ÉCOLE
CENTRALE LYON

Compte-Rendu BE 4 : Programmation Parallèle

Groupe :
Tom Calamai
Vianney Dardonville

ÉCOLE CENTRALE LYON

Tables des Matières :

Introduction :

Pour rendre une application concurrente, elle doit être rapide et pouvoir s'exécuter en parallèle d'autre systèmes sans perturber leurs exécutions. La solution technologique pour l'ensemble de ces problèmes est la programmation parallèle avec les threads. Un programme classique effectue le code de façon linéaire, une opération à la fois, cependant il est possible d'utiliser les différents coeurs du processeur simultanément en créant plusieurs threads.

L'objectif de ce BE est de créer plusieurs programmes utilisant la programmation parallèle et donc les threads. Dans ce rapport nous présentons les programmes et résultats obtenus dans 3 exemples différents : le calcul de pi avec la méthode de monte-carlo, la simulation d'un système composé de capteurs et d'actionneurs pour gérer la température et enfin une simulation du jeu de la vie.

I. Calcul de Pi :

1. Principe de l'algorithme :

A l'aide de l'algorithme de Monte-Carlo, il est possible d'obtenir une valeur de PI approchée. L'algorithme est le suivant :

- lancer *n* points aléatoires dans un carré de côté 1
- Si $x^2+y^2<=1$ alors le point appartient au cercle de rayon 1
- compter le nombre de points dans ce cercle

Il alors plutôt simple de réaliser cet algorithme de façon linéaire :

```
public static void QuartDeCercleNoThread(int n) {
    int NombreDePointDansLeQuart = 0;
    //Récupération du random local pour avoir une génération de
point aléatoire plus efficace
    ThreadLocalRandom R = ThreadLocalRandom.current();

    //Coord des points
    double x,y;

    for (int i = 0; i < n; i++)
    {
        //Lancé d'un point
        x = R.nextDouble();
        y = R.nextDouble();

        //Le point appartient-il au cercle ?
        if(x*x + y*y <= 1)
            NombreDePointDansLeQuart++;
    }

    System.out.println("Fin des calculs :
"+NombreDePointDansLeQuart);
}
```

figure 1 : Fonction de calcul de PI sans Thread

On obtient ainsi un résultat satisfaisant, cependant il serait possible d'obtenir ce résultat plus rapidement en utilisant les différents coeurs du processeur.

2. Algorithme parallélisé :

On a donc parallélisé l'algorithme. En effet, il est possible d'effectuer le lancer des points uniquement dans un quart de cercle à la fois. On peut alors faire le calcul pour chacun des quarts de cercle en même temps.
Il faut alors attendre que les calculs soient terminés à l'aide de la fonction *join* avant d'afficher le résultat final.

```
int n = 1000;

//Variables en commun sont dans database
Database db = new Database(0);

//Création de threads :
Thread Q1 = new Thread(new Calcul(n, db));
Thread Q2 = new Thread(new Calcul(n, db));
Thread Q3 = new Thread(new Calcul(n, db));
Thread Q4 = new Thread(new Calcul(n, db));

//Lancement des Threads
Q1.start();
Q2.start();
Q3.start();
Q4.start();

//On attend que les threads aient fini leur calculs
Q1.join();
Q2.join();
Q3.join();
Q4.join();
```

figure : Création des 4 Thread qui lance les calculs dans chacun des quarts

Il faut alors utiliser des classes qui implémentent *runnable*. Elles réalisent un calcul similaire à celui de la figure 1. Cependant elles partagent des données : la somme totale. Il faut donc protéger la modification de cette variable. Pour ce faire nous avons utilisé une classe *database* qui contient l'ensemble des données en commun, ici simplement *nombreDePoint* et qui sert de verrou. Les fonctions *AddPoints* et *getNombreDePoint* sont synchronisées.

```
public class Database {
   public Database(int nombreDePoint);
   //Ajout de point, on synchronise pour qu'un seul thread à la
fois ajoute des
   //point au nombre total
   synchronized void AddPoints(int Points);
   //Récupération du nombre total de point
   synchronized int getNombreDePoint();
}
```

figure : Prototype de la classe database

3. Résultat de la parallélisation :

Pour les petites valeurs de n, c'est à dire pour peu de lancets, le gain n'est pas significatif. En effet il faut construire les threads, les lancer, les initialiser mais aussi les détruire, ce qui ajoute un certain nombre d'opérations.
Cependant une fois un seuil n0 atteint, le temps de calcul avec 4 threads est significativement meilleur.

