Informatique parallèle
Threads en JAVA

Erick STATTNER

Maître de Conférences en Informatique

Université des Antilles

erick.stattner@univ-antilles.fr

www.erickstattner.com

Description de l'enseignement

Objectifs pédagogiques:

- Comprendre les principes de parallélisation
- Maitriser les Threads en JAVA
- Transformer un algo séquentiel en algo parallèle
- Synchroniser les processus

Organisation:

- 36h
  - 1 Partie sur la parallélisation
  - 1 Partie sur la distribution
- 1 projet

Sommaire

Chapitre I.
Threads en JAVA

I. Introduction à la parallélisation

Contexte

- De nombreux programmes s'exécutent sur une machine
- Temps de calculs influencé
  - Lorsque de nombreux programmes tournent en même temps
  - Lorsqu'un même programme génère top de calculs
- Conséquence: augmentation des délais de traitements

Pour améliorer les temps de calculs

- Processeurs permettent d'exécuter plusieurs tâches/processus simultanément
- Nombre de cœurs augmenté sur un même processeur
- Cœurs sont "**multi/*hyperthreadés***" pour démultiplier la puissance de calculs

I. Introduction à la parallélisation

Processeurs

- **Cœur**
  - Unité de calculs physiquement présente sur le processeur
- **Threads**
  - Séquence d'instructions qu'un cœur va exécuter
  - Avec le ***multi-threading*** le processeur prend en charge (*en général*) 2 threads à la fois
  - On pale de ***cœurs logiques***
  - ***Multi-threading*** n'est pas présent sur tous les processeurs

De nombreux autres paramètres influencent les performances: fréquence, cache, socket, etc.

I. Introduction à la parallélisation

- Exemple de processeurs

Modèle	Cœurs	Threads	Fréquence	Turbo Boost note 1	Cache L1	Cache L2	Cache L3	Mult.	Tension	Révision	TDP	bus	Socket	Référence	Commercialisation Début	Commercialisation Fin
Core i7 900 Extreme Edition																
990X	6	12	3,46 GHz	3,60 (1) - 3,60 (1) - 3,60 (1) - 3,60 (1) - 3,73 (2) - 3,73 (2)	6 × 64 Kio	6 × 256 Kio	12 Mio	26		D0	130 W	QPI 6,4 GT/s + 3×DDR3 1,333 GT/s	LGA 1366		01 jan 2011	
980X	6	12	3,33 GHz	3,46 (1) - 3,46 (1) - 3,46 (1) - 3,46 (1) - 3,60 (2) - 3,60 (2)	6 × 64 Kio	6 × 256 Kio	12 Mio	25		D0	130 W	QPI 6,4 GT/s + 3×DDR3 1,333 GT/s	LGA 1366		11 mars 2010	
Core i7 900																
970	6	12	3,20 GHz	3,33 (1) - 3,33 (1) - 3,33 (1) - 3,33 (1) - 3,46 (2) - 3,46 (2)	6 × 64 Kio	6 × 256 Kio	12 Mio	24			130 W	QPI 4,8 GT/s + 3×DDR3 1,333 GT/s	LGA 1366		juillet 2010 [10]	
Core i7 800																
880	4	8	3,06 GHz	3,33 (2) - 3,33 (2) - 3,60 (4) - 3,73 (5)	4 × 64 Kio	4 × 256 Kio	8 Mio	23		B1	95 W	DMI 2,5 GT/s + 2×DDR3 + 16×PCI-express 2.0	LGA 1156			
870 [archive]	4	8	2,93 GHz	3,20 (2) - 3,20 (2) - 3,46 (4) - 3,60 (5)	4 × 64 Kio	4 × 256 Kio	8 Mio	22		B1	95 W		LGA 1156	BV80605001905AI	9 septembre 2009	
860 [archive]	4	8	2,80 GHz	2,93 (1) - 2,93 (1) - 3,33 (4) - 3,46 (5)	4 × 64 Kio	4 × 256 Kio	8 Mio	21		B1	95 W		LGA 1156	BV80605001908AK	9 septembre 2009	1er août 2010
Core i7 800S																
870S	4	8	2,66 GHz	2,80 (1) - 2,80 (1) - 3,20 (4) - 3,20 (4)	4 × 64 Kio	4 × 256 Kio	8 Mio	20		B1	82 W	DMI 2,5 GT/s + 2×DDR3 + 16×PCI-express 2.0	LGA 1156			
860S	4	8	2,53 GHz	2,53 (0) - 2,53 (0) - 3,33 (6) - 3,46 (7)	4 × 64 Kio	4 × 256 Kio	8 Mio	19		B1	82 W		LGA 1156		1er trim. 2010 [12]	

