the Grid: Blueprint for a New Computing Infrastructure*, 1999: 311-337.

Longepierre, L., A. Robert, F. Levi, et P. Francour. «How invasive alga species (Caulerpa taxifolia) induces changes in foraging strategies of the benthivorous fish Mullus surmuletus in coastal Mediterranean ecosystems.» *Biodiversity Conservation*, 2005: 365-376.

Lönnblad, L. «CLHEP – a project for designing a C++ class library for high energy physics.» *Computer Physics Communication*, 1994: 307-316.

Lorenz, E. N. «Deterministic nonperiodic flow.» 1963: 130-141.

Luckow, A., et B. Schnor. «Migol: A fault-tolerant service framework for MPI applications in the grid.» *Future Generation Computer Systems*, 2008: 142-152.

Mackie, T.R. *Applications of the Monte Carlo method in radiotherapy.* Vol. 3, chez *The dosimetry of ionizing radiation*, édité par Kase K.R., B.E. Bjärngard et F.H. Attix, 541-620. San Diego: Academic Press, 1990.

Maigne, L., et al. «Parallelization of Monte Carlo simulations and submission to a grid environment.» *Parallel Processing Letters*, 2004: 177-196.

Marsaglia, G. A. «Random Numbers fall mainly in the planes.» *Proceedings of the National Academy of Sciences.* 1968.

Marsaglia, G., et W.W. Tsang. «Some difficult-to-pass tests of randomness.» *Journal of Statistical Software*, 2002: 1-8.

Marsaglia, G.A. «A current view of random number generators.» *Computer Science and Statistics: The Interface*, 1985: 3-10.

Mascagni, M. «Parallel Pseudorandom Number Generation.» *SIAM News*, 1999: 1-10.

Mascagni, M., D. Ceperley, et A. Srinivasan. «SPRNG: A Scalable Library for Pseudorandom Number Generation,.» *ACM Transaction on Mathematical Software*, 2000: 618-619.

—. «SPRNG: A Scalable Library for Pseudorandom Number Generation,.» *ACM Transaction on Mathematical Software*, 2000: 618-619.

Mascagni, M., et A. Srinivasan. «Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation.» *ACM Transaction on Mathematical Software*, 2000: 436-461.

—. «Parameterizing parallel multiplicative lagged-Fibonacci generators.» *Parallel Computing*, 2004: 899-916.

Mascagni, M., et H. Chi. «Parallel linear congruential generators with Sophie-Germain moduli.» *Parallel Computing*, 2004: 1217-1231.

Mateos, C., A. Zunino, et M. Campo. «JGRIM: An approach for easy gridification of applications.» *Future Generation Computer Systems*, 2008: 99-118.

Matsumoto, M., et T. Nishimura. «Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator.» *Proceedings of the 29th conference on Winter simulation.* 1997. 127-134.

—. «A Nonempirical Test on the Weight of Pseudorandom Number Generators.» *Monte Carlo and Quasi-Monte Carlo methods*, 2002, éd. Ed. K.T. Fang, F.J.Hickernel, and H. Niederreiter: 381-395 .

—. «Dynamic Creation of Pseudorandom Number Generators.» *Monte Carlo and Quasi-Monte Carlo Methods*, 2000, éd. Springer: 56-69.

—. «Sum discrepancy test on pseudorandom number generators.» 2003: 431-442.

Matsumoto, M., M. Saito, H. Haramoto, et T. Nishimura. «Pseudorandom Number Generation: Impossibility and Compromise.» *Jounal of Universal Computer Science*, 2006: 672-690.

Maurer, U. «A Universal Statistical Test for Random Bit Generators.» *Journal of Cryptology*, 1992: 89-105.

Mazumdar, S., R. Mathew, et J.F. Jr. Leathrum. «A Strategy for Distributing Simulation for Statistical Analysis.» *Proceedings of the 2004 Summer Computer Simmulation Conference.* 2004. 294-299.

McNab, R., et F.W. Howell. «Using Java for Discrete Event Simulation.» *Proceedings Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW).* Edinburgh, 1996. 219-228.

Mendes, B., et A. Pereira. «Parallel Monte Carlo Driver (PMCD)-a software package for Monte Carlo simulations in parallel.» *Computer Physics Communications*, 2003: 89-95.

Mendes, B., et A. Pereira. «Software Manual for the Parallel Monte Carlo Driver.» Dep. de Physique, Université de Stockholm, Stockholm, 2007, 1-45.

Metropolis, N., et S. Ulam. «"The Monte Carlo Method.» *Journal of the American Statistical Association*, 1949: 335-341.

