Parallélisme

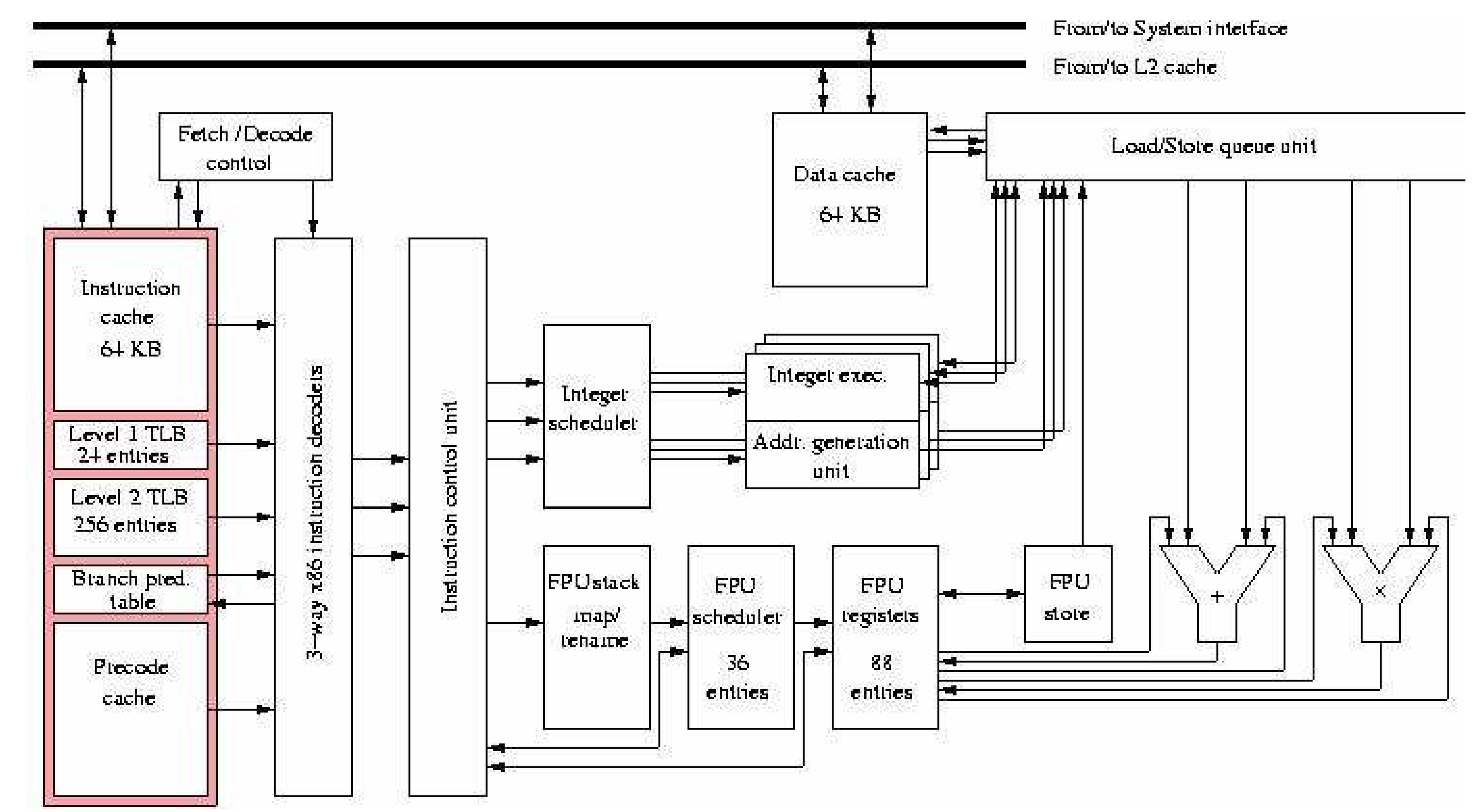
Eric Goubault
Commissariat à l'Energie Atomique
Saclay

Pourquoi le parallélisme?

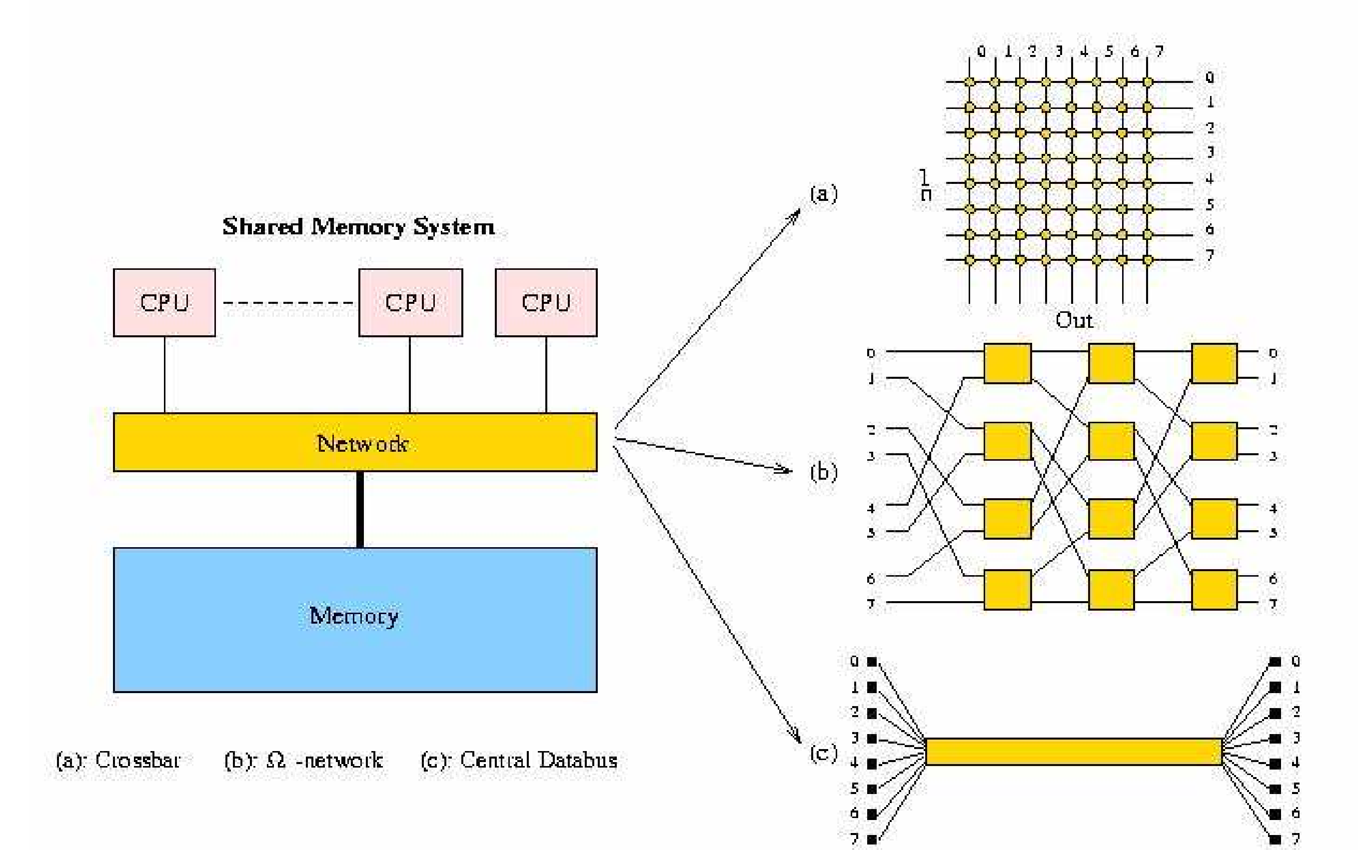
Le parallélisme, d'exotique il y a encore peu, est devenu indispensable:

