Travail remis à Guy Tremblay

INF7235-10 : Devoir 2
À remettre au plus tard le mercredi, 23 avril, 16h00
Ne pas mettre dans une enveloppe

Nom	
Prénom	
Code permanent	
Courriel	
Nom	
Prénom	
Code permanent	
Courriel	

Description du problème, des solutions choisies et justification, conclusion		10 pts
Qualité générale du code		10 pts
Bon choix d'approches et respect de ces approches		10 pts
Résultats expérimentaux et analyse		10 pts
Description de la stratégie de tests, bon fonctionnement et preuve de bon fonctionnement (dont qualité des tests)		10 pts
Forme et qualité du français		5 pts
(Bonus) Difficulté, originalité du problème choisi		5 pts
Total		55 pts

Note globale :	/ 10

Programmation parallèle haute performance

Travail pratique #2 - INF7235 (gr. 10)

Hiver 2014

Parallélisation en MPI/C d'une méthode Monté Carlo appliquée à un jeux de hasard

Auteurs :

Johann Dubois

Clément De Figueiredo

16 avril 2014

Table des matières

I Description du problème

"*Le terme méthode de Monte-Carlo, ou méthode Monte-Carlo, désigne toute méthode visant à calculer une valeur numérique en utilisant des procédés aléatoires, c'est-à-dire des techniques probabilistes.*" [1]
Dans le programme élaboré dans le cadre d'un travail de session pour le cours de programmation parallèle haute performance, la méthode de Monte Carlo est appliquée aux jeux de hasard. Plus précisément, c'est le jeu du Loto qui a été choisi afin d'y appliquer la méthode de Monte Carlo. En effet, l'objectif est de savoir quelles sont les probabilités qu'un nombre apparaisse plus qu'un autre. Dans ce but, plusieurs techniques de parallélisme ont été mise en place et leurs performances ont été mesurées afin de connaître quelle était la meilleure approche dans le contexte de la méthode de Monte-Carlo appliquée au jeu du Loto.

II Approches utilisées

II.1 Implémentation de Monte-Carlo

Le langage choisit afin d'utiliser la librairie **MPI** et le langage **C**. Ce langage a été choisi car la librairie **MPI** est compatible avec ce langage et nous disposions de connaissances suffisante en **C** afin d'implémenter cette solution de Monte-Carlo.

Dans un premier temps, MPI est initialisé avec **MPI_Init** puis les processus esclaves sont identifiés avec l'instruction **MPI_Comm_rank**. Ensuite, si le processus actuel est le processus maître alors divers opérations sont effectuées : séquentielle, parallèle statique et parallèle dynamique (sac de tâches). Chaque temps d'exécution des divers calculs sont stockés grâce à la variable **MPI_Wtime** et sont affichés à la fin d'exécution du programme.

II.2 Génération de nombres aléatoires

La génération du nombre aléatoire utilisé dans la cadre de la fonction de tirage au sort a été faite de façon « Thread Safe ». C'est-à-dire que cela prévient les problèmes d'exclusion mutuelle qui pourrait avoir lieu si 2 threads voudraient récupérer une valeur, car la méthode de génération aléatoire se base sur l'horloge pour proposer un nombre.

L'utilisation de nombres aléatoires en C se fait avec les fonctions *srand()* et *rand()*. La première prend un argument qui servira de graine pour la génération de nombre. L'idée est alors de fournir une valeur de temps couplé avec le numéro du thread pour que la valeur soit différente pour chaque processus.
Le code suivant nous donne alors une valeur aléatoire « Thread Safe » :
srand(time(NULL) ^ numProc);

II.3 Versions parallèles

II.3.1 Statique

La version parallèle statique correspond à une version parallèle à granularité fine avec association statique entre tâches et threads. Autant de tâches vont être créées qu'il y a de processeurs. Chaque processus fait effectuer le tirage et les résultats vont être stockés localement. Ensuite, l'instruction **MPI_Reduce** va avoir pour rôle d'effectuer une réduction et d'additionner (grâce à l'argument **MPI_SUM**) tous ces résultats entre eux, le résultat final étant stocké dans un tableau final. Pour finir, on affiche les données issues du tableau récapitulatif ainsi que le temps d'exécution total.