Modèle	Cœurs (Thread)	Fréquence	Turbo Boost note 6	Cache L1	Cache L2	Cache L3	Mult.	Tension	Révision	TDP	bus	Socket	Référence	Commercialisation Début	Commercialisation Fin
Core i5 500M															
580M	2 (4)	2,66 GHz	2,93 (2) - 3,33 (5)	2 × 64 Kio	2 × 256 Kio	3 Mio	20			35 W	DMI 2,5 GT/s + FDI + 2×DDR3 + 16×PCI-express 2.0				
560M	2 (4)	2,66 GHz	2,93 (2) - 3,20 (4)	2 × 64 Kio	2 × 256 Kio	3 Mio	20			35 W					
540M	2 (4)	2,53 GHz	2,80 (2) - 3,06 (4)	2 × 64 Kio	2 × 256 Kio	3 Mio	19			35 W				4 janvier 2010	
520M	2 (4)	2,40 GHz	2,66 (2) - 2,93 (4)	2 × 64 Kio	2 × 256 Kio	3 Mio	18			35 W				4 janvier 2010	
Core i5 400M															
460M	2 (4)	2,53 GHz	2,80 (2) - 2,80 (2)	2 × 64 Kio	2 × 256 Kio	3 Mio	19			35 W	DMI 2,5 GT/s + FDI + 2×DDR3 + 16×PCI-express 2.0			4 janvier 2010	
450M	2 (4)	2,40 GHz	2,66 (2) - 2,66 (2)	2 × 64 Kio	2 × 256 Kio	3 Mio	18			35 W				4 janvier 2010	
430M	2 (4)	2,26 GHz	2,53 (2) - 2,53 (2)	2 × 64 Kio	2 × 256 Kio	3 Mio	17			35 W				4 janvier 2010	
Core i5 500UM															
560UM	2 (4)	1,33 GHz	1,86 (4) - 2,13 (6)	2 × 64 Kio	2 × 256 Kio	3 Mio	10			18 W	DMI 2,5 GT/s + FDI + 2×DDR3 + 16×PCI-express 2.0			4 janvier 2010	
540UM	2 (4)	1,20 GHz	1,73 (4) - 2,00 (6)	2 × 64 Kio	2 × 256 Kio	3 Mio	9			18 W					
520UM	2 (4)	1,06 GHz	1,60 (4) - 1,86 (6)	2 × 64 Kio	2 × 256 Kio	3 Mio	8			18 W					
Core i5 400UM															
470UM	2 (4)	1,33 GHz	1,87	2 × 64 Kio	2 × 256 Kio	3 Mio	?			18 W	DMI 2,5 GT/s + FDI + 2×DDR3 + 16×PCI-express 2.0				
430UM	2 (4)	1,20 GHz	1,46 (2) - 1,73 (4)	2 × 64 Kio	2 × 256 Kio	3 Mio	9			18 W					

I. Introduction à la parallélisation

Exercice 1:

Identifier sur votre machine:

- le nombre de cœurs
- le nombre de processeurs logiques

I. Introduction à la parallélisation

Votre expérience de la programmation

- Programme pensé et exécuté de façon séquentielle
- N'exploite pas toutes les possibilités de calculs de la machine: ***cœurs et threads***
- Temps de calculs importants pour les gros traitements

Solution: Parallélisation

- Transformer un code source écrit pour une machine séquentielle en un exécutable parallélisé pour ordinateur multi-processeurs
- Exploiter toute la capacité du processeur
- Réduire les temps d'exécution en éclatant les calculs sur plusieurs coeurs

I. Introduction à la parallélisation

Etapes de la parallélisation

1. Analyser le problème
2. Identifier les étapes indépendantes
3. Identifier les synchronisations nécessaires pour récupérer les données de chaque sous-processus
4. Implémenter la solution parallèle

Exemple

- Programme permettant de générer une liste contenant tous les entiers naturels de 1 à N

Tous les programmes ne sont pas parallélisables !

I. Introduction à la parallélisation

Exercice 2:

Les programmes suivants sont-ils parallélisables ?

