Master 1 Informatique
Systèmes Distribués et Parallélisme

Éric Leclercq

Révision : Décembre 2011

Résumé

Ce document contient l'ensemble des exercices de TP du module de Systèmes Distribués. Les exercices notés avec une étoile ne font pas l'objet d'un compte rendu détaillé, il s'agit seulement d'une prise en main des outils étudiés. Nous aborderons les différents paradigmes de la programmation parallèle et distribuée [1] que nous illustrerons dans différents langages de programmation (Java, C et C++, Scala, Erlang) et au moyen de bibliothèques (Threads POSIX, MPI, CUDA). Les exercices sont organisés slon 3 grandes parties selon les architectures physique des machines (machine à mémoire partagée machine à mémoire distribuée et hybride avec les GPGPU). Les modèles de programmation (parallèle ou distribuée) sont étudiés de manière découplée des architectures de machines.

Partie 1

Table des matières

1. souvent rassemblés, un peu rapidement, sous le terme de programmation concurrente

1 Notions élémentaires

1.1 Mesure et analyse de performance

Pour chaque exercice, sur une machine donnée vous devez exécuter un programme séquentiel et l'utiliser comme référence pour évaluer les différentes versions du programme parallèle. Vous devez ensuite être en mesure de tracer un courbe d'accélération et d'étudier l'influence de la taille des données traitées sur le comportement de votre programme.

1.2 Loi d'Amdhal

L'accélération $S(n)$ d'un programme donné, traitant un jeu de données fixé, est définie au moyen des variables suivantes :

- n représente le nombre de processeurs ou de cœurs
- t_s représente le temps d'une exécution séquentiel du programme
- $t_p(n)$ représente le temps d'une exécution parallèle du programme sur n processeurs
- s est la portion séquentielle du programme
- p est la portion du programme qui peut être exécutée en parallèle

Ainsi l'accélération est définie par :

$$S(n) = \frac{t_s}{t_p(n)} \tag{1}$$

Dans une version parallèle ou distribuée du programme, le temps séquentiel étant fixé et incompressible, on cherche à diminuer le temps parallèle, on obtient donc de nouvelles expression pour t_s et $t_p(n)$ avec l'hypothèse $s + p = 1$ en utilisant le temps séquentielle comme une référence :

$$t_s = s + p \tag{2}$$

p représentant la partie parallélisable, on obtient :

$$t_p(n) = s + \frac{p}{n} + \text{overhead} \tag{3}$$

l'overhead correspond aux temps de communication de synchronisation, d'ordonnancement etc. En général on néglige l'overhead. Soit remis dans (1) :

$$S(n) = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{(1-p)+\frac{p}{n}} \tag{4}$$

en utilisant un grand nombre de processeur, l'accélération limite est :

$$\lim_{n\to\infty} S(n) = \frac{1}{1-p} \tag{5}$$

Les valeurs remarquables pour $S(n)$ sont :

- $S(n) < 1$ perte de performance
- $1 < S(n) \leq n$ accélération normale
- $S(n) = n$ accélération linéaire (cas idéal)
- $S(n) > n$ accélération supra-linéaire (hyper-accélération) qu'il faut expliquer (utilisation de caches, etc.)

La loi d'Amdahl nous apprend qu'on n'obtient pas toujours une bonne accélération en parallélisant une application donnée dont une partie est inévitablement séquentielle. Si le programme contient (en temps d'exécution) la moitié de code non parallélisable, l'accélération maximum est de 2. Si le code est à 90 pourcent parallélisable l'accélération maximum est de 10.

2 Modèle mémoire partagée : threads, concurrence et scalability

Dans cette série d'exercice vous devez savoir créer des threads en Java, en C et gérer les problèmes de concurrence avec des verrous, sémaphores et/ou des moniteurs. Les programmes réalisés doivent être *scalable*[2] par conséquent, le nombre de threads doit, au moins, être fixé dans une constante ou mieux dans un fichier de configuration.