II.3.2 Dynamique

La version parallèle dynamique correspond à une version parallèle avec association dynamique entre tâches et threads de type « sac de tâches ». Cette version réalise la même opération que la version statique cependant quelques différences sont présentes. Un processus maître appelé « Coordinateur » a pour rôle de distribuer les tâches aux différents processus appelés « travailleurs ». Les « travailleurs » reçoivent les différentes tâches à effectuer avec l'instruction

III Résultats expérimentaux

III.1 Application de test

L'application doit être compilée et lancée avec au moins la version 4.4.7 de **gcc** (présent sur le cluster Turing de l'UQAM). Un makefile est également fourni afin d'exécuter les différentes versions du programme.

Les commandes disponibles dans le makefile sont les suivantes :

- **make compile** (comportement par défaut de l'instruction make) pour compiler le code source
- **make tests** afin de vérifier le bon fonctionnement du programme
- **make mesures** pour exécuter le programme et en mesurer les performances

Des variables d'exécution pour également être modifiées. La variable **I** correspond au nombre total d'itérations effectués par la fonction **tirage** ainsi que la variable **NP** qui indique le nombre de processeurs utilisé par le programme.

III.2 Conditions des tests

Pour réaliser différents tests, plusieurs valeurs différentes vont être testées dans un environnement identique afin que les mesures soient comparables. L'application étant dépendante de la génération de nombre aléatoire, c'est pourquoi chaque mesure a été effectuée cinq fois et la moyenne de ces cinq tests a été retenue comme valeur finale.

III.3 Résultats

Afin d'étudier les effets d'une implémentation parallèle, nous avions varié trois paramètres : le nombre d'itérations du tirage au sort, le nombre de threads et le nombre de tâches.
Dans les résultats présentés par la suite, *S* correspond à « séquentiel », *PS* à « parallèle statique » et enfin *PD* à « parallèle dynamique ».

III.3.1 Variation du nombre d'itérations

La méthode de Monte-Carlo consiste à réaliser un grand nombre de fois une opération pour que la probabilité qu'un certain nombre soit choisi soit élevée, dans le cas de notre application de la méthode au jeu de Loto. Nous allons donc faire varier la valeur correspondante aux itérations du calcul sur une échelle allant de dix à cents millions (Tab. 1 et Fig. 1).

TABLE 1 – Résultat de la variation de la répétition

	10	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
S	0,000	0,001	0,001	0,009	0,084	0,818	5,125	47,876
PS	0,003	0,001	0,001	0,003	0,021	0,206	1,973	20,189
PD	0,005	0,002	0,003	0,005	0,031	0,181	1,827	18,003