1. Programme permettant de calculer le terme n de la suite Un+2 = Un+1 + Un

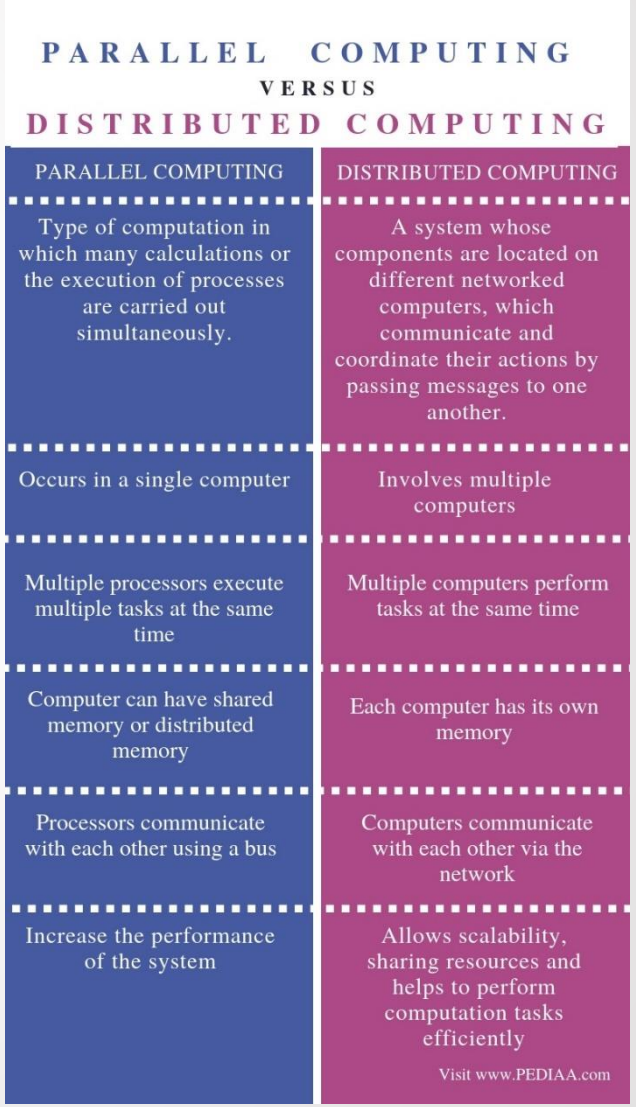
2. Programme permettant de compter le nombre d'occurrences d'une lettre c dans une chaine de caractères

I. Introduction à la parallélisation

Parallélisation VS Distribution

- **Parallélisation**:
  Permet à une unique machine d'exécuter plusieurs tâches en parallèle

- **Distribution**:
  Permet à un ensemble de machines de communiquer pour réaliser
  des taches



2. Threads en JAVA

Définition

- Un thread est une unité d'exécution faisant partie d'un programme
- Fonctionne de façon autonome et parallèlement à d'autres threads

Avantages

- Paralléliser les traitements d'un même programme
- Disposer d'une mémoire partagée
- Contrôle de l'exécution du Thread depuis le programme
- Gestion de bas niveau géré par la JVM

2. Threads en JAVA

Création d'un Thread JAVA

- La classe
  - doit hériter de la classe ***Thread***
  - Doit redéfinir la méthode ***run()***
    qui contient le code qui sera exécuté dans le thread.
  - Peut être alimenté par des attributs ou des méthodes

```
public class AfficheNFois extends Thread {
        private int nb;
        private String message;

        public AfficheNFois (int n, String m) {
                nb = n;
                message = m;
        }

        public void run() {
                for(int i = 0; i < nb; i++) {
                        System.out.println(message);
                }
        }
}
```

2. Threads en JAVA

Lancement d'un Thread JAVA

- L'objet
  - Peut être crée depuis n'importe quel endroit du programme
  - L'appel au constructeur ne lance par le thread selon l'objet est crée
  - Le lancement de l'exécution du thread avec la méthode **start()**

```
public class Main {
        public static void main(String args[]) {
                AfficheNFois a1 = new AfficheNFois (10, "Bonjour" );
                a1.start();
        }
}
```

2. Threads en JAVA

Exercice 3:
Qu'affiche le programme suivant:

```
public class Main {
		public static void main(String args[]) {

				AfficheNFois a1 = new AfficheNFois (10, "Bonjour" );
				AfficheNFois a2 = new AfficheNFois (10, "Toto" );


				a1.start();
				a2.start();
		}
}
```

2. Threads en JAVA

Mise en attente d'un thread

- On peut vouloir que le Thread se mette en pause un instant
- Utiliser la fonction ***sleep(int temps)***
  - Prend en paramètre un temps en millisecond
  - Peut lever une **InterruptExcepion**
- Le thread est mis sommeil
- Les autres threads s'exécutent normalement