Miller, D.C. «The DOD High Level Architecture and the Next Generation of DIS.» *Proceedings of the 4th Workshop on the Standards for the Interoperability of Distributed Simulation.* Orlando, Floride, 1996.

Milojicic, D.S., et al. «Peer-to-Peer Computing.» HP Laboratories Palo Alto, Palo Alto, 2002, 53.

Mutka, M., et M. Livny. «The available capacity of a privately owned workstation environment.» *Perfomance Evaluation*, 1991: 269-284.

Nakajima, K. «Parallel iterative solvers for finite-element methods using an OpenMP/MPI hybrid programming model on the earth simulator.» *Parallel Computing*, 2005: 1048 - 1065.

Nisan, N., S. London, O. Regev, et N. Camiel. «Globally distributed computation over the internet-the popcorn project.» *Proceedings of the 18th International conference on Distributed Computing Systems.* Amsterdam, Netherlands, 1998. 592-60.

NIST. «Secure hash standard. Federal Information Processing Standard.» *FIPS-180-1.* 1995.

Oksendal, B. *Stochastic Differential Equations.* Berlin : Springer-verlag And Heidelberg Gmbh & Co. Kg, 2005.

Oram, A. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* Sebastopol, CA: O'Reilly & Associates, 2001.

Overeinde, B.J., L.O. Hertzberger, et P.M.A. Sloot. «Parallel Discrete Event Simulation.» Édité par W.J. Withagen. *Proceedings of the Third Workshop Computersystems, Faculty of Electrical Engineering.* Eindhoven, The Netherlands, 1991. 19-30.

Page, E.H., R.L. Moose, et S.P. Griffin. «Web-Based Simulation in Simjava using Remote Method Invocation.» *Proceedings of the 1997 Winter Simulation Conference.* Atlanta, GA, 1997. 468-474.

Panneton, F., P. L'Ecuyer, et M. Matsumoto. «Improved Long-Period Generators Based on Linear Recurrences Modulo 2.» *ACM Transactions on Mathematical Software*, 2006: 1-16.

Park, S.K., et K.W. Miller. «Random Number Generators: Good Ones are Hard to Find.» *Communications of the ACM*, 1988: 1192-1201.

Pawlikowski, K. «Do Not Trust All Simulation Studies of Telecommunication Networks.» *Proceeding of International Conference on Information Networking, ICOIN'03.* Jeju Island, Korea, 2003. 899-908.

—. «Towards Credible and Fast Quantitative Stochastic Simulation.» *Proceedings of International SCS Conference on Design Analysis and Simulation of Distributed Systems, DASD'03.* Orlando, Florida, 2003b.

Pawlikowski, K., et V. Yau. «AKAROA: a Package for Automatic Generation and Process Control of Parallel Stochastic Simulation.» *Australian Computer Science Communications*, 1993: 71-82.

Pawlikowski, K., H.D.J. Jeong, et J.S.R. Lee. «On Credibility of Simulation Studies of Telecommunication Networks.» *IEEE Communications Magazine*, 2002: 132-139.

Peacock, J.K., J.W. Wong, et E.G. Manning. «A distributed approach to queuing network simulation, (), pp. , .» *Proceedings of the Winter Simulation Conference.* San Diego, CA, 1979. 399-406.

—. «Distributed simulation using a network of processor.» *Computer Network*, 1979b: 44-56.

Pedroso, H., L.M. Silva, et J.G. Sil. «Web based metacomputing with JET.» *Concurrency: Practice and Experience*, 1997: 1169-1173.

Peschanski, F, et J.P. Briot. «Architectures de composants répartis.» Dans *Ingénierie des composants*, de M Oussalah, 247-280. Vuibert, 2005.

Peschanski, F., et Briot J.P. «Adaptations dynamiques et orthogonales de composants logiciels distribués.» *Technique et Science Informatiques*, 2004: 151-174.

Petty, M.D. «Comparing high level architecture data distribution management specifications 1.3 and 1516.» *Simulation Practice and Theory*, 2002: 95-119.

Press, W.H., B.P. Flannery, S.A. Teukolsky, et W.T. Weterling. *Numerical Recipes in C, First Edition.* Cambridge University Press, 1988.

Qi, J., et R.H. Huesman. «Effect of errors in the system matrix on maximum a posteriori image reconstruction.» *Phys. Med. Bio.*, 2005: 3297-3312.

RAND Corporation. *A Million Random Digits with 100,000 Normal Deviate.* 1955.