- Les microprocesseurs, les machines ne sont presque plus "séquentielles"
- Les systèmes d'exploitation (UNIX...) font tourner en permanence des dizaines de tâches au moins
- Pour des raisons d'efficacité de calcul en temps et en mémoire, de plus en plus de programmes sont parallélisés (historiquement, chez les numériciens)
- Certains problèmes sont intrinsèquement parallèles ou distribués (grandes bases de données: transactions simultanées! etc.)

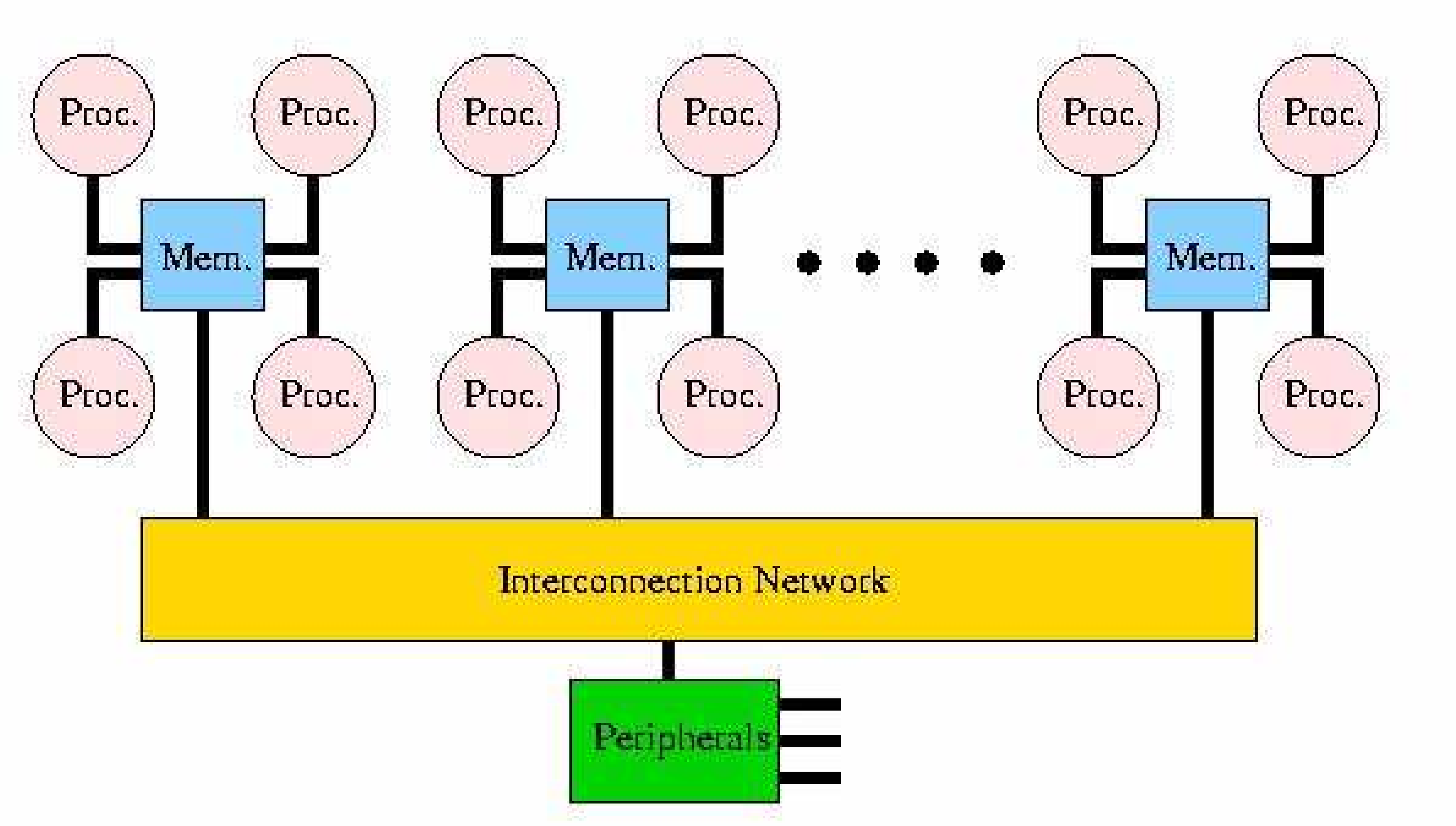
Au niveau d'un microprocesseur (Athlon)...



Au niveau d'une machine...



Au niveau d'un réseau de machines...