n	Durée (1 Thread)	Durée (4 Thread)	Rapport des durées
100	0.006520057 sec	0.00405408 sec	1.6
10000	0.011356106 sec	0.008072291 sec	1.4
1000000	0.381533225 sec	0.124533169 sec	3
10000000	3.610470001 sec	1.025333375 sec	3.5
100000000	35.310909089 sec	8.741468101 sec	4.039

Le rapport tend logiquement vers 4, car il y a 4 Threads au lieu d'un. Le temps de calcul va donc être approximativement divisé par 4.

II. Température :

1. Principe :

On cherche à simuler le fonctionnement d'un système multi-tâche, soit un système exécutant plusieurs actions simultanément. Ici, ce système est un régulateur de température et de pression. Le système est donc composé d'un système de chauffage, d'un pompe, d'un écran informant l'utilisateur de la température et de la pression, d'un gestionnaire de température et de pression qui seront intégré ici dans un contrôleur qui coordonne l'ensemble des composants. On va créer un thread par composant afin de simuler un système multi-tâche.

2. Programme :

Comme indiqué ci-dessus, on va décomposer le système en threads : un par composant. Nous avons fait le choix d'utiliser des threads et non des processus pour une meilleur rapidité de l'IPC (les threads partageant le même espace mémoire). Chacun des threads a été implanté à l'aide de l'interface Runnable et la redéfinition de la méthode run().
Exemple sur le thread gérant le chauffage :

```
16  public class chauffage implements Runnable{
17
18      //Implémentaion de l'interface runnable
19      int j;
20      int Z;
21      Controleur ctr; //A définir dans le constructeur pour
22
23      public chauffage(int j, int Z, Controleur ctr) {
24          this.j = j;
25          this.Z = Z;
26          this.ctr = ctr;
27      }
28
29      public void run() {
30          //La méthode run est une éthode de l'interface run
31      boolean Continuer=true;
32      //Tant que la température n'est pas bonne, on repete l
33      while (Continuer){
34          boolean a=ctr.get_chauffage();
35          if (a){
36
37                  Temperature.memT++;
38                  //On simule l'augmentation de la températu
39          }
40
41          else {
42              Continuer = false;
43              System.out.println("Arrêter le chauffage");
```

Figure : Classe chauffage implémentant Runnable

Les threads communiquent entre eux via un espace de mémoire partagée. Bien que cette utilisation soit rapide et efficace elle peut provoquer des problèmes d'accès et d'utilisation/modification simultanée des données par les différents threads. Plusieurs méthodes de synchronisation existe pour pallier à cet inconvénient. Les variables memT et memP sont lus par différents threads mais modifié respectivement que par chauffage et pompe. La déclaration de ces variables en volatiles qui a pour effet de "flusher" la mémoire cache en RAM pour que les autres threads aient accès à la toute dernière valeur est un bon moyen dans notre cas de s'affranchir de ce problème de synchronisation :

```
12    public class Temperature {
13
14        static volatile double memT=10;
15        static volatile double memP=5;
```