2. Threads en JAVA

Mise en attente d'un thread

```
public class AfficheNFois extends Thread {
        private int nb;
        private String message;

        public AfficheNFois (int n, String m) {
                nb = n;
                message = m;
        }

        public void run() {
                try {
                        for(int i = 0; i < nb; i++) {
                                System.out.println(message);
                                sleep(1000);
                        }
                }
                catch(InterruptExcepion e) {}
        }
}
```

2. Threads en JAVA

Exercice 4:
Qu'affiche le programme suivant après ajout de la fonction sleep

```
public class Main {
		public static void main(String args[]) {

				AfficheNFois a1 = new AfficheNFois (10, "Bonjour" );
				AfficheNFois a2 = new AfficheNFois (10, "Toto" );


				a1.start();
				a2.start();
		}
}
```

2. Threads en JAVA

En JAVA

- Un programme comporte toujours un Thread principal
  la méthode ***main***
- Lors de l'appel à la fonction ***sleep***, la machine virtuelle met le Thread en attente
  et donne la main à un autre thread (y compris le main).
- La méthode ***start*** ne peut être appelée qu'une seule fois
  dans le cas contraire, une exception est levée
- La méthode ***sleep*** doit être utilisée à bon escient

3. Interruption d'un Thread

Attendre la fin de l'exécution d'un thread

- Les threads s'exécutent en parallèle
- Le main s'exécute en parallèle comme les autres
  - Le main peut se terminer avant les autres threads
- Nécessité d'attendre la fin d'un Thread
- Utilisation de la fonction **join()**
  - Attend que le thread sur lequel la méthode est appelée se termine
- Exemple:

```
public class Main {
        public static void main(String args[]) {

                AfficheNFois a1 = new AfficheNFois (10, "Bonjour" );
                a1.start();
                a1.join();
                System.out.println("JE SUIS ICI" );

        }
}
```

3. Interruption d'un Thread

Arrêt d'un thread

- Un Thread s'arrête
  - Fin des traitements
  - ***ThreadDeath*** Exception est levé
- Un thread peut être ***stoppé*** volontairement
- Deux approches
  - Utilisation des méthodes d'interruption

Type	Method
void	***interrupt()*** (Met fin à l'exécution du thread)
void	***stop()*** (*Déprécié* – Met fin à l'exécution du thread)

  - Mise en place de stratégies d'interruption

3. Interruption d'un Thread

Exemple d'interruption avec stop() / interrupt()

```
public class Main {
        public static void main(String args[]) {

                AfficheNFois a1 = new AfficheNFois (10, "Bonjour" );

                a1.start();
                a1.stop();
                System.out.println("JE SUIS ICI");

        }
}
```

3. Interruption d'un Thread

Exemple d'interruption d'un thread via stratégie d'arrêt

```
public class AfficheTantQue extends Thread {
        private String message;
        private boolean arret;

        public AfficheNFois (String m) {
                message = m;
                arret = false;
        }

        public void run() {
                try {
                        while( !arret ) {
                                System.out.println(message);
                                sleep(1000);
                        }
                }
                catch(InterruptExcepion e) {}
        }

        public void arreter() {
                arret = true;
        }
}
```