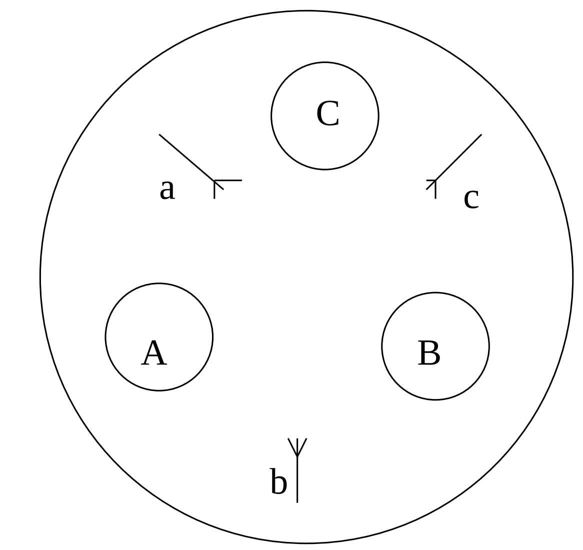


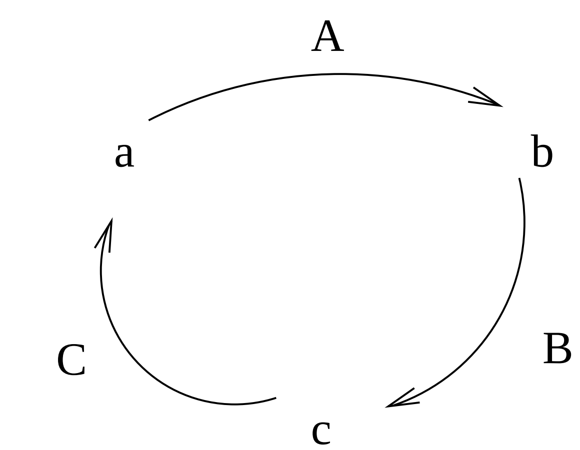
La machine numéro 1 actuellement...

(IBM ASCI White, Lawrence Livermore National Laboratory, 2000, 8192 processeurs Power3 à 375Mhz: 12.3 teraflops! - simulation des armes thermonucléaires)

Mais subtil à programmer...

- Nouvelles primitives (→ programmation)
- Nouveaux types de bugs, dans la coordination des tâches (→ sémantique mathématique); exemple académique (trois philosophes qui dînent):





- Difficulté d'atteindre les performances crêtes (→ algorithmique)

Programme

- Un peu d'architecture parallèle,
- Les "threads" en JAVA
- Les "nouveaux" problèmes liés au parallélisme (problème de coordination, exclusion mutuelle, points morts etc.)
- L'algorithmique parallèle de base
- L'ordonnancement de tâches
- Au delà du multitâche ("clusters" de stations, distribution etc.)

Organisation

- Complémentaire du cours système (...avec une intersection non vide!)
- Cours suivis de TPs et de TDs (utilisation des threads JAVA)
- Ecrit+Miniprojets
- Voir par exemple le cours de l'année dernière: http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/ParaI.html

Plan de ce cours en particulier

- Introduction
- Modèles du parallélisme, une première approche
- Architectures Parallèles et Distribuées, quelques exemples
- Compléments JAVA (extraits du cours de Didier Rémy)
- Threads JAVA

Introduction

Machine **parallèle** = ensemble de **processeurs** qui **coopèrent** et **communiquent**

Historiquement, **réseaux** d'ordinateurs, machines **vectorielles** et faiblement parallèles (années 70 - IBM 360-90 vectoriel, IRIS 80 triprocesseurs, CRAY 1 vectoriel...)

Permet de résoudre bien plus rapidement certains problèmes:

- Résolution numérique d'équations aux dérivées partielles, par exemple,
  - prévisions météorologiques,
  - essais aérodynamiques (simulations),
  - ...
- Cryptographie (décrypter des messages DES),

ainsi que de résoudre certains problèmes spécifiques,

- Bases de données réparties (réservation aérienne...),
- ...

Taxonomie de **Tanenbaum**:

Quatre types: **SISD**, **SIMD**, **MISD** et **MIMD**.

Basée sur les notions de flot de **contrôle** et flot de **données**.

Taxonomie de **Tanenbaum** (1):

Machine **SISD** = **S**ingle **I**nstruction **S**ingle **D**ata
= Machine Von Neuman

Exemple

Le code suivant,

```
int A[100];
...
for (i=1;100>i;i++)
   A[i]=A[i]+A[i+1];
```

s'exécute sur une machine séquentielle en faisant les additions `A[1]+A[2]`, `A[2]+A[3]`, `etc.`, `A[99]+A[100]` à la suite les unes des autres.

Taxonomie de **Tanenbaum** (2):

Machine **SIMD** = **S**ingle **I**nstruction **M**ultiple **D**ata

(1) Parallèle (Connection Machine)

(2) Systolique

En général exécution **synchrone**

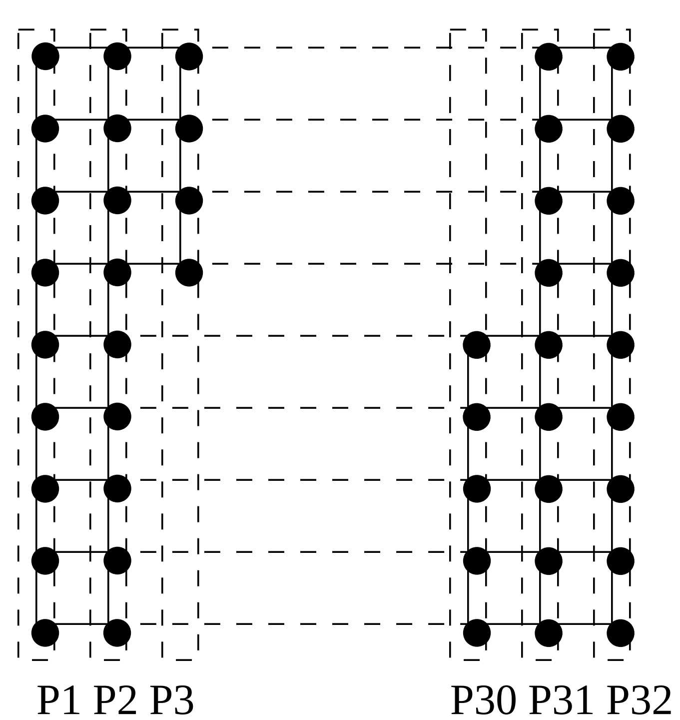
Exemple

En CM-Fortran sur la Connection Machine-5 avec 32 processeurs,

```
        INTEGER I,A(32,1000)
CMF$    LAYOUT A(:NEWS,:SERIAL)
        ...
        FORALL (I=1:32,J=1:1000)
     $     A(I:I,J:J)=A(I:I,J:J)+A(I:I,(J+1):(J+1))
```