Reuillon, R., D.R.C Hill, C. Gouinaud, Z. El Bitar, V. Breton, et I. Buvat. «Monte Carlo Simulation With The GATE Software Using Grid Computing.» *Proceedings of NOuvelles TEchnologies de la REpartition (NOTERE).* Lyon, France, 2008b. 201-204.

Reuillon, R., D.R.C. Hill, Z. El Bitar, et V. Breton. «A Java Based Toolbox for the Distribution of Parallel Monte Carlo Simulations. Application to Nuclear Medicine Using the GATE Simulation Package.» *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications.* 2007. 122-136.

—. «Rigorous distribution of stochastic simulations using the DistMe toolkit.» *Transaction on Nuclear Science*, 2008a: 595 - 603.

Reuillon, R., et D.R.C. Hill. «DistMe: A Generic Toolkit for Stochastic Simulation Distribution.» *Proceedings of the European Simulation & Modelling Conference.* 2006. 127-133.

Reynolds, P. «A shared resource algorithm for distributed simulation.» *Proceedings of the 9th International Symposium on Computer Architecture.* Austin, Texas, 1982. 259-266.

Rogers, D.W.O., et A.F. Bielajew. *Monte Carlo techniques of electron and photon transport for radiation dosimetry.* Vol. 3, chez *The dosimetry of ionizing radiation*, édité par Kase K.R., Bjängard B.E. et F.H. Attix, 427-539. San Diego: Academic Press, 1990.

Rütti, M. *A Random Number Generator Test Suite for the C++ Standard. Diploma Thesis.* 2004.

Rütti, M., M. Troyer, et W.P. Petersen. *A Generic Random Number Generator Test Suite.* 2004. http://arxiv.org/pdf/math/0410385.pdf.

Ryu, K.D., et J. Hollingsworth. «Exploiting the fine grained idle periods in networks of workstations.» *IEEE Transaction on Parallel and Distributed Systems*, 2000: 683-698.

Saito, M., et M. Matsumoto. «SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator.» Dans *Monte Carlo and Quasi-Monte Carlo Methods*, 607-622. Springer, 2008.

Sarmenta, L.F.G., S. Hirano, et S.A. Ward. «Towards Bayanihan: Building an Extensible Framework for Volunteer Computing Using Java.» *Proceedings of ACM Workshop on Java for High-Performance Network Computing.* Palo Alto, 1998. 1015-1019.

Schmid, F., et N.B. Wilding. «Error in Monte Carlo simulations using shift register random number generators.» *Int. J. Mod. Physics C*, 1995: 781-787.

Schollmeier, R. «A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications.» *Proceedings of the First International Conference on Peer-to-Peer Computing.* 2001. 101-102.

Sen, S., B. Baudry, et H. Vangheluwe. «Domain-specific model editors with model completion.» *Proceeding of the Multi-paradigm Modelling Workshop at MoDELS 2007.* Nashville, TN, 2007.

Seo, C., S. Park, B. Kim, S. Cheon, et B.P. Zeigler. «Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment.» *Proceedings of the Advanced Simulation Technologies Conference.* 2004. 9-15.

Shamir, A. «On the generation of cryptographically strong pseudorandom sequences.» *ACM Trans. Comput. Syst.*, 1983: 38-44.

Shoch, J., et J. Hupp. «Computing Practices: The 'Worm' Programs - Early Experience with a Distributed Computation.» *Communication of the ACM*, 1982: 172-180.

Sipper, M. «Generating parallel random number generators by cellular programming.» *Intl. J. Modern Phys. C*, 1996: 181-190.

Sleke, W., A.L. Talapov, et L.N. Schur. «Cluster-flipping Monte Carlo algorithm and correlation in “good” random number generators.» *JETP Letter*, 1993: 665-668.

Sobol, I.M. «Uniformly distributed sequences with an additional uniform property.» *USSR Computational Mathematics and Mathematical Physics*, 1976: 236-242.

Soley, R., et OMG Staff Strategy Group. *Model Driven Architecture.* 2000.

Soto, J. «Randomness Testing of the Advanced Encryption Standard Finalist Candidates.» NIST, 2000.

Srinivasan, A., D.M. Ceperley, et M. Mascagni. «Random Number Generators for Parallel Applications.» *Monte Carlo Methods in Chemical Physics, Advances in Chemical Physics Series*, 1999: 13-36.

Srinivasan, A., M. Mascagni, et D.M. Ceperley. «Testing parallel Random Number Generators.» *Parallel Computing*, 2003: 69-94.

Stainforth, D.A. et al. «Uncertainty in the predictions of the climate response to rising levels of greenhouse gase.» *Nature*, 2005: 403-406.