figure : Déclaration des variables globales volatiles

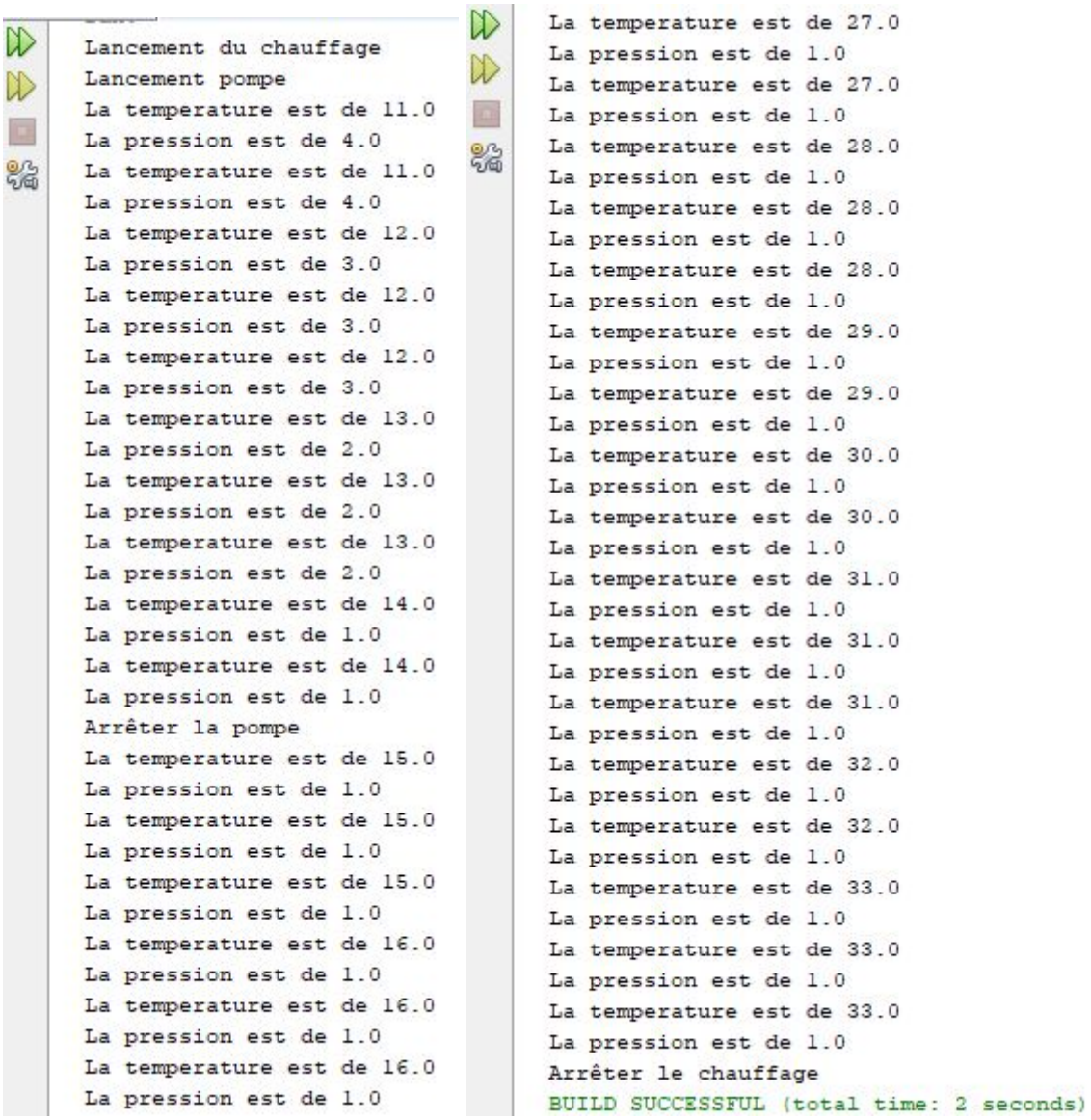
Le contrôleur est au centre du système. A chaque tour de boucle, il défini par une série de test (if/else) s'il faut allumer ou éteindre le chauffage ou la pompe :

```
45            for (int i=1 ; i<=j ; i++){
46                T=Temperature.memT;
47                P=Temperature.memP;
48                // On va chercher les variables globales memT et memP
49
50                //On teste ici la valeur de la température/Pression par rapport au seuil deman
51                if(T>=Seuil_T){
52                    go_chauffage = false;}
53                else {
54                    if(!go_chauffage)
55                        System.out.println("Lancement du chauffage");
56                    go_chauffage=true;
57                        }
58
59                if(P> Seuil_P){
60                    if(!go_pompe)
61                        System.out.println("Lancement pompe");
62                    go_pompe = true;
63                    }
64                else{
65                    go_pompe = false;
66
67                    }
68
69                try {
                      sleep(x);
71                } catch (InterruptedException ex) {
72                    Logger.getLogger(Controleur.class.getName()).log(Level.SEVERE, null, ex);
73                }
74        }}
```