Chaque processeur i, $1 \leq i \leq 32$ a en sa mémoire locale une tranche du tableau `A`: `A(i,1), A(i,2), ..., A(i,1000)`. Il n'y a pas d'interférence dans le calcul de la boucle entre les différentes tranches: tous les processeurs exécutent la même boucle sur sa propre tranche en même temps.

Exemple



Taxonomie de **Tanenbaum** (3):

Machine **MISD** = **M**ultiple **I**nstruction **S**ingle **D**ata

Processeurs vectoriels, architectures pipelines

Exemple

Pipelinage d'une addition vectorielle,

```
FOR  i:=1 to n DO
   R(a+b*i):=A(a'+b'*i)+B(a''+b''*i);
```

`A`, `B` et `R` sont placés dans des registres vectoriels qui se remplissent au fur et à mesure du calcul,

Exemple

Temps	$A(i)$	$B(i)$	$R(i)$
1	1 . . .	1 . . .	. . . .
2	2 1 . .	2 1 . .	. . . .
3	3 2 1 .	3 2 1 .	. . . .
4	4 3 2 1	4 3 2 1	. . . .
5	5 4 3 2	5 4 3 2	1 . . .
6	6 5 4 3	6 5 4 3	2 1 . .
etc.			

Taxonomie de **Tanenbaum** (4):

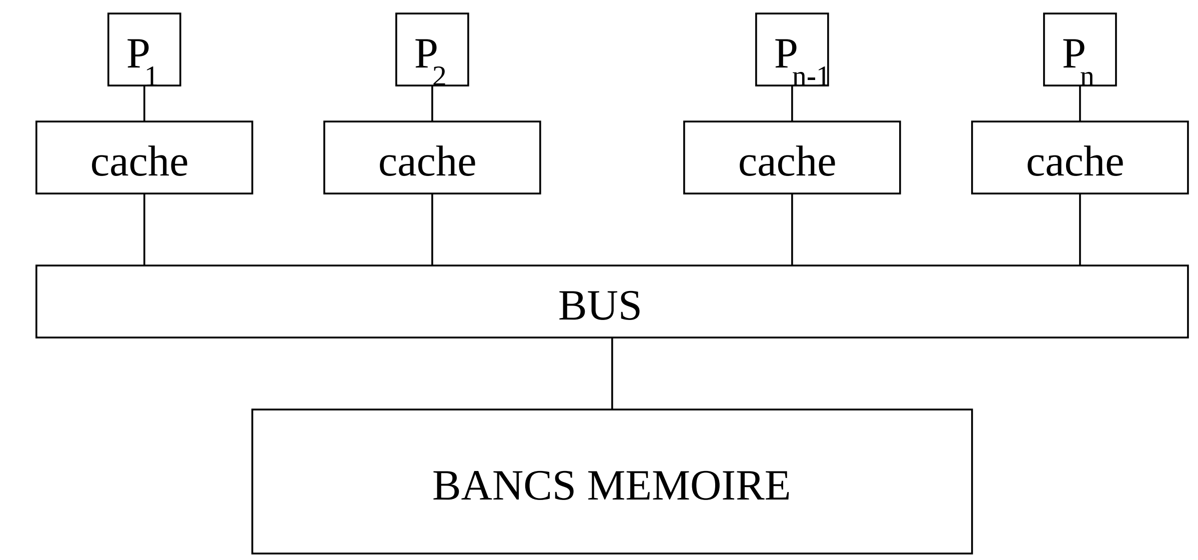
Machine **MIMD** = **M**ultiple **I**nstruction **M**ultiple **D**ata

(1) Mémoire partagée (Sequent)

(2) Mémoire locale + réseau de communication (Transputer, Connection Machine, local, par réseau d'interconnexion) - Système réparti

C'est le cas (2) que l'on va voir plus particulièrement avec JPVM. On pourra également simuler le cas (1) (que l'on verra plus avec les threads JAVA).