Indications

- Dans tous les exercices générer de jeux de tests dont le temps d'exécution dépasse les 20s ;
- Les commandes `time`, `top`, `ps` doivent être maîtrisées (fonctionnalités, utilisation des options, résultats produits) ;
- Pour l'exercice OpenMP vous pouvez commencer par lire le document à l'adresse suivante : `https://computing.llnl.gov/tutorials/openMP/`
- Pour le langage C, si vous avez des lacunes, consulter le documents : Introduction au langage C `http://www-clips.imag.fr/commun/bernard.cassagne`
- Compilation avec gcc avec les bibliothèque mathématiques, thread POSIX et avec l'affichage de tous les warnings : `gcc -Wall -o pgm pgm.c -lpthread -lm`
- Machines à disposition : une fois vos programmes testés sur votre station de développement vous pouvez les exécuter sur les machines `celine` et `tesla` ayant respectivement 8 cœurs, 16Go de RAM et 4 cœurs 4Go de RAM. Ainsi que sur les machines Solaris Opteron (butor, stendhal et genet) et Solaris SPARC T1 (eluard). Sur les machines SUN vous devez recompiler votre programme soit avec les compilateurs constructeurs (CC ou cc) soit avec les compilateurs GNU (g++ ou gcc).

Exercice 1. Threads et équilibrage de charge

1. On souhaite réaliser un programme pour énumérer les nombres premiers dans l'intervalle $[0, n]$. Expliquer pourquoi la méthode qui consiste à attribuer à chaque thread une portion de l'intervalle ne permet pas d'exploiter au mieux les performance de la machine.
2. Écrire en Java puis en C un programme pour énumérer les nombres premiers en utilisant un thread serveur et des threads clients multiples qui viennent chercher le nombre à tester auprès du thread serveur. Vous utiliserez la méthode qui consiste à tester les diviseurs du nombre à traiter i (jusqu'à $\sqrt(i)$) . Mesurer l'accélération par rapport au même algorithme exécuté en séquentiel au moyen d'un seul thread. Dresser une courbe en d'accélération en fonction du nombre de threads client lancés.
3. Discuter de l'adéquation du crible d'Eratosthène avec une programmation à base de thread sur machine à mémoire partagée.

Exercice 2. Multiplication de matrices

1. Écrire un programme de multiplication de matrices séquentiel puis en parallèle en utilisant plusieurs threads. Pour les jeux de test, générer des matrices de nombres donnant un résultat pouvant être anticipé, puis générer des tableaux de nombres aléatoires de grande taille. Mesurer le gain en performance du programme parallèle.

2. capacité d'un programme à pouvoir prendre en compte une charge importante et à la repartir sur plusieurs processeurs (souvent traduit par les termes de mise à l'échelle)

2. En utilisant le pattern *Bag of Task* (dont le principe reprend celui de la question 2 de l'exercice 1) implanter la multiplication de matrices. Montrer que ce *pattern* permet d'implanter des programmes *scalable* en fonction du nombre de processeurs mais qu'il n'offre pas forcement de bonnes de performance.
3. Ajuster les granularité des tâches pour améliorer les performances du *Bag of Task*. Mesurer l'évolution des performances en fonction‘

Exercice 3. OpenMP

1. Résumer les fonctionnalités d'OpenMP.
2. Implanter le programme de multiplication de matrice avec OpenMP et comparer l'accélération par rapport au programme écrit en 2.1
3. Discuter des avantages et inconvénients de l'approche OpenMP par rapport aux solutions étudiées dans les 2 exercices précédents.

3 Modèles mémoire distribuée

3.1 Interactions de type *Message Passing*

Dans plusieurs exercices vous devez implanter un algorithme, afin de valider votre solution il est nécessaire d'étudier le comportement séquentiel et parallèle des programmes. Comme dans la série d'exercices précédente, vous devrez comparer les temps des exécutions parallèles (avec différents nombres de nœuds) par rapport à l'exécution séquentielle de référence (sur un nœud du même type) . Vous devrez également étudier l'influence du nombre de nœuds et l'influence de la taille des données sur le comportement de votre programme.