Sterling, T., D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, et C.V. Packer. «BEOWULF: A Parallel Workstation for Scientific Computation.» *Proceedings of the 1995 International Conference on Parallel Processing (ICPP).* 1995. 11-14.

Stevens, R., P. Woodward, T. DeFanti, et C. Catlett. «From the I-WAY to the National Technology Grid.» *Communications of the ACM*, 1997: 50-60.

Stewart, I. *Dieu joue-t-il aux dés ? Les nouvelles mathématiques du chaos, deuxième édition.* Flammarion, 1998.

Stoica, I., et al. «Chord: a scalable peer-to-peer lookup protocol for internet applications.» *IEEE/ACM Trans. Netw.*, 2003: 17–32.

Stojanovski, T., et L. Kocarev. «Chaos-based random number generators-part I: analysis[cryptography].» *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on [see also Circuits and Systems I: Regular Papers, IEEE Transactions on]*, 2001.

Sunderam, V. «PVM: a framework for distributed computing.» *Concurrency: Practice and Experience*, 1990: 315–339.

Surridge, M., S. Taylor, D. De Roure, et E. Zaluska. «Experiences with GRIA — Industrial Applications on a Web Services Grid.» *Proceedings of the First International Conference on e-Science and Grid Computing.* 2005. 98-105.

Tan, C.J.K. «The PLFG parallel pseudo-random number generator.» *Future Generation Computer Systems*, 2002: 693-698.

Tippett, L. H. C. *Random sampling numbers.* Cambridge, UK, 1927.

Tomassini, M., M. Sipper, M. Zolla, et M. Perrenoud. «Generating high-quality random numbers in parallel by cellular automata.» *Future Generation Computer Systems*, 1999: 291-305.

Torguet, P. «VIPER : Un modèle de calcul réparti pour la gestion d'environnements virtuels.» Manuscipt de Thèse, Informatique, Université Paul Sabatier, Toulouse, 1998.

Traoré, M., et D.R.C. Hill. «The use of random number generation for stochastic distributed simulation: application to ecological modeling.» *Proceedings of the 13th European Simulation Symposium.* Marseille, France, 2001. 555-559.

Tsouloupas, G., et M. Dikaiakos. «GridBench: A Tool for Benchmarking Grids.» *Proceedings of the 4th International Workshop on Grid Computing.* Phoenix, Arizona, 2003. 60-67.

Tuecke, S., et al. «Open Grid Services Infrastructure (OGSI) Version 1.0, Global Grid Forum Draft Recommendation.» 2003, 86.

Tugend, T. «UCLA to be first station in nationwide computer network.» *UCLA Press Release*, 3 July 1969.

Urbán, P., X. Défago, et A. Schiper. «Neko: A single environment to simulate and prototype distributed algorithms.» *Journal of information science and engineering*, 2002: 981-997.

Vangheluwe, H., et J. de Lara. «Domain-Specic Modeling with AToM3.» *Proceeding of the 4th OOPSLA Workshop on Domain-Specific Modeling.* Vancouver, Canada, 2004.

Varga, A. «The omnet++ discrete event simulation system.» *Proceedings of the European Simulation Multiconference (ESM'2001).* Prague, Czech Republic, 2001. 319-324.

Vattulainen, I. «Framework for testing random numbers in parallel calculations.» *Physical review E*, 1999: 7200-7204.

Vattulainen, I., T. Ala-Nissila, et K. Kankaala. «Physical models as tests of randomness.» *Physical Review E*, 1995: 3205-3214.

Walker, D.W. «The design of a standard message passing interface for distributed memory concurrent computers.» *Parallel Computing*, 1994: 657-673.

Wang, L. «Implementation and performance evaluation of the parallel CORBA application on computational grids.» *Advances In Engeenering Software*, 2008: 211-218.

Wei, B., G. Fedak, et F. Cappello. «, Towards efficient data distribution on computational desktop grids with BitTorrent.» *Future Generation Computer Systems*, 2007: 983-989.

Wichmann, B.A., et I.D. Hill. «An efficient and portable pseudorandom number generator.» *Appl. Stat.*, 1982: 188-190.

Wolfram, S. *A New Kind of Science.* Wolfram Media, Inc, 2002.

Wolniewicz, P., et al. «Accessing Grid computing resources with g-Eclipse platform.» *Computational Methods in Science and Technologie*, 2007: 131-141.

Wu, P., et K. Huang. «Parallel use of multiplicative congruential random number generators.» *Computer Physics Communications*, 2006: 25–29.

Xie, Y., Y.M. Teo, W. Cai, et S.J. Turner. «Service provisioning for HLA-based distributed simulation on the grid.» *Principles of Advanced and Distributed Simulation*, 2005: 282-291.