Mémoire Partagée

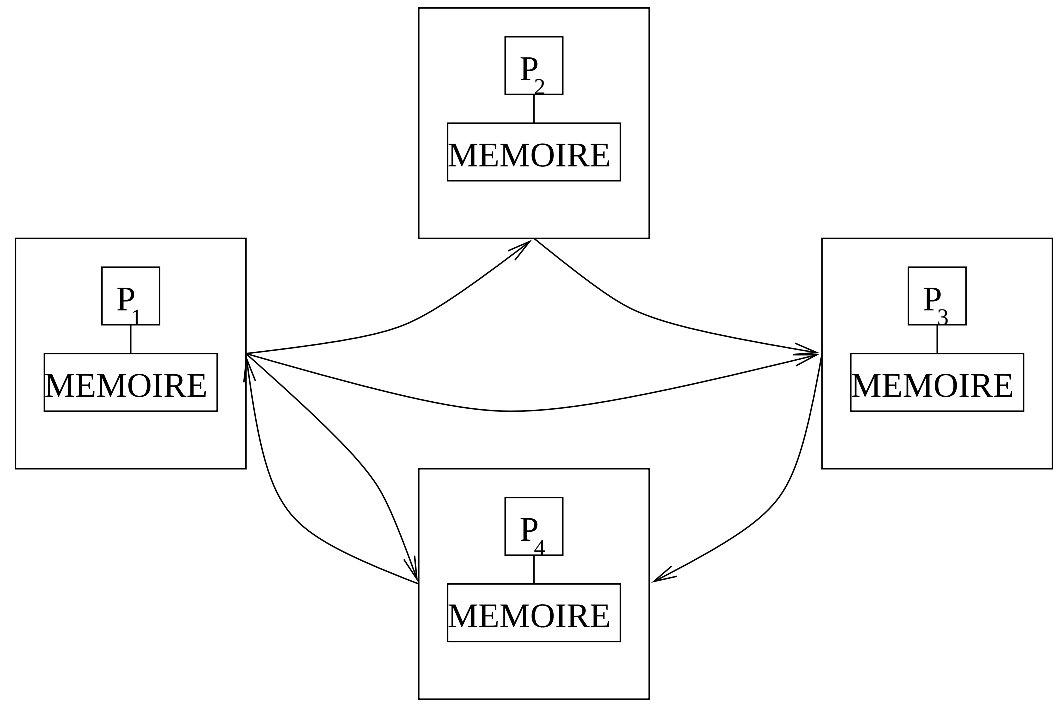


Mémoire Partagée

Synchronisation par,

- Barrières de synchronisation,
- Sémaphores : deux opérations P et V.
- Verrou (mutex lock) : sémaphore binaire qui sert à protéger une section critique.
- Moniteurs : construction de haut niveau, verrou implicite.

Machine distribuée



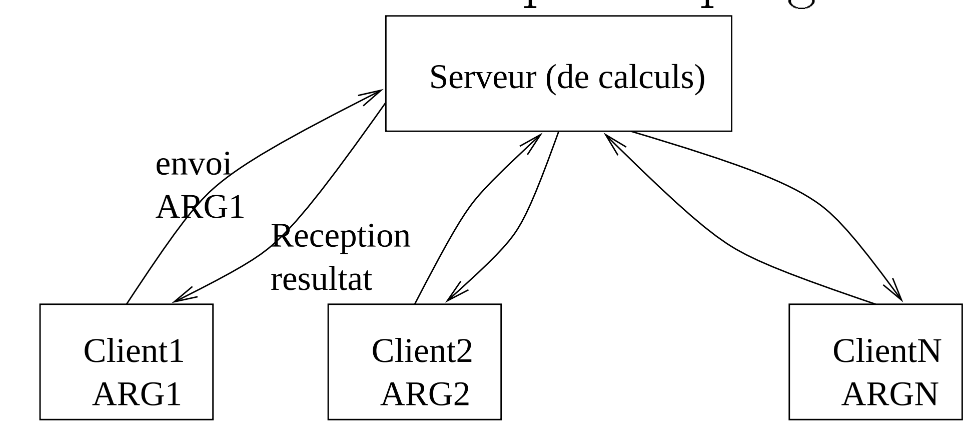
Machine distribuée

Synchronisation et échange d'information par,

- Appel de procédure distribuée (RPC) :
  - réponse synchrone
  - réponse asynchrone
- Envoi/Réception de message asynchrone (tampon de taille limitée ou non); active polling ou gestion à l'arrivée par une procédure `handler`.
- Envoi/Réception de message synchrone : rendez-vous.
- Mélange des deux derniers cas.

Exemple – Clients/Serveurs

Le protocole RPC ("Remote Procedure Call") entre machines UNIX avec réseau Ethernet est un exemple de programmation MIMD:



Remarques

Architectures plus ou moins adaptées à certains problèmes.

Gain de temps espéré: au plus N (nombre de processeurs). Il existe néanmoins des problèmes avec gains supérieurs (recherche distribuée).

En pratique, parallélisme difficile à contrôler

Programmation parallèle:

(1) Language séquentiel: le compilateur **parallélise** (Fortran...),

(2) Constructions parallèles **explicites** (Parallel C, Occam...)

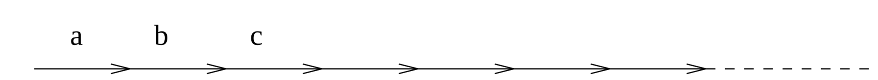
Constructions parallèles explicites

- Création de parallélisme,
  - Itération simultanée sur tous les processeurs (**FOR** parallèle de Fortran),
  - Définition d'un ensemble de **processus** (**COBEGIN**),
  - Création de **processus** (**FORK** de Unix)
- Contrôle du parallélisme,
  - Synchronisation (**Rendez-vous** d'Ada),
  - Passage de messages (synchrones/asynchrones, **broadcast**,...),
  - Section critiques (gestion de la mémoire partagée),
  - Stopper une exécution parallèle (**COEND**, **WAIT** d'Unix)

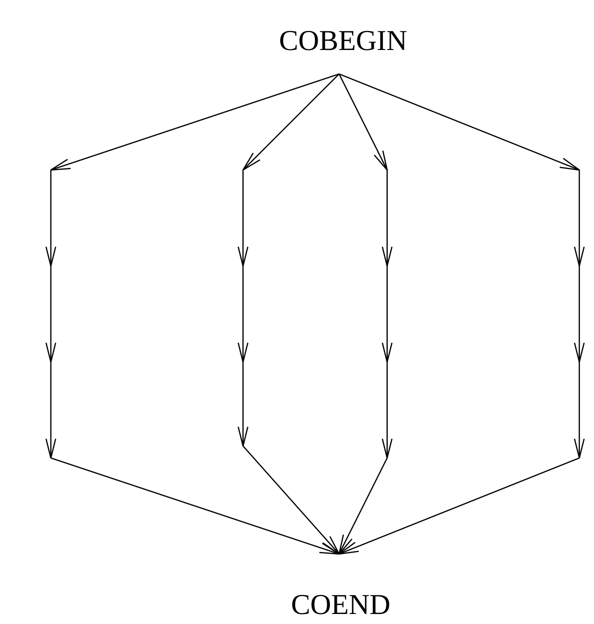
Processus (MIMD)

Un **Processus** est une unité de programme **séquentielle** (abstraction d'un processeur) avec,

- sa propre pile de données,
- son compteur ordinal,
- éventuellement sa mémoire locale



Constructions parallèles explicites (suite):



Constructions parallèles explicites (suite):

Synchronisation: but = assurer qu'à un moment donné tous les processus ont suffisament avancés leurs calculs.