Figure : Boucle de tests dans controleur

3. Résultats :

En initialisant les seuils à 33° et 1 bar et la température de départ à 10°, la pression à 5 bar on obtient le résultat suivant :



```
Lancement du chauffage
Lancement pompe
La temperature est de 11.0
La pression est de 4.0
La temperature est de 11.0
La pression est de 4.0
La temperature est de 12.0
La pression est de 3.0
La temperature est de 12.0
La pression est de 3.0
La temperature est de 12.0
La pression est de 3.0
La temperature est de 13.0
La pression est de 2.0
La temperature est de 13.0
La pression est de 2.0
La temperature est de 13.0
La pression est de 2.0
La temperature est de 14.0
La pression est de 1.0
La temperature est de 14.0
La pression est de 1.0
Arrêter la pompe
La temperature est de 15.0
La pression est de 1.0
La temperature est de 15.0
La pression est de 1.0
La temperature est de 15.0
La pression est de 1.0
La temperature est de 16.0
La pression est de 1.0
La temperature est de 16.0
La pression est de 1.0
La temperature est de 16.0
La pression est de 1.0
La temperature est de 27.0
La pression est de 1.0
La temperature est de 27.0
La pression est de 1.0
La temperature est de 28.0
La pression est de 1.0
La temperature est de 28.0
La pression est de 1.0
La temperature est de 28.0
La pression est de 1.0
La temperature est de 29.0
La pression est de 1.0
La temperature est de 29.0
La pression est de 1.0
La temperature est de 30.0
La pression est de 1.0
La temperature est de 30.0
La pression est de 1.0
La temperature est de 31.0
La pression est de 1.0
La temperature est de 31.0
La pression est de 1.0
La temperature est de 31.0
La pression est de 1.0
La temperature est de 32.0
La pression est de 1.0
La temperature est de 32.0
La pression est de 1.0
La temperature est de 33.0
La pression est de 1.0
La temperature est de 33.0
La pression est de 1.0
La temperature est de 33.0
La pression est de 1.0
Arrêter le chauffage
BUILD SUCCESSFUL (total time: 2 seconds)
```

On voit que le programme tourne correctement et affiche les bonnes informations à l'écran, notamment concernant l'affichage des instructions "Arrêter la pompe", "Arrêter le chauffage"...

III. Jeu de la vie :

1. Principe du jeu :

Le jeu de la vie (Game of life) est un automate cellulaire, réalisé en 1970 par John Conway afin de montrer que des comportements complexes peuvent émerger d'un jeu de règles très simple.
Le jeu se déroule sur un plateau de taille fixe ou infini, pour la simulation réalisée lors de ce BE, la taille est fixée par la variable *L* dans la fonction main.

Chaque case représente une cellule qui peut être dans deux états différents : morte ou vivante.

Les règles sont alors très simple :

- Chaque cellule vivante doit avoir 2 ou 3 voisins pour survivre
- Une cellule avec plus de 3 voisins meurt de surpopulation
- Une cellule avec moins de 2 voisins meurt d'isolement
- Enfin une cellule non vivante avec exactement 3 voisins devient vivante

La réalisation de ce genre de jeu permet d'observer des objets nouveaux aux comportements inattendus tels que les Vaisseaux qui peuvent "se déplacer" le long du plateau.

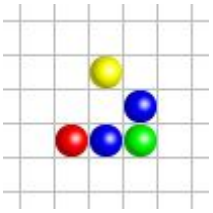


figure : Structure d'un Vaisseau simple

Afin de réaliser une simulation du jeu de la vie il est nécessaire de vérifier à chaque tour toutes les cases pour savoir si elles seront vivantes ou non au tour suivant. Cela représente donc L^2 opération. Pour parallélisé l'ensemble des vérifications, il est alors possible de créer un Thread par cellule, pour effectuer l'ensemble des vérifications simultanément.