Yao, A.C. «Theory and applications of trapdoor functions.» *Annual IEEE Symp. Found. Comp. Sci.* 1982. 80-91.

Zajac, K., A. Tirado-Ramos, Z. Zhao, et P. Sloot. «Grid Services for HLA-Based Distributed Simulation Frameworks.» *Lecture Notes in Computer Science: Grid Computing*, 2004: 116-124.

Zajac, K., M. Bubak, M. Malawski, et P.M.A. Sloot. «Towards a grid management system for HLA-based interactive simulations.» *Proceedings of Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications.* Delft, The Netherlands, 2003. 4–11.

Zeigler, B.P. «Theory of Modeling and Simulation.» John Wiley, 1976.

Zhang, Y., et al. «Grid-Aware Large Scale Distributed Simulation of Agent-Based Systems.» *Proceedings of the 2005 European Simulation Interoperability Workshop.* 2005.

Zhou, Z., et C. Whiteman. «Motions of a double pendulum.» 1996: 1177-1191.

Ziegler, B.P., T.G. Kim, et H. Praehofer. *Theory of Modeling and Simulation, 2nd Edition.* New York: Academic Press, 2000.

Annexes techniques

Durant ma thèse j'ai déployé des machines de grille gLite pour la grille EGEE (Enabling Grids for E-sciencE) au LIMOS (Laboratoire d'Informatique, de Modelisation et d Optimisation des Systemes). Lors de la mise en place de ces machines j'ai écrit une documentation technique afin d'expliquer comment installer des machines de grille. Cette partie expose la documentation pour l'installation d'une interface utilisateur. Cependant la technologie et les procédures d'installation pour gLite évoluent vite. Nous avons donc fondé l'initiative « *grid-workshop* » encadrée par l'association HealthGrid[61] afin de produire et de tenir à jour un site internet sur l'administration de machines de grille. Pour une documentation à jour, j'invite donc le lecteur à se rendre sur le site internet : http://trac.healthgrid.org/grid-workshop/.

I Etapes préliminaires

L'installation se fait à partir d'une machine de grille utilisant le système d'exploitation Scientific Linux 3.

[61] http://www.healthgrid.org/

Configurez votre système pour synchroniser son horloge avec celle d'un serveur NTP. Une liste des serveurs français est disponible sur le site : http://www.cru.fr/services/ntp/serveurs_francais

Configurez le fichier `/etc/ntp.conf` en ajoutant les lignes :

```
restrict <time_server_IP_address> mask 255.255.255.255 nomodify notrap noquery
server <time_server_name>
```

Vous pouvez ajouter plusieurs serveurs.

Editez le fichier `/etc/ntp/step-tickers` ajouter la liste des adresses des serveurs comme suit :

```
chronos.espci.fr
ntp.uhb.fr
```

Si vous avez un firewall noyau sur votre machine il faut autoriser les connections entrante sur le port NTP. Pour iptable ajoutez les lignes suivantes dans `/etc/sysconfig/iptables` :

```
-A INPUT -s NTP-serverIP-1 -p udp --dport 123 -j ACCEPT
-A INPUT -s NTP-serverIP-2 -p udp --dport 123 -j ACCEPT
```

Pour le firewall les règles sont prises en compte dans l'ordre donc faite attention aux clauses REJECT précédant les lignes ajoutées. Recharger le firewall :

```
# /etc/init.d/iptables restart
```

Activez le service ntp :

```
# ntpdate <your ntp server name>
# service ntpd start
# chkconfig ntpd on
```

Vous pouvez voir les informations sur le deamon ntpd en utilisant la commande suivante :

```
# ntpq -p
```

Téléchargez et installer apt :

```
# wget ftp://ftp.scientificlinux.org/linux/scientific/30x/i386/SL/RPMS/apt-XXX.i386.rpm
```

```
# rpm -ivh apt-XXX.i386.rpm
```

Créer les fichiers `lcg.list` et `lcg-ca.list` :

```
# echo 'rpm http://glitesoft.cern.ch/EGEE/gLite/APT/R3.0/ rhel30 externals Release3.0 updates' >/etc/apt/sources.list.d/lcg.list
# echo 'rpm http://linuxsoft.cern.ch/ LCG-CAs/current production' >/etc/apt/sources.list.d/lcg-ca.list
```

Installez yaim :

```
# apt-get update
# apt-get install glite-yaim
```

Pour les machines de type autre que UI, BDII ou WN, il est nécessaire de disposer d'un certificat serveur. La machine correspondant au certificat doit posséder une adresse publique, une entrée DNS valide. De plus, il est nécessaire de fournir lors de la demande du certificat serveur un alias mail pour l'administration (type admin@isima.fr).