Erreurs possibles: **Interblocage** (deadlock, livelock), **Famine**,... (cours 2 et 3)

Exemples:

- X dit à Y: "Donne moi ton nom et je te dirai le mien". Y dit la même chose à X. Ceci est une situation d'interblocage.
- X et Y veulent tous deux utiliser un objet. X est plus rapide qu'Y et obtient régulièrement l'utilisation de cet objet avant Y qui ne l'obtient jamais. On dit que Y est en situation de famine.

Constructions parallèles explicites (suite):

MIMD mémoire partagée,

Erreurs possibles: **incohérence** des données.

Partant de $x = 0$, on exécute $x := x + x$ en parallèle avec $x := 1$. Par exemple, on a ces trois exécutions possibles:

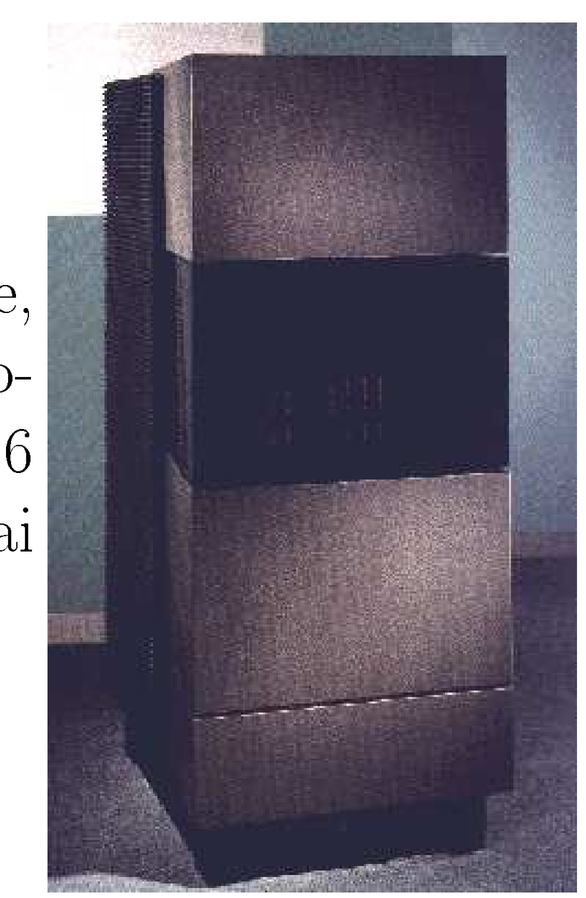
`LOAD x,R1`	`WRITE x,1`	`LOAD x,R1`
`LOAD x,R2`	`LOAD x,R1`	`WRITE x,1`
`WRITE x,1`	`LOAD x,R2`	`LOAD x,R2`
`WRITE x,R1+R2`	`WRITE x,R1+R2`	WRITE x,R1+R2
Résultat x=0	Résultat x=2	Résultat x=1

Pour y remédier, **sections critiques** et **exclusion mutuelle**.

La Connection Machine-5

- De 32 à 65536 noeuds qui chacun comporte,
  - un processeur Sparc (32 MHz, 22 Mips, 5 MFlops)
  - 4 unités vectorielles (32 MFlops chacune) pour les opérations flottantes
  - 4 bancs mémoire de 8 Mo de DRAM
- Réseau d'interconnexion en arbre gras (hypercube), entre voisins on a: bande passante: 20 Mo/s, latence: 5 μs
- Entrées/Sorties en parallèle à un système de disques, bande passante: 33 Mo/s, latence: 75 ms

Plusieurs modèles installés en France, dont un à l'ETCA (32 noeuds, 2 processeurs de contrôle, 1 Go de RAM, 16 Go de disques, 4 GFlops (crête), mai 1993).



CM-5

Fonctionne en,

- mode SIMD: CM Fortran (sous-ensemble de Fortran 90 et instructions spécifiques d'organisation des données en parallèle), C*
- mode MIMD: de même mais en utilisant la librairie CMMD de passage de messages (on retrouve des ordres similaires dans PVM)

Cray T3E

La machine la plus puissante chez CRAY, deux types de modèles, refroidissement à air (pas plus de 128 processeurs) ou par un liquide (jusqu'à 2048 processeurs). En amalgamant les deux,

- De 16 à 2048 noeuds comprenant chacun,
  - un DEC Alpha EV5 (600 MFlops)
  - de 64 Mo à 2 Go de mémoire vive
- Un réseau d'interconnexion en tore 3D (bande passante de 2 à 128 Go/s). Entrées sorties par un réseau en double anneau (allant dans des sens inverses), "GigaRing"

Performance de crête de 9.6 GFlops à 1.2 TFlops, jusqu'à 4 To de mémoire vive.

CRAY T3E

CRAY T3E

Peut être utilisé,

- langages: Fortran 90, C et C++ (compilateurs optimisants)
- mode MIMD en utilisant la "Cray Message Passing Toolkit" supportant PVM, MPI (passage de messages) ou la "Shared Memory Library" (le CRAY T3E est organisé logiquement en mémoire partagée même si physiquement ce n'est pas le cas)
- mode SIMD en utilisant Fortran 90, CRAFT ou HPF