2. Programme :

Dans la fonction principale, il s'agit de générer le plateau de jeu composé d'un certain nombre de cellules. La fonction *afficher* affiche simplement le plateau de jeu, c'est à dire l'ensemble des cellules. A chaque tour on met donc à jour le plateau avec la fonction *mise_a_jour* puis on affiche le résultat obtenu.

```
public static void main(String[] args) throws Exception
{
   int l = 4;//largeur du plateau
   cellule[][] plateau = new cellule[l][l];
   //Génération du plateau aléatoirement :
   for(int i = 0; i < l; i++)
       for(int j = 0; j < l; j++)
           if(Math.random() > 0.6) //Probabilité d'avoir une cellule
vivante 40%
               plateau[i][j] = new cellule(true);
           else
               plateau[i][j] = new cellule();
   //On effectue 5 tours
   for(int i = 0; i < 5; i++)
   {
```

```
        System.out.println("tour " + i);
        afficher(plateau); //Affichage du plateau
        plateau = mise_a_jour(l, l, plateau); //Mise à jour du plateau
    }
}
```

figure : boucle principale

Il faut donc définir la classe cellule qui représente une case du plateau. Il s'agit simplement d'une classe qui donne l'état de la cellule : vivante ou non.

```
public class cellule {

    //Etat de la cellule : true = vivante, false = morte
    private boolean Vivant;
    //Constructeur
    public cellule();
    //Constructeur qui permet de donner l'état de la cellule
    public cellule(boolean vie);
    //Retourne l'état de la cellule
    public boolean EstVivant();
    //met la cellule dans l'état vivant
    public void Vivant();
    //met la cellule dans l'état mort
    public void Mort();
}
```

figure : prototype de la classe cellule

La fonction mise à jour permet de créer et d'intialiser les threads qui vont traiter les cellules pour mettre à jour le plateau.
threads est la liste qui contient l'ensemble des threads afin de pouvoir facilement réaliser des opérations sur les différents threads et en particulier attendre que l'ensemble des calculs soient terminés avec la fonction *join.*
La classe *jeu* est celle qui va réaliser l'ensemble des opérations en parallèle c'est à dire évaluer l'état de la cellule au tour suivant.

```
public static cellule[][] mise_a_jour(int colonnes, int lignes,
cellule[][] plateau) throws Exception
{
    //Liste de l'ensemble des threads :
    ArrayList<Thread> threads = new ArrayList<>();
    //Création du nouveau plateau :
    cellule[][] nouveauPlateau = new
cellule[plateau.length][plateau[0].length];
    //Verification de toutes les cases :
```

```
    for(int l = 0; l < lignes; l++)
    {
        for(int c = 0; c < colonnes; c++)
        {
            //Créer la nouvelle cellule
            nouveauPlateau[l][c] = new cellule();
            Integer loc_c = c;
            Integer loc_l = l;
            //Création du thread qui s'occupe de cette case :
            Thread verification_cellule = new Thread(new Jeu(plateau,
nouveauPlateau[loc_l][loc_c], loc_c, loc_l));
            verification_cellule.start();
            threads.add(verification_cellule); //On l'ajoute à notre
liste de threads;
        }
    }
    //attendre que tous les threads finissent
    for(Thread t : threads)
        t.join();

    return nouveauPlateau;//On retourne le nouveau plateau de jeu mis à
jour
}
```

figure : fonction de mise à jour du plateau

Enfin la classe jeu, comme expliqué ci-dessus, réalise l'évaluation de la cellule. Plus particulièrement elle réalise 2 opérations : compter le nombre de voisin et selon le nombre de voisin changer l'état de la case. Cette classe implémente l'interface *Runnable* qui permet de lancer le thread, elle possède donc une fonction *run* qui permet de réaliser le traitement de la cellule.

```
public class Jeu implements Runnable{
    //Initialisation de l'ensemble des variables necessaire pour la mise
à jour de la cellule
    public Jeu(cellule[][] plateau, cellule nouvelleCell, int colonne,
int ligne);
    //Lancement l'opération réalisé dans le thread
    @Override
    public void run();
    //Fonction de verification et de mise à jour :
    private void UpdateCell();
}
```

figure : prototype de la classe Jeu

Ce code permet ainsi de réaliser la simulation numérique du jeu de la vie à l'aide de plusieurs threads.

3. Résultat :

On constate que la simulation se déroule sans encombre, le plateau est bien mis à jour. On peut retrouver des objets classiques du jeu de la vie.