La demande de certificat serveur s'effectue sur le site : https://igc.services.cnrs.fr/GRID-FR/. Pour ce faire, vous devez posséder un certificat personnel.

Une fois le certificat obtenue, copiez le dans `/etc/grid-security`. Le certificat est composé de deux fichiers. Le premier, la clef privée, doit être nommé hostkey.pem doit être accessible en lecture uniquement par le propriétaire (root). Le second, le certificat public doit être accessible en lecture à tous.

II Installation d'une User Interface (UI)

II.1 Rôle

L'UI est le point d'accès à la grille pour l'utilisateur. C'est la « prise » sur laquelle il se branche pour bénéficier de la puissance de calcul disponible. Elle comprend de nombreuses fonctions qui permette de soumettre des jobs, de surveiller leur état d'avancement, de manipuler des données, d'avoir des informations sur l'état de la grille…

II.2 Installation

Un fichier d'exemple de configuration de yaim se trouve sur votre système : `/opt/glite/yaim/examples/siteinfo/site-info.def`

Vous pouvez vous en servir comme base pour votre configuration. Remplir le fichier avec les informations correspondant à votre site. Vous devez pour cela de la documentation de `yaim` dont l'URL est fournie à la fin de ce document.

Je vous conseille de stoker votre fichier `site-info.def` dans le répertoire `/opt/glite/yaim/etc`.

Ce fichier est le fichier de configuration pour l'utilitaire yaim pour l'UI de l'ISIMA et la configuration des VO biomed et auvergrid :

```
MY_DOMAIN=univ-bpclermont.fr

RB_HOST=rbroker.mrs.grid.cnrs.fr
PX_HOST=myproxy.cern.ch
WMS_HOST=rbroker.mrs.grid.cnrs.fr
BDII_HOST=cclcgtopbdii01.in2p3.fr
MON_HOST=my-mon.cern.ch
REG_HOST=lcgic01.gridpp.rl.ac.uk

# Debug variable.  If set then  some function will print additional
# debugging information.
# Possible values: NONE, ABORT, ERROR, WARNING, INFO, DEBUG
YAIM_LOGGING_LEVEL=INFO

# VO-BOX - Set this if you are building a VO-BOX
VOBOX_PORT=1975

# Set this to "yes" your site provides an X509toKERBEROS Authentication
Server
# Only for sites with Experiment Software Area under AFS
GSSKLOG=no
GSSKLOG_SERVER=my-gssklog.$MY_DOMAIN

YAIM_VERSION=3.0.1-17

# Repository settings
```

```
LCG_REPOSITORY="'rpm  http://glitesoft.cern.ch/EGEE/gLite/APT/R3.0/  rhel30
externals Release3.0 updates'"
CA_REPOSITORY="rpm http://linuxsoft.cern.ch/ LCG-CAs/current production"
REPOSITORY_TYPE="apt" # or "yum"

# For the relocatable (tarball) distribution, ensure
# that INSTALL_ROOT is set correctly
INSTALL_ROOT=/opt

# You will probably want to change these too for the relocatable dist
OUTPUT_STORAGE=/tmp/jobOutput
JAVA_LOCATION="/usr/java/latest"

# Set this to '/dev/null' or some other dir if you want
# to turn off yaim installation of cron jobs
CRON_DIR=/etc/cron.d

# Set this to your prefered and firewall allowed port range
GLOBUS_TCP_PORT_RANGE="20000 25000"

# Site-wide settings
SITE_EMAIL=root@localhost
SITE_CRON_EMAIL=$SITE_EMAIL  # not yet used will appear in a later release
SITE_SUPPORT_EMAIL=$SITE_EMAIL
SITE_NAME=isima
SITE_LOC="Clermont-Ferrand, France"
SITE_LAT=45.47 # -90 to 90 degrees
SITE_LONG=3.05 # -180 to 180 degrees
SITE_WEB="http://www.isima.fr"
SITE_TIER="TIER 2"
#SITE_SUPPORT_SITE="my-bigger-site.their_domain"

# Space separated list of email addresse which will be written into
/root/.forward"
ROOT_EMAIL_FORWARD="reuillon@isima.fr"

# If you have a http proxy configure it on order to decrease the load on
the CA hosts
#SITE_HTTP_PROXY="myproxy.my.domain"

# Space separated list of supported VOs by your site
```

```
VOS="biomed auvergrid"
QUEUES=${VOS}

# For each queue define a _GROUP_ENABLE variable which is a list
# of VO names and VOMS FQANs
# Ex.: MYQUEUE_GROUP_ENABLE="ops atlas cms /VO=cms/GROUP=/cms/Susy"
# In DNS like VO names dots and dashes shoul be replaced with underscore:
# Ex.: MYQUEUE_GROUP_ENABLE="my.test-queue"
#      MY_TEST_QUEUE_GROUP_ENABLE="ops atlas"
# You have to explicitely add the sgm's and prd's FQANs in order to allow
them to
# submit to the queue.

AUVERGRID_GROUP_ENABLE="auvergrid"
BIOMED_GROUP_ENABLE="biomed"

VO_BIOMED_SW_DIR=$VO_SW_DIR/biomed
VO_BIOMED_DEFAULT_SE=$CLASSIC_HOST
VO_BIOMED_STORAGE_DIR=$CLASSIC_STORAGE_DIR/biomed
VO_BIOMED_VOMS_SERVERS="vomss://cclcgvomsli01.in2p3.fr:8443/voms/biomed?/bi
omed/"
VO_BIOMED_VOMSES="biomed     cclcgvomsli01.in2p3.fr     15000     /O=GRID-
FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr biomed"
#VO_BIOMED_RBS="rbroker.mrs.grid.cnrs.fr"

VO_AUVERGRID_SW_DIR=$VO_SW_DIR/auvergrid
VO_AUVERGRID_DEFAULT_SE=$CLASSIC_HOST
VO_AUVERGRID_STORAGE_DIR=$CLASSIC_STORAGE_DIR/auvergrid
VO_AUVERGRID_VOMS_SERVERS="vomss://cclcgvomsli01.in2p3.fr:8443/voms/auvergr
id?/auvergrid/"
VO_AUVERGRID_VOMSES="'auvergrid   cclcgvomsli01.in2p3.fr   15002   /O=GRID-
FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr auvergrid'"
```