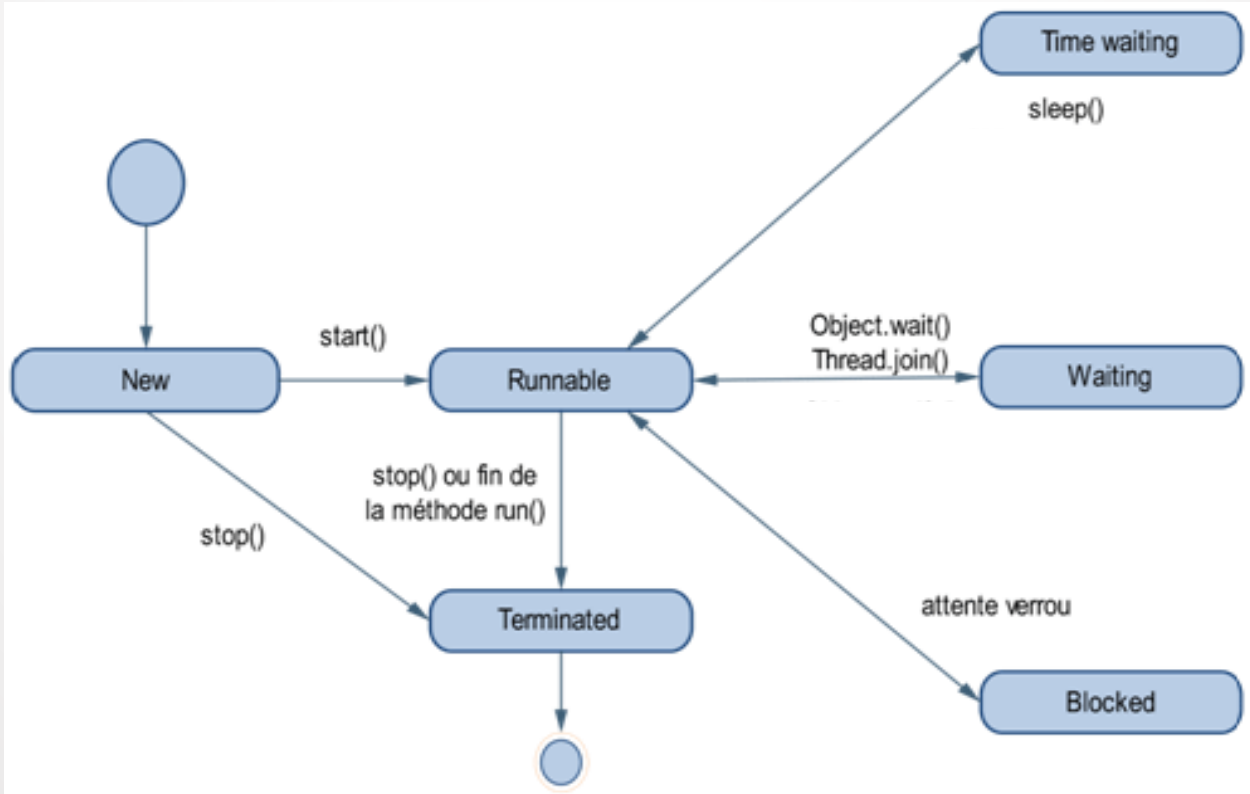
3. Interruption d'un Thread

Exemple d'interruption d'un thread via stratégie d'arrêt

```
public class Main {
        public static void main(String args[]) {

                AfficheTantQue a1 = new AfficheTantQue ("Bonjour" );

                a1.start();

                System.out.println("JE SUIS ICI");

                a1.arreter();

        }
}
```

3. Interruption d'un Thread

Cycle de vie des Threads



3. Interruption d'un Thread

Etat d'un Thread

Méthode	Rôle
boolean isAlive()	Renvoyer un booléen qui précise si le thread est en cours d'exécution. Elle renvoie true tant que le thread a été démarré et qu'il n'est pas arrêté
Thread.State getState()	Renvoyer le statut du thread
boolean isInterrupted()	Renvoyer un booléen qui précise si le thread est interrompu

4. Synchronisation de Threads

Avantages des Threads

- Partage la mémoire
- Il peuvent modifier un même objet

Inconvénients

- Parfois, il faut éviter que plusieurs Threads accèdent en même temps au même objet
- Il faut attendre qu'un Thread termine un traitement avant que l'autre ne poursuive son execution

4. Synchronisation de Threads

Méthodes synchronisées

- Appartient à une classe quelconque,
  pas nécessairement à un Thread
- A un instant donné, une seule méthode synchronisée peut accéder à l'objet
- Le verrou est attribué à la méthode synchronisée et libérée à la fin
- Aucune autre méthode synchronisée ne peut le recevoir
- Les méthodes non-synchronisée s'exécutent normalement
- **L'utilisation du mot clé synchronized sur plusieurs méthodes d'une même classe peut dégrader les performances**
- **Il est préférable d'utiliser le mot clé synchronized sur les portions de code critiques**

4. Synchronisation de Threads

Méthode synchronisée

- L'objet est verrouillé pour toute la durée de la méthode

```
public synchronized void maMethode() {
        …
}
```

4. Synchronisation de Threads

Exemple du tableau

- Soit le code suivant exécuté par 2 Threads T1 et T2

```
class ListeTab {
	private String[] tab = new String[50];
	private int index = 0;

	public void ajoute(String s) {
		tab[index] = s;
		index++;
	}
}
```

Exécutions possibles

- *(a1) (a2) (b1) (b2)*, est une exécution possible, cohérente
- *(b1) (b2) (a1) (a2)*, est une exécution possible, cohérente
- *(a1) (b1) (b2) (a2)*, est une exécution possible, mais incohérente : le tableau ne contient pas la chaîne de caractères ajoutée par *T1*, et une case de la liste est vide.

4. Synchronisation de Threads

Solution

- Soit le code suivant exécuté par 2 Threads T1 et T2

```
class ListeTab {
	private String[] tab = new String[50];
	private int index = 0;

	public void syncrhonized ajoute(String s) {
		tab[index] = s;
		index++;
	}
}
```