Exercice 4. Prise en main de MPI(*)

Réaliser plusieurs petits programmes MPI afin de mettre en œuvre les fonctions suivantes de MPI (se référer au indications si dessous) :

1. Initialisation d'un ensemble de nœuds envoi et réception de messages en mode point à point (fonctions : `MPI_Init`, `MPI_Comm_size`, `MPI_Send`, `MPI_Recv`).
2. Synchronisation des nœuds par barrière (fonction `MPI_Barrier`).
3. Envoi de messages collectifs (fonction `MPI_Bcast`) ;
4. Distribution de données et collecte de résultats (fonctions : `MPI_Scatter`, `MPI_Gather`, `MPI_Allgather`, `MPI_Reduce`, `MPI_Allreduce`).

Exercice 5. Multiplication de matrices

1. Reprendre le programme de multiplication de matrices parallèle et l'implanter au moyen de MPI (répartition du calcul des lignes et/ou des colonnes). Ce programme est-il scalable ?
2. En utilisant le pattern *Bag of Task* implanter, en MPI, la multiplication de matrices. Montrer que ce pattern permet d'implanter des programmes scalable en fonction du nombre de processeurs mais présentes quelques limites (à identifier) en terme de performance.

Exercice 6. En hommage à B. Mandelbrot

L'ensemble de Mandelbrot est une fractale qui définit l'ensemble des points du plan complexe pour lesquels les suites S_c ne tendent pas vers l'infini :

$$S_c \begin{cases} z_{n+1} &= z_n^2 + c \\ z_0 &= 0 \end{cases}$$

Pour c donné, si au bout de n itérations, $|z| < 2$ alors la suite ainsi définie appartient à l'ensemble de Mandelbrot. Dans les autres cas, il est possible d'affecter une couleur au point du plan défini par c, fonction de la rapidité avec laquelle la suite diverge Afin d'explorer le comportement des suites S_c, on reformule l'expression sans utiliser les nombres complexes c'est-à-dire, en remplaçant z_n par le couple (x_n, y_n) et c par le couple (a, b), on obtient alors :

$$S_c \begin{cases} x_{n+1} &= {x_n}^2 - {y_n}^2 + a \\ y_{n+1} &= 2x_n y_n + b \\ z_0 &= 0 \end{cases}$$

1. Réaliser un programme séquentiel pour calculer l'ensemble de Mandelbrot pour c compris dans un disque de rayon 2, c'est-à-dire $(a, b) \in [-2, 2] \times [-2, 2]$.
2. Implanter votre programme en MPI.

Exercice 7. Tri Parallèle

1. Écrire un programme séquentiel pour implanter l'algorithme Quick Sort.
2. Quels caractéristiques le rendent difficile à paralléliser ?
3. Proposer plusieurs stratégies pour le paralléliser.
4. Modifier votre programme pour l'adapter à une architecture parallèle à mémoire distribuée exploitée au moyen de MPI.

Indications

- Tutorial MPI : `https://computing.llnl.gov/tutorials/mpi/`
- Commandes essentielles :
  1. `mpicc` compilateur pour les programmes C avec les bibliothèques MPI :
     `mpicc -o prog prog.c`
  2. `mpirun` lancement du programme principal et activation des nœuds participant au calcul au moyen de l'option `--hostfile` :
     `mpirun --prefix /usr/lib/openmpi -np 16 --hostfile A104 prog`
- Il existe plusieurs implémentation de MPI, les plus répendues sont OpenMPI (`http://www.open-mpi.org/`) et MPIch (`http://www.mcs.anl.gov/research/projects/mpi/mpich1/`)
- Installation sur les machines du département IEM :
  1. Installation pour tests en local sur la machine `celine` : OpenMPI
  2. Installation sur cluster de stations : 1 cluster par salle peut être défini au moyen d'un fichier machine et spécifié au lancement de mpi par l'option `--machinefile`. Il est préférable d'utiliser des machines homogène en terme de CPU et mémoire. Pensez à recompiler votre programme si vous changez d'architecture.