- Carré : 4 cellules vivantes disposées en carré formant une structure figé

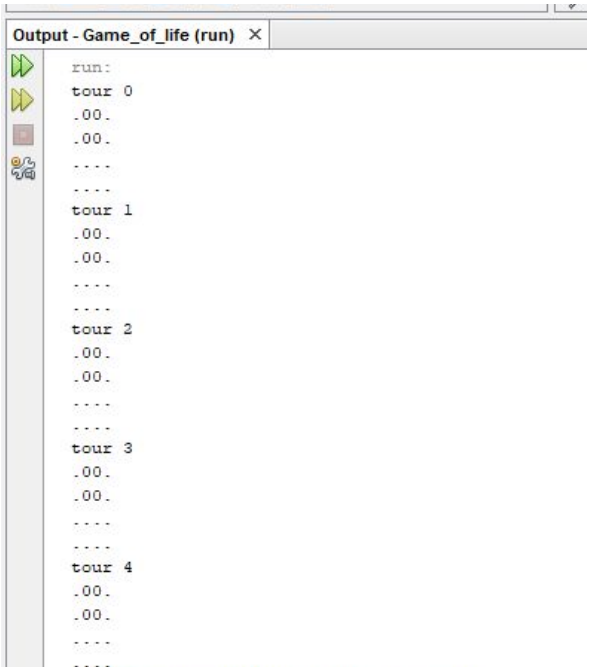


figure : Carré

- Ligne oscillante : 3 cellules alignées et cote à cote, alternativement verticale et horizontale

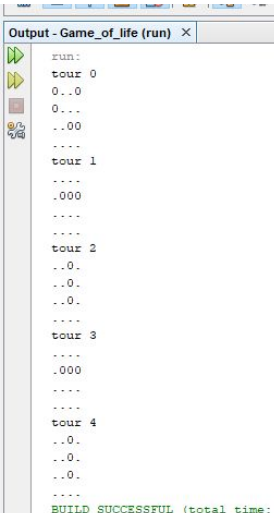


figure : ligne

- Cycle : tout cycle de cellule est stable car chaque cellule possède 2 voisins exactement

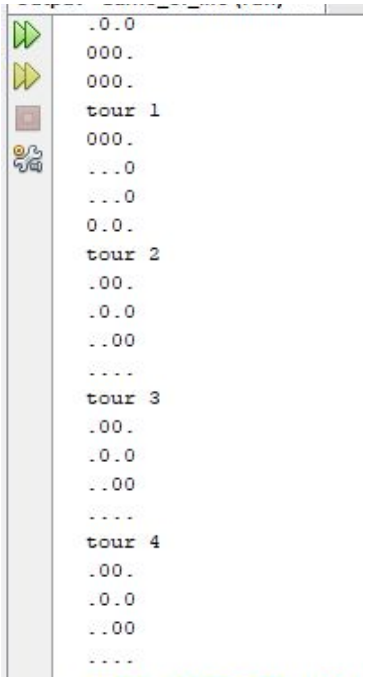


figure : cycle

Le jeu de la vie tend toujours vers un système stable et périodique, c'est à dire qu'après un certain nombre de tours, les seules structures encore existantes sont soit constantes soit périodiques.
Nous avons décidé de créer un thread par cellule mais cela est tout de même très coûteux en terme de processeur et très peu de machines sont capables de créer plus de 8 threads réellement parallèles, le gain en vitesse de calcul ne doit donc pas être si important. De plus la taille du plateau est limitée par le nombre de threads que l'on peut instancier.
Nous aurions pu créer un thread par ligne pour réduire le nombre de threads, ou encore créer un nombre fixe de threads et répartir les cellules à traiter dans ces différents threads. Le code serait sensiblement identique.

Conclusion :

Ces trois exemples présentent certains des aspects essentiels des threads et de la programmation parallèle. Ils font appels à des fonctions communes et des variables partagées qui doivent donc être synchronisées et protégées.
Le calcul de pi met en avant le gain de temps qu'entraîne une architecture parallèle, car plusieurs opérations sont effectuées simultanément. Le jeu de la vie présente la possibilité de traiter de nombreuses opérations en même temps. Enfin le système chauffage et pompe, introduit la notion de découplage entre prise de mesure et traitement de l'information ainsi qu'une solution simple de synchroniser l'accès en mémoire à l'aide des variables volatiles.

La programmation parallèle est essentielle dans de nombreux domaines, en informatique comme en ingénierie de manière générale et permet la réalisation d'applications concurrentes plus efficaces.