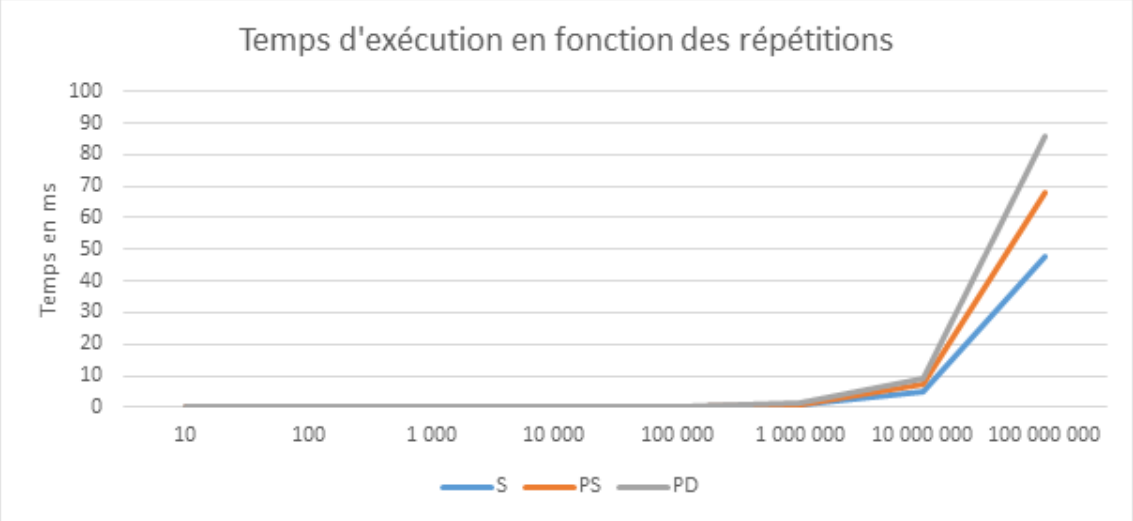

FIGURE 1 – Temps d'exécution en fonction des répétitions

On observe que le temps d'exécution est sensiblement le même entre les trois versions jusqu'à un million de répétitions (entre 0 et 10 millisecondes). Une fois ce seuil franchi, le temps d'exécution augmente plus rapidement en séquentiel qu'en parallèle. Pour ce qui est de l'accélération, les versions statique et dynamique sont presque pareilles à un détail près. L'accélération de la version dynamique va continuer à croitre jusqu'à un million d'itérations là où l'accélération de la version statique va commencer à décroitre vers cents mille itérations (Tab. 2 et Fig. 2).

TABLE 2 – Résultat de l'accélération avec la variation de la répétition

	10	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
PS	0,000	1,111	0,900	3,107	4,038	3,970	2,598	2,371
PD	0,000	0,476	0,346	1,611	2,727	4,511	2,805	2,659

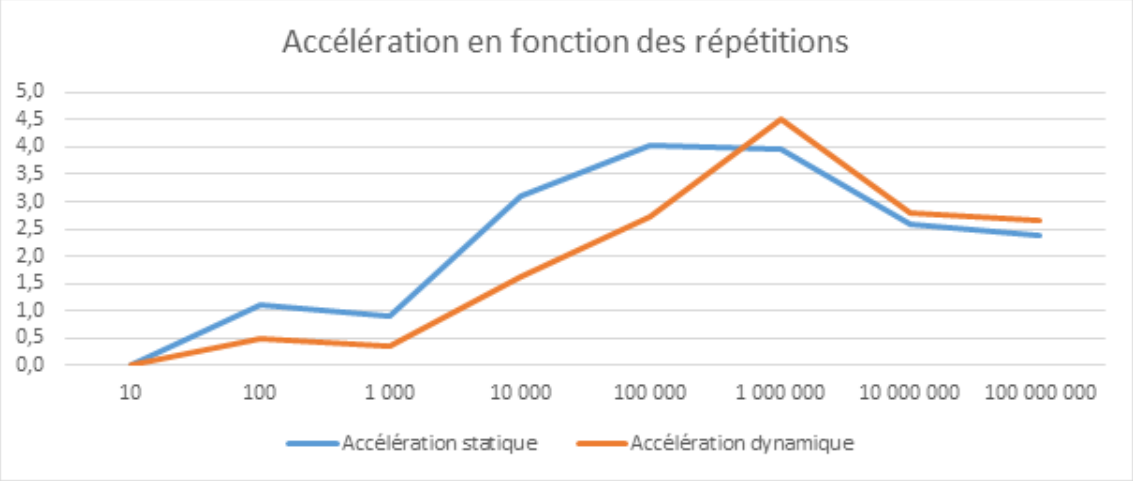

FIGURE 2 – Accélération en fonction des répétitions

III.3.2 Variation du nombre de processus

Ce deuxième test consiste à faire varier le nombre de processus sur lequel/-lesquels l'application va s'exécuter. Des valeurs vont être choisi allants de un à deux–cents avec des paliers de vingt-cinq. On peut voir que la méthode statique est légèrement plus rapide que la méthode dynamique. Au notera aussi qu'il n'y a aucun résultat pour la méthode dynamique lors de l'exécution du programme sur un processus. En effet, la mise en place d'une stratégie coordonnateur/travailleurs implique obligatoirement au moins 2 processus, ce qui explique le manque de résultat pour 1 processus (Tab. 3 et Fig. 3).