3. sur les machines Apple de la salle A105, OpenMPI est installé dans le répertoire `/opt/local`
   - si besoin, fixer la variable de la manière suivante :
     `export PATH=/opt/local/bin:$PATH`
   - pour compiler :
     `openmpicc -o prog prog.c`
   - pour lancer une exécution :
     `openmpirun -n 8 --prefix /opt/local/lib/openmpi -machinefile A105 prog`

– Utiliser des clés pour vous loger en ssh sans mot de passe sur l'ensemble du parc :

1. à partir de la machine de laquelle vous lancez l'exécution de MPI, générez les clés privée et publique : `ssh-keygen -tdsa`, ne rentrez pas de passphrase pour une utilisation dans le réseau IEM.
2. les clés sont générées dans le répertoire `.ssh/` de votre home directory, copiez la clé publique `id_dsa.pub` dans le fichier `$HOME/.ssh/authorized_keys` de chaque machine sur laquelle vous voulez vous connecter. Pour les machines de l'IEM les fichiers sont partagé par NFS :
   `cat $HOME/.ssh/id_dsa.pub >>$HOME/.ssh/authorized_keys`
   Attention lors de la première connexion il y a un échange de clés machines soumis à validation de l'utilisateur. Il convient donc de se connecter au moins une fois sur chaque machine (`ssh MI104-XY`) avant de lancer une exécution par `mpirun`.

– Pour l'exercice sur les fractales, si vous souhaitez afficher les fractales produites utiliser la librairie SDL. Une discussion sur la colorisation des point en fonction de leur `vitesse d'échappement` est disponible à l'URL : `http://fractals.iut.u-bordeaux1.fr/jpl/couleurs.html`

3.2 Objets distribués : Java RMI

Exercice 8. Prise en main de Java-RMI(*)

Le but de ce premier exercice (application directe du cours) est de prendre en main l'environnement Java-RMI et de se familiariser avec le processus de développement d'applications objets distribués. Réaliser le programme client serveur `HelloWolrd` vu en cours.

1. Tester l'ensemble sur une seule machine.
2. Observer la génération du stub et du squelette en utilisant la commande `rmic -keep`
3. Séparer le code du client et celui du serveur dans deux répertoires, quels sont les fichiers nécessaires à chacun pour la compilation et pour l'exécution ?
4. Lancer le serveur et le client sur deux machines différentes de même architecture.
5. Répéter l'opération précédente sur deux machines d'architecture différentes en prenant garde d'activer la même version de la machine virtuelle.
6. Peut-on utiliser 3 machines afin d'avoir un référentiel d'interfaces séparé, que constatez-vous ?

Exercice 9.

Réaliser un programme de simulation de gestion de compte bancaires. Vous devez obligatoirement adopter une approche objet et définir une classe `compte` permettant la

création d'un compte, proposant les opérations de dépôt, retrait, affichage du solde et permettant un archivage des opérations effectuées.

- Lancer plusieurs clients simultanément. Que se passe-t-il ? Comment est (doit-être) gérée la concurrence ?
- Comment réaliser une application qui gère 200 000 comptes ?

Exercice 10. Bag of Task

1. Implanter le pattern bag of task en Java RMI.
2. Expérimentez le en réalisant un produit de matrices.
3. Comment automatiser le lancement des clients.

Indications sur les JDK disponibles :

Plusieurs versions de J2SE (jdk) sont installées sur les serveurs, l'ensemble des serveurs SUN Solaris et sur les machines Linux. Les installations résident sous `/usr/local`. Afin de pouvoir utiliser la version choisie, vous devez positionner la variable d'environnement `PATH` et `JAVA_HOME` comme par exemple :

```
export PATH=/usr/local/jdk-1.4/bin:$PATH
export JAVA_HOME=/usr/local/jdk-1.4
```

Dans le cas contraire, c'est la machine virtuelle Java du système d'exploitation qui est utilisée et pas forcement la version de SUN (Oracle). Afin d'observer précisement le comportement de Java RMI, ne pas utiliser d'environnement de développement mais un simple éditeur comme vi, xemacs, nedit ou bluefish.

Le service `rmiregistry` écoute par défaut sur le port 1099. Si vous partagez une machine avec d'autres binômes, utiliser un autre port.

4 Modèle hybride : GPGPU

La technologie CUDA (*Compute Unified Device Architecture*) pour le développement de programmes parallèles sur GPGPU (*General-Purpose Computing on Graphics Processing Units*) utilise un ou plusieurs processeurs graphiques (GPU) pour exécuter des calculs habituellement exécutés par le processeur central (CPU).

