TP 4 (Suite) - Calcul de PI par une Méthode de Monte Carlo.

Implémentation pour environnement à mémoire distribué avec les Socket.

Dans ce TP nous souhaitons porter un code Monte Carlo (MC) pour le calcul de PI sur une architecture à mémoire distribuée.
Précédemment nous avons :

- Analysé l'algorithme MC et proposé deux variantes possibles suivant un modèle de programmation par tâche :
  - Parallélisme de boucle, une tâche par itération
  - Structure Maître Esclave, plusieurs itération par tâche
- Etudié deux implémentations en Java pour architecture à mémoire partagée :
  - Code 1 : Assignment 102

https://gist.github.com/krthkj/9c1868c1f69142c2952683ea91ca2a37

  - Code 2 : Pi.java

http://web.cs.iastate.edu/~smkautz/cs430s14/examples/thread_pool_examples/pi/Pi.java

Nous souhaitons maintenant porter l'algorithme pour des architecture à mémoire distribuée. Dans ce contexte, chaque tâche doit pouvoir communiquer avec une autre tâche par envoi de message car la mémoire n'est pas partagée.

Etant donnée les conclusions de l'analyse précédente, nous proposons une implémentation Maître Esclave. Cette implémentation a été proposée dans le code Pi.java en mémoire partagée.
Comment réaliser une telle implémentation en mémoire distribuée ?

D'après le cours, le paradigme Maître Esclave peut être vu comme l'opposé du paradigme Client Serveur. C'est à dire que le Maître est considéré comme un client, et les Esclaves sont considérés comme des serveurs.
Il faut donc :

- Créer plusieurs serveurs
- Créer un client
- Mettre en place l'envoi de message entre client et serveurs.

Pour cela nous allons nous intéresser au Socket en Java.

Exercice 1. : Java Socket

Téléchargez l'archive compressée distributedMC_step1.tar.gz.
Utilisez le code (Voir README et Figure 1.) puis détaillez sa conception.

Vous pourrez vous aider du cours en ligne OpenClassroom qui explique la mise en place des communications entre client et serveur via les Socket:
- les sockets côté client : https://openclassrooms.com/fr/courses/2654601-java-et-la-programmation-reseau/2668710-les-sockets-cote-client
- les sockets côté serveur : https://openclassrooms.com/fr/courses/2654601-java-et-la-programmation-reseau/2668874-les-sockets-cote-serveur

Ou du guide du site baeldung : https://www.baeldung.com/a-guide-to-java-sockets

Exercice 2. : Implémentation de MC pour un Worker

Faites évoluer le code pour pouvoir calculer PI.
On se posera les questions suivantes :

- Comment ajouter le calcul de PI par MC dans un worker ?
- Quel(s) message(s) le client doit envoyer ?
- Peut-on ré-utiliser Pi.java ? Si oui, dans un premier temps fixez numWorkers de Pi.java à 1. (1 thread)

Exercice 3. : Evaluation de votre implémentation

Evaluez la scalabilité forte de votre code pour un nombre total d'itération de
64 000 000 et/ou 16 000 000 ?
Remarque, chaque worker n'a qu'un thread.
Comparez cette scalabilité forte à celle du code à mémoire partagée.

Exercice 4. : Parallélisation à 2 niveau

Faites évoluer le code pour pouvoir calculer en exploitation deux niveaux d'architecture.

- Sur un premier niveau on considère une architecture à mémoire distribuée. Les Worker communique avec le Master par envoi de message.
  Le Master est un client qui envoie aux Worker la demande de calcul et reçoit des Worker le nombre de points dans la cible, et ce via les Socket.
- Sur un second Niveau on considère une architecture à mémoire partagée. Au sein d'un Worker on considère également une implémentation Maître Esclave. Le Worker est donc un Master et communique avec plusieurs worker, que l'on appeler sous-worker via la mémoire partagée.

Evaluez la scalabilité forte de votre code pour un nombre total d'itération de
64 000 000 et/ou 16 000 000 ?
Remarque, chaque worker peut avoir plusieurs sous-workers (thread).
Comparez cette scalabilité forte à celle des codes sur un niveau à mémoire partagée et à mémoire distribuée.

FIGURE 1. UTILISATION DU CODE POUR 1 MASTER ET 2 WORKERS