Installer et configurer l'UI :

```
/opt/glite/yaim/bin/yaim -i -s site-info.def -m glite-UI
/opt/glite/yaim/bin/yaim -c -s site-info.def -n UI
```

III Installation d'un Berkely Database Information Index (BDII)

III.1 Rôle

Le BDII (Berkely Database Information Index) index les informations sur les ressources de la grille.

III.2 Installation

Ce fichier est le fichier de configuration pour l'utilitaire yaim pour le BDII de l'ISIMA et la configuration des VO biomed et auvergrid :

```
MY_DOMAIN=univ-bpclermont.fr



RB_HOST=rbroker.mrs.grid.cnrs.fr
PX_HOST=myproxy.cern.ch
WMS_HOST=rbroker.mrs.grid.cnrs.fr
BDII_HOST=limosBDII.$MY_DOMAIN
#BDII_HOST=cclcgtopbdii01.in2p3.fr
MON_HOST=my-mon.cern.ch
REG_HOST=lcgic01.gridpp.rl.ac.uk

YAIM_LOGGING_LEVEL=INFO

VOBOX_PORT=1975

GSSKLOG=no
GSSKLOG_SERVER=my-gssklog.$MY_DOMAIN

YAIM_VERSION=3.0.1-17

LCG_REPOSITORY="'rpm  http://glitesoft.cern.ch/EGEE/gLite/APT/R3.0/  rhel30
externals Release3.0 updates'"
CA_REPOSITORY="rpm http://linuxsoft.cern.ch/ LCG-CAs/current production"
REPOSITORY_TYPE="apt" # or "yum"

INSTALL_ROOT=/opt

OUTPUT_STORAGE=/tmp/jobOutput
JAVA_LOCATION="/usr/java/latest"
```

```
CRON_DIR=/etc/cron.d

GLOBUS_TCP_PORT_RANGE="20000 25000"

GRIDICE_SERVER_HOST=$MON_HOST

SITE_EMAIL=root@localhost
SITE_CRON_EMAIL=$SITE_EMAIL  # not yet used will appear in a later release
SITE_SUPPORT_EMAIL=$SITE_EMAIL
SITE_NAME=isima
SITE_LOC="Clermont-Ferrand, France"
SITE_LAT=45.47 # -90 to 90 degrees
SITE_LONG=3.05 # -180 to 180 degrees
SITE_WEB="http://www.isima.fr"
SITE_TIER="TIER 2"

ROOT_EMAIL_FORWARD="gridadmin@isima.fr"

BDII_SITE_TIMEOUT=120
BDII_RESOURCE_TIMEOUT=`expr "$BDII_SITE_TIMEOUT" - 5`
GIP_RESPONSE=`expr "$BDII_RESOURCE_TIMEOUT" - 5`
GIP_FRESHNESS=60
GIP_CACHE_TTL=300
GIP_TIMEOUT=150

BDII_HTTP_URL="http://grid-deployment.web.cern.ch/grid-deployment/gis/lcg2-
bdii/dteam/lcg2-all-sites.conf"

VOS="biomed auvergrid"
QUEUES=${VOS}

AUVERGRID_GROUP_ENABLE="auvergrid"
BIOMED_GROUP_ENABLE="biomed"

VO_BIOMED_SW_DIR=$VO_SW_DIR/biomed
VO_BIOMED_DEFAULT_SE=$CLASSIC_HOST
VO_BIOMED_STORAGE_DIR=$CLASSIC_STORAGE_DIR/biomed
VO_BIOMED_VOMS_SERVERS="vomss://cclcgvomsli01.in2p3.fr:8443/voms/biomed?/bi
omed/"
```

```
VO_BIOMED_VOMSES="biomed      cclcgvomsli01.in2p3.fr      15000      /O=GRID-
FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr biomed"

VO_AUVERGRID_SW_DIR=$VO_SW_DIR/auvergrid
VO_AUVERGRID_DEFAULT_SE=$CLASSIC_HOST
VO_AUVERGRID_STORAGE_DIR=$CLASSIC_STORAGE_DIR/auvergrid
VO_AUVERGRID_VOMS_SERVERS="vomss://cclcgvomsli01.in2p3.fr:8443/voms/auvergr
id?/auvergrid/"
VO_AUVERGRID_VOMSES="'auvergrid   cclcgvomsli01.in2p3.fr   15002   /O=GRID-
FR/C=FR/O=CNRS/OU=CC-LYON/CN=cclcgvomsli01.in2p3.fr auvergrid'"
```