TABLE 3 – Résultat de la variation du nombre de processus

	1	**26**	**51**	**76**	**101**	**126**	**151**	**176**	**201**
S	0,079	0,083	0,086	0,089	0,093	0,095	0,180	0,181	0,200
PS	0,080	0,007	0,012	0,016	0,015	0,022	0,024	0,030	0,037
PD	-	0,020	0,039	0,063	0,054	0,029	0,054	0,049	0,087

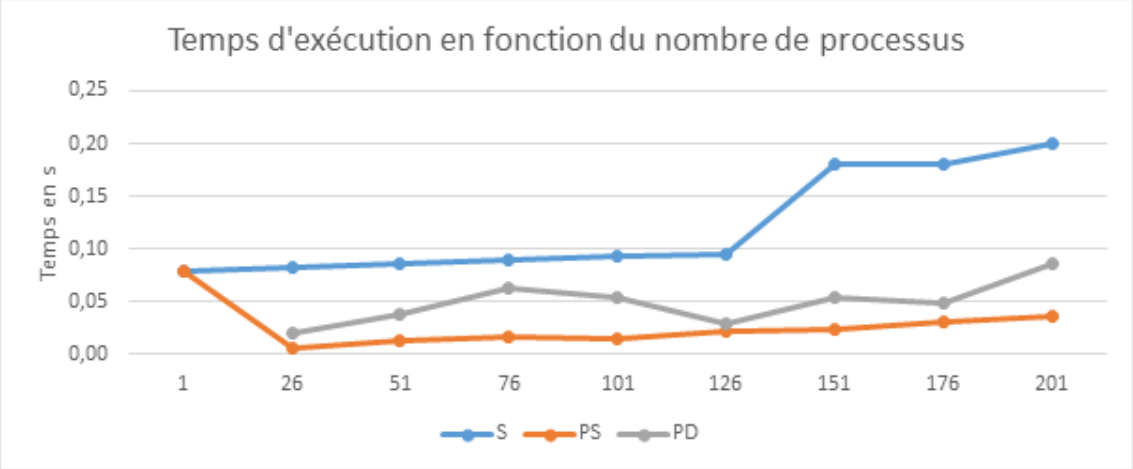

FIGURE 3 – Temps d'exécution en fonction du nombre de processus

La mesure d'accélération montre bien que la version statique est plus efficace que la version dynamique, surtout à partir de vingt-six processus. Cependant, les valeurs d'accélération ont tendances à se rapprocher vers cent vingt-six processus (Tab. 4 et Fig. 4).

TABLE 4 – Résultat de l'accélération avec la variation du nombre de processus

	1	**26**	**51**	**76**	**101**	**126**	**151**	**176**	**201**
PS	0,990	12,831	6,895	5,547	6,393	4,338	7,500	5,960	5,405
PD	-	4,212	2,204	1,422	1,733	3,276	3,333	3,686	2,299