IBM SP2

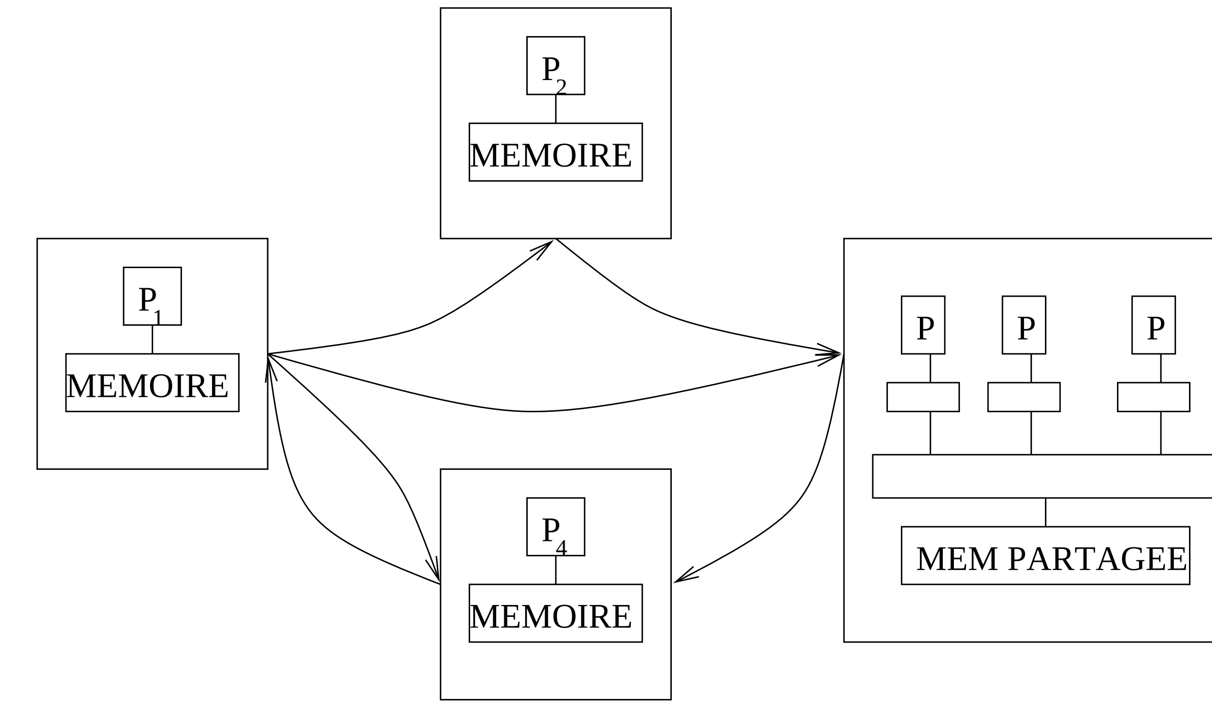
IBM RISC/6000 Scalable POWERparallel System (SP):

- Chaque noeud est constitué d'un processeur RS/6000 soit à 120 MHz ("thin node" – 480 MFlops) avec 256Mo de mémoire vive et 1,8 Go de mémoire disque, soit à 135 MHz ("wide node" – 540 MFlops) avec 1 à 2 Go de mémoire vive.
- Communication par passage de messages à travers un réseau cross-bar à deux niveaux; bande passante de 150Mo par seconde.
- Fonctionne en MIMD avec PVM ou MPI
- Fonctionne en SIMD avec HPF (High Performance Fortran)

Réseaux de stations

Pas d'horloge globale, mode MIMD, communication par passage de messages (typiquement gérés par PVM) sur un réseau local ou d'interconnexion.

Architectures hybrides



Combinaison sur un réseau local de stations et de machines multi-processeurs (elles-même gérant leur parallélisme interne par mémoire partagée ou par bus ou réseau). JPVM peut également gérer ce genre d'architectures.

Réseau de machines sous PVM

PVM ("Parallel virtual Machine"):

- librairie C (et Fortran) de communication par passage de messages développée au "Oak Ridge National Laboratory"

  (http://www.netlib.org/pvm3)

- composé d'un processus "démon" pvmd3 et de bibliothèques d'interfaçage (libpvm3.a et libgpvm3.a pour le C)

Réseau de machines sous PVM

Stations	OS	Super-calculateurs
DEC	BSD	Convex C2, Exemplar
IBM	System-5	Cray C90, T-3D
HP	Linux	Intel Paragon, Hypercube
Silicon Graphics		IBM SP-2, 3090
Sun		TMC CM-2, CM-5
NeXT		Kendall Square
Data General		Sequent
386/486/586 Unix		Maspar
		Encore
		Fujitsu
		Stardent

La machine PVM

La machine PVM est une collection de machines réelles sur un même réseau sur lesquelles un démon pvmd3 tourne:

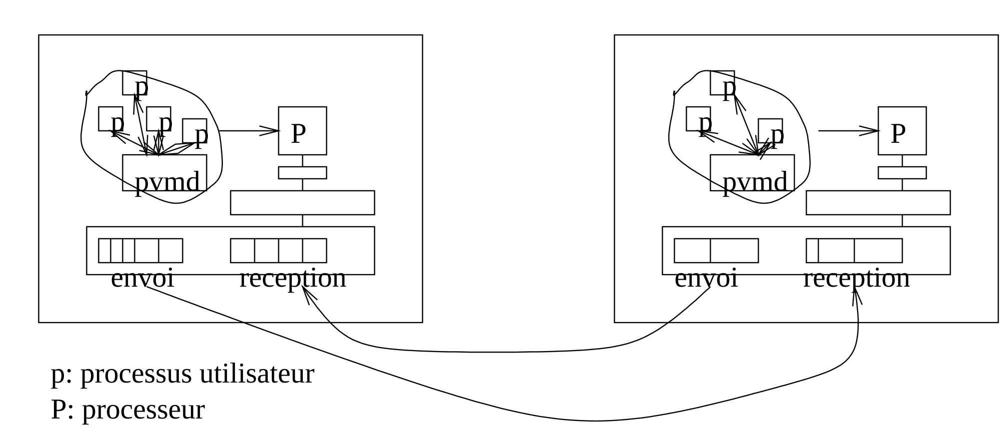
- pvmd3 est un processus UNIX qui contrôle les processus utilisateurs au sein d'une application PVM et qui coordonne les communications entre les machines qui composent la machine PVM
- un démon pvmd3 par ordinateur et par utilisateur de la machine PVM qui maintient la configuration globale et locale de la machine PVM
- chaque machine (et architecture) doit avoir son propre pvmd3

La machine PVM

La librairie PVM permet de communiquer avec ces démons pvmd3 pour,

- créer et détruire des processus,
- gérer des tampons de communication, envoyer, recevoir (point à point ou par broadcast) des messages (de façon synchrone ou asynchrone),
- synchroniser des processus (par barrières de synchronisation)
- connaître l'état de la configuration de la machine PVM et pour la modifier (dynamiquement)

