INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Département Informatique

Devoir surveillé
Programmation Parallèle
lundi 14 décembre 2020

3 pages, durée 1h30
Les 3 exercices sont indépendants.
Tous documents autorisés mais pas d'utilisation d'outils logiciels pour des tests de code ni d'accès à des sites Internet.
Pas de réponse aux questions pendant l'examen. En cas d'ambiguïté dans l'interprétation du sujet, explicitez ce que vous avez compris.
Le DS est sur 21 points.

Java (8 points)

Construire un système client-serveur qui illustre l'utilisation de Java RMI dans un cadre applicatif (on donnera seulement le code du serveur).
Le code doit mettre en évidence :

- l'utilisation du rmiregistry
- l'appel à distance dans différents cas :
    - uniquement en utilisant des types de base (paramètres et résultat)
    - avec un résultat rendu par le serveur qui soit une référence vers un de ses objets
    - avec un résultat rendu par le serveur qui soit une copie d'un de ses objets
    - au moins un cas d'utilisation d'une exception autre que RemoteException

Quelques idées de contexte applicatif (non limitatif): bibliothèque avec prêt de livres numériques, agence de location de voitures, prêt de matériel, location de salles, covoiturage... Pas d'exemples trop proches de sujets passés ou de TPs (chat, calculatrice, station météo,...).

Chacun de ces points doit être illustré séparément et sera évalué (noté) indépendamment des autres points. Les explications seront notées au même titre que le code. On pourra ne pas écrire les détails du code "métier" et les remplacer par des commentaires.
La cohérence du contexte applicatif avec l'utilisation du rmi sera également évalué (cad: pourquoi faire un appel par référence/un appel par valeur dans tel ou tel cas, etc.).

Flot de données (8points)

On vous demande de développer une application de traitement de flux de données dans le cadre du suivi du trafic routier dans une ville. Vous avez accès en temps réel aux signaux de positionnement des véhicules la traversant : régulièrement, chaque véhicule envoie un signal contenant sa position.

L'application doit être capable de compter, sur une période donnée, et pour chaque quartier de la ville, le nombre de véhicules le traversant. Si les signaux des véhicules arrivent en continu, le début d'une période (et la fin de la période précédente), et donc la mise à zéro des compteurs, se fait manuellement : un opérateur humain le déclenche.

L'écosystème logiciel à votre disposition inclut Storm. On vous indique que votre topologie de traitement devrait inclure les éléments suivants :

- Un spout s1 injectant les données de position
- Un spout s2 transmettant le signal de réinitialisation des compteurs
- Un bolt b1 transformant une position en le quartier correspondant
- Un bolt b2 comptant le nombre de véhicule pour chaque quartier

Tous les éléments ci-dessus ont un niveau de parallélisme strictement supérieur à 1 (à part s2.)

Q1 : Définir une topologie combinant ces éléments qui répondent à la spécification de l'application : donnez les liens de dépendances de données entre ces éléments en précisant les « groupings » que vous utiliserez. Le détail du fonctionnement interne des éléments n'est pas demandé.

Q2 : Quel algorithme de Storm peut-on utiliser pour s'assurer qu'un tuple a bien été traité de bout en bout (y compris les tuples issus de son traitement.) Illustrez ce mécanisme avec un tuple émis par le spout s1.

Q3 : On distingue dans Storm deux niveaux de parallélisme : celui voulu par l'utilisateur, et celui effectif obtenu à l'exécution. Comment le premier peut-il être spécifié ? De quoi dépend le second ?

Q4 : Comment peut-on facilement transformer un bolt stateful en un bolt stateless (sans en changer la fonctionnalité) ?

GPU (5 points)

Q1. Supposons qu'un kernel CUDA est lancé avec 1000 blocs de threads, chaque bloc ayant 512 threads.
(i) Si une variable est déclarée comme variable locale dans le kernel, combien de versions de la variable seront créées pendant la durée de vie de l'exécution du kernel ?
(ii) Si la variable est déclarée comme variable de mémoire partagée (shared memory), combien de versions de la variable seront créées ?

Q2. Comment les architectures GPU peuvent-elles tolérer de longues latences d'accès à la mémoire ?

Nous utilisons le kernel suivant pour ajouter deux vecteurs (le code est différent de l'exemple du cours). La taille d'un bloc de threads est de 512 threads.

```
__global__ void vecAddKernel(float *A_d, float *B_d, float *C_d, int n) {

  int i = blockIdx.x * blockDim.x * 2 + threadIdx.x;
  if (i < n)
    C_d[i] = A_d[i] + B_d[i];
  i += blockDim.x;
  if (i < n)
    C_d[i] = A_d[i] + B_d[i];
}
```

Q3. Si chaque vecteur a 20000 éléments et la taille d'un bloc de threads est de 512 threads, donnez le code d'invocation du kernel (1 instruction).

Q4. Nous exécutons notre programme d'addition de vecteurs et nous constatons qu'il est plus lent qu'un programme C séquentiel équivalent. Donnez une explication.