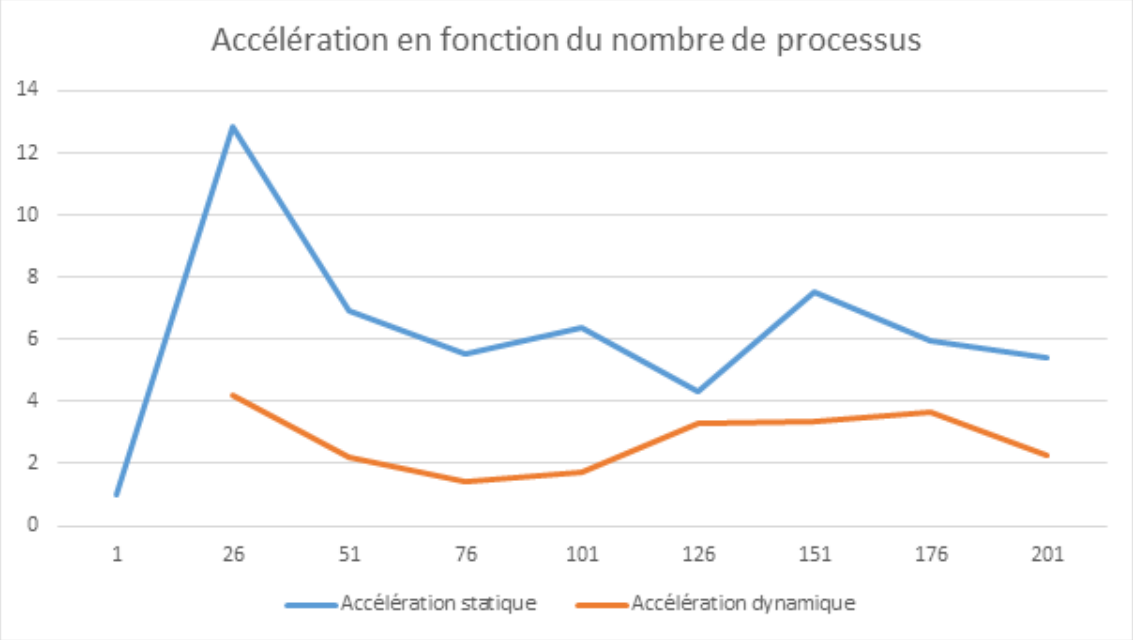

FIGURE 4 – Accélération en fonction du nombre de processus

III.3.3 Variation du nombre de tâches

Dans la version parallèle dynamique de notre implémentation de la méthode de Monte-Carlo, on divise le nombre de tâches en plusieurs groupes pour que chaque thread réalise une tâche après l'autre jusqu'à la fin de la queue. Chaque thread récupère une tâche lorsque sa tâche précédente est terminée, dans le cas contraire il attend une opération de communication afin de transmettre ces résultats au thread maître.

Nous allons donc tester différentes valeurs allant de dix tâches à cent millions de tâches afin d'observer les effets de la variation d'un petit ou grand nombre de tâche sur les différentes versions du programme.

Comme prévu, les résultats ne varient pas avec les méthodes séquentielle et statique car le nombre de tâche est fixe au contraire de la version dynamique. On peut donc voir que pour des valeurs inférieures à un million de tâches par processus la

version statique est plus rapide que la version dynamique pour atteindre un temps d'exécution constant de deux millisecondes à partir d'un million de tâches (Tab. 5 et Fig. 5).

TABLE 5 – Résultat de la variation du nombre de tâches

	10	**100**	**1000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
S	0,080	0,080	0,079	0,079	0,079	0,079	0,079	0,079
PS	0,021	0,021	0,021	0,021	0,022	0,021	0,023	0,025
PD	0,571	0,057	0,031	0,034	0,083	0,002	0,002	0,002