4.1 Modèle

Le CPU utilise le ou les GPGPU comme un coprocesseurs scientifiques pour certains calculs adaptés aux architectures de type SIMD. Le CPU et le GPGPU sont tous les deux des unit de traitement multi-coeurs et sont chacune à une hierarchie de mémoires. Un GPGPU est un ensemble de n machines SIMD indépendantes partageant une mémoire globale.

Le modèle de programmation qui est proposé par l'architecture CUDA repose sur la notion de kernel, c'est-à-dire est une fonction C exécutée en parellèle n fois par n threads sur la carte GPGPU.

4.2 Qualifieurs

Un kernel est défini par le programmeur en utilisant la directive `__global__` devant le prototype de la fonction. La fonction est ensuite appelée au moyen de la séquence `<<< param >>>` après le nom de la fonction et avant les paramètres.

Chaque thread qui exécute un kernel obtient un identifiant (*threadID*) accessible par la variable `threadIdx`. La variable `threadIdx` est en fait un vecteur à 3 composantes qui permet de manipuler les threads par groupes ou bloc (*block*). Les threads d'un même bloc peuvent coopérer entre eux au moyen d'une mémoire partagée et synchroniser leur exécution au moyen de la fonction `_syncthreads()` qui agit comme une barrière à l'intérieur d'un bloc. Les autres abstractions offertes par le modèle de programmation sont : les groupes de threads, la mémoire partagée et les barrières de synchronisation.

Exercice 11. Prise en main(*)

1. Réaliser un programme qui effectue la somme ou le produit de deux vecteurs et affiche le résultat. Pour valider une première version de votre programme, ajouter par exemple le numéro de chaque thread qui effectue une addition au vecteur nul ou unité. Conclure sur le coût de création de threads dans le modèle CUDA ?
2. Réaliser un programme pour additionner deux matrices.

Exercice 12.

1. Reprendre le programme de test de nombres premiers et l'implanter en C CUDA. Choisir la stratégie la plus appropriée.
2. Reprendre le programme de multiplication de matrices en parallèle.

Pour les deux programmes comparer leur exécution avec les exécutions dans les autres modèles vus dans les sections précédentes, conclure (en utilisant votre connaissance sur l'architecture CUDA). Que pouvez vous dire d'une implémentation des fractales de Mandelbrot avec CUDA ?

Exercice 13. CUDA et OpenCL

1. Quels sont les technologies concurrentes de CUDA : en citer au moins deux et les décrire.
2. Réaliser l'un des programmes en OpenCL.
3. Dressez un comparatif entre le modèle CUDA et autres modèles de programmation étudiés. Pour ce faire identifier et définir au moins 4 critères de comparaison.

Indications.

Les programmes CUDA ne sont exécutables que sur la machine `tesla` au moyen du compilateur `nvcc`, positionner les variables d'environnement de la manière suivante :

```
export PATH=/usr/local/cuda/bin/:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64/:$LD_LIBRARY_PATH
```

La documentation est accessible à l'url suivante : `http://www.nvidia.com/object/cuda_develop.html`

Références

[1] Christophe Blaess. *Programmation système en C sous Linux Signaux, processus, threads, IPC et sockets*. Eyrolles, 2005.

[2] Clay Breshears. *The Art of Concurrency : A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly, 2009.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clid Stein. *Introduction to Algorithms*. MIT Press, 2009.

[4] Simon Thompson Francesco Cesarini. *Erlang Programming A Concurrent Approach to Software Development*. O'Reilly, 2009.

[5] Frank Thomson Leighton. *Introduction aux algorithmes et architectures parallèles : Grilles, arbres, hypercubes*. International. Thomsom Publicashing, 1995.

[6] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 2005.

[7] Peter Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann, 1996.

[8] Dean Wampler and Alex Payne. *Programming Scala*. O'Reilly, 2009.

[9] Barry Wilkinson and Michael Allen. *Parallel Programming : Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, 2004.

`https://moodle.polymtl.ca/course/view.php?id=999`