Installer et configurer le BDII :

```
/opt/glite/yaim/bin/yaim -i -s site-info.def -m glite-BDII
/opt/glite/yaim/bin/yaim -c -s site-info.def -n BDII
```

Notez que ce fichier à été utilisé ensuite pour reconfigurer l'UI afin qu'elle utilise le BDII récemment installé.

Résumé

Contrairement aux modèles déterministes, le déroulement d'un modèle stochastique est conditionné par la réalisation de variables aléatoires. L'utilisation de hasard permet d'approcher un résultat le plus souvent incalculable de manière déterministe. En contrepartie, il est nécessaire d'estimer les paramètres des distributions associées aux quantités aléatoires en sortie du modèle stochastique. Ce calcul requiert l'exécution de multiples réplications indépendantes de la même expérience et de ce fait, d'une importante quantité de calcul.
Toutes les simulations stochastiques comportent par conception un aspect naturellement parallèle. Elles représentent ainsi une des applications phares pour l'utilisation d'environnements de calculs distribués permettant de partager de la puissance de calcul à l'échelle mondiale, appelée grille de calcul.
Bien que 50% des cycles des plus gros super-calculateurs de la planète soient consommés par des calculs stochastiques, les techniques de génération parallèle de nombres pseudo-aléatoires sont méconnues. Il existe de ce fait un risque bien réel de produire et de publier des résultats de simulations stochastiques erronés. Cette thèse présente l'état de l'art des méthodes pour la distribution des réplications de simulations stochastiques et contribue à leur développement. Elle propose ainsi des méthodes novatrices permettant d'assurer une traçabilité dans le processus complexe de distribution de simulations stochastiques. Elle expose enfin des applications dans les domaines de l'imagerie médicale nucléaire et des simulations environnementales totalisant plus de 70 années de calcul sur un ordinateur séquentiel.

Abstract

Contrary to deterministic models, the execution of stochastic models depends on the realization of random variables. The use of randomness allows to estimate a result often way too long to compute in a deterministic manner. As a counterpart, parameters of the distribution law of the output random variables should be estimated by computing multiple independent replications of the same experiment and therefore requires a significant computation time.
Every stochastic simulation comprise by conception a naturally parallel aspect. They should be considered as killer-applications for distributed computing environments allowing to share computing power at a world wide scale, called computing grids.
Whereas 50% of the processing cycle of the biggest super-computer on earth are used for stochastic simulation parallel pseudo-random number generation techniques are neglected. Therefore there is a real risk of producing and publishing erroneous results of stochastic simulations. This PhD presents the state of the art of methods for distributing the replications of stochastic simulations and contribute to their development. It proposes novel methods for insuring traceability in the complex process of distributing stochastic simulations. At the end, it exposes applications in the domains of medical imaging and environmental simulations totaling more than 70 years of computation on a sequential computer.