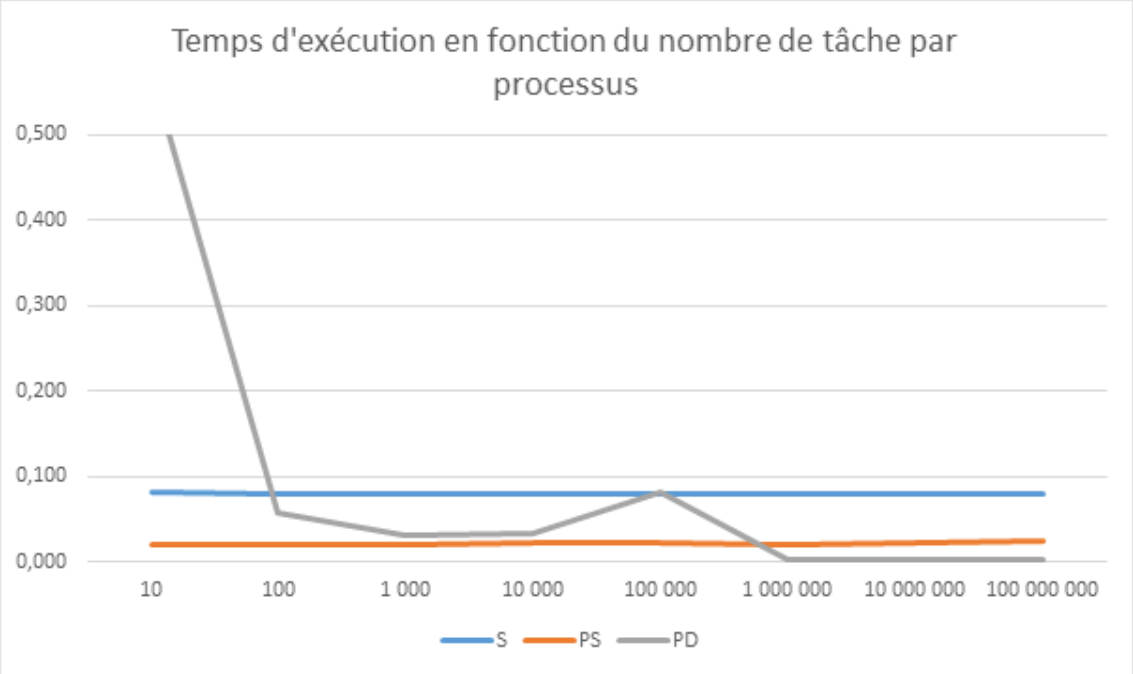

FIGURE 5 – Temps d'exécution en fonction du nombre de tâches

Afin de mieux observer les résultats, voici le calcul de l'accélération des deux méthodes parallèles. On voit clairement que la version dynamique est bien plus performante passé un million de tâche allant jusqu'à une accélération de trente-quatre fois supérieur à la version séquentielle (Tab. 6 et Fig. 6).

TABLE 6 – Résultat de l'accélération avec la variation du nombre de tâches

	10	**100**	**1 000**	**10000**	**100000**	**1000000**	**10000000**	**100000000**
PS	3,903	3,813	3,845	3,726	3,563	3,766	3,433	3,169
PD	0,141	1,389	2,555	2,324	0,959	34,217	34,478	37,571

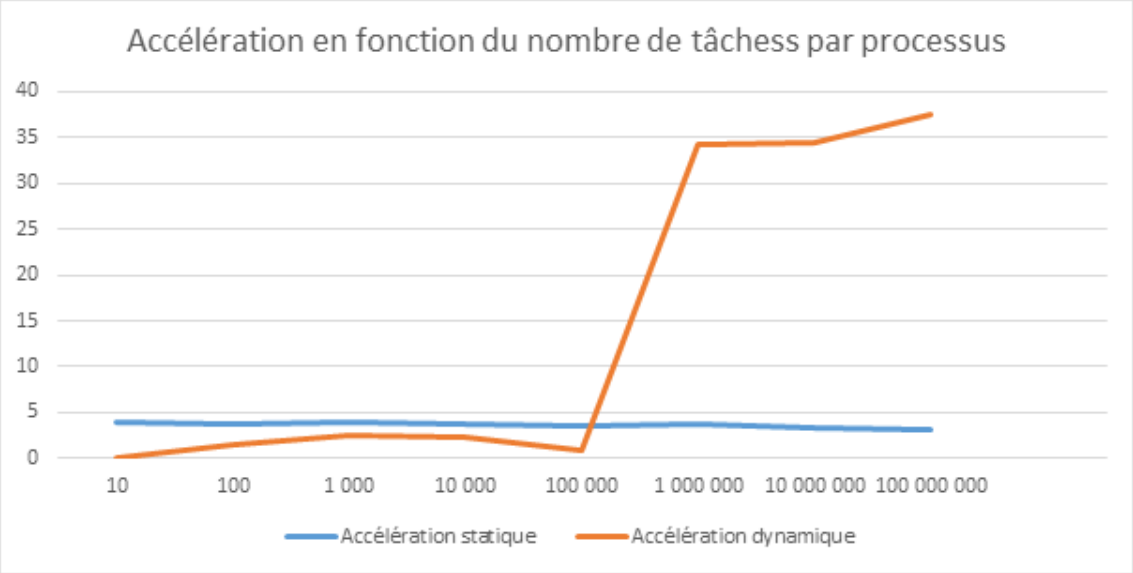

FIGURE 6 – Accélération en fonction du nombre de tâches

III.4 Analyse

IV Ouvertures

Références

[1] Wikipedia, méthode de monte-carlo. URL : http://fr.wikipedia.org/wiki/M%C3%A9thode_de_Monte-Carlo.