La machine PVM



Calcul de Pi

Supposons que nous voulions calculer Pi en parallèle par la formule suivante,

$$
\begin{aligned}
\pi &= \int_0^1 \frac{4}{1+x^2}dx \\
&\cong \sum_{i=1}^{n} \frac{1}{n}\frac{4}{1+\left((i-\frac{1}{2})\frac{1}{n}\right)^2}
\end{aligned}
$$

Calcul de Pi

Une solution est le paradigme **Maître/Esclave**: Un maître va lancer `N` esclaves chargés de calculer les sommes partielles,

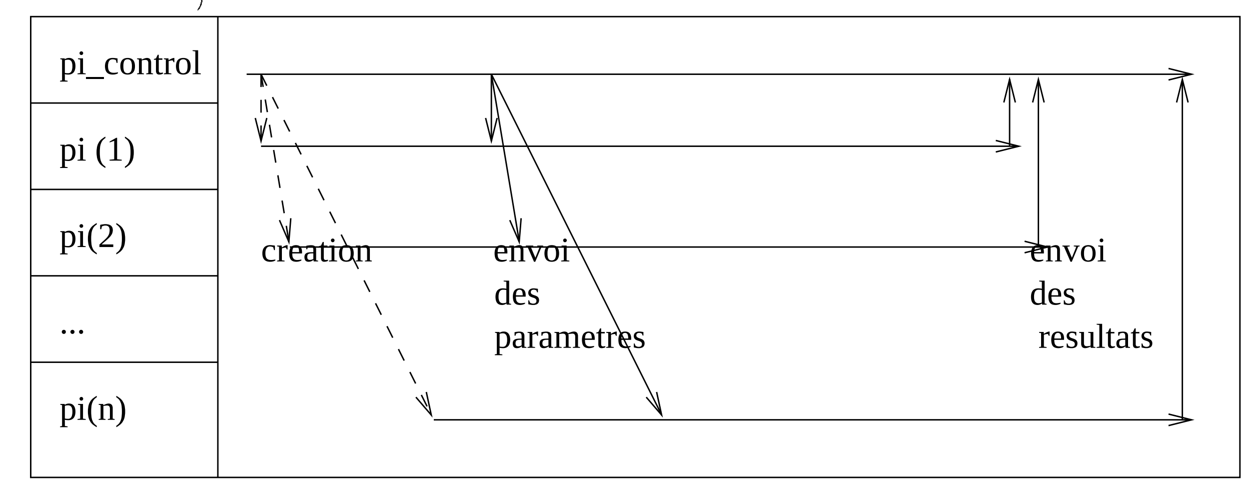
$$P_k = \Sigma_{i=k*n/N}^{(k+1)*n/N} \frac{1}{n}\frac{4}{1+\left((i-\frac{1}{2})\frac{1}{n}\right)^2}$$

pour $k = 0, \cdots, \mathtt{N}-\mathtt{1}$.

C'est une parallélisation par décomposition fonctionnelle. On pourrait aussi faire de la décomposition de domaine pour calculer une fonction s'appliquant independemment à des sous-blocs d'un tableau.

Flôt de contrôle

Donc le maître `pi_control` va lancer `N` copies d'un esclave `pi` avec l'entier k correspondant, puis attendre que tous aient terminé leur exécution, enfin effectuer la somme de tous les résultats partiels.



Threads JAVA

- Concept répandu mais annexe dans d'autres langages de programmation (C ou C++ par exemple),
- Concept intégré dans JAVA (mais aussi ADA)
- Thread = "Thread of Control", processus léger etc.
- un Thread est un programme séquentiel, avec sa propre mémoire locale, s'exécutant en même temps, et sur la même machine virtuelle JAVA, que d'autres threads
- communication à travers une mémoire partagée (ainsi que des méthodes de synchronisations)

Utilité des threads

- Plus grande facilité d'écriture de certains programmes (extension de la notion de fonction)
- Systèmes d'exploitation
- Interfaces graphiques
- Ordinateurs multi-processeurs (bi-pentium etc.)

Création

Pour créer un thread, on crée une instance de la classe `Thread`,
`Thread Proc = new Thread();`

- on peut configurer `Proc`, par exemple lui associer une priorité,
- on peut l'exécuter en invoquant la méthode `start` du thread,

Exécution

- `start()` va à son tour invoquer la méthode `run` du thread (qui ne fait rien par défaut)
- C'est possible par exemple si on définit `Proc` comme une instance d'une sous-classe de `Thread`, dans laquelle on redéfinit la méthode `run`.
- Les variables locales sont des champs de cette sous-classe.

Complément JAVA: les classes

```
class Point {
  int x; int y = 0;
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; } }
```

La classe Point est constituée de:

- de variables (x), qui peuvent être introduites avec une valeur par défaut.
- de constructeurs (Point), dont le corps est de type void; un constructeur est reconnu au fait que son nom est le même que celui de sa classe et que son type de retour n'est pas indiqué.
- de méthodes (bouge) dont le corps est arbitraire.

Les constructeurs

On crée une instance d'une classe avec la construction new suivie d'un constructeur de classe et de ses arguments, par exemple `new Point (3)`.
Le typage ne garantit pas que toutes les variables sont initialisées. Cependant, le compilateur affecte aux variables non initialisées une valeur par défaut du type déclaré.

This

Dans le corps d'une méthode, this désigne l'objet en train d'exécuter cette méthode.

```
class Point {
  int x = 0;
  Point () { }
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
  void bouge_bouge () { this.bouge(); this.bouge(); }
}
```


L'héritage

```
class Bipoint extends Point {  int y;
  Bipoint () { }
  Bipoint (int x0, int y0) { super(x0); y = y0; }
  void bouge () { super.bouge (); y = y + 1; }
}
```

Dans une sous-classe, super désigne la classe parente. Un constructeur d'une sous-classe doit faire appel à un constructeur de la classe parente. Cet appel doit être la première instruction. Si ce n'est pas fait, le compilateur ajoute super().

La surcharge

- Les constructeurs de classe et les méthodes sont surchargés.
- Les constructeurs de classes ne sont pas hérités.
- Les méthodes sont héritées. Ainsi, l'ensemble des définitions d'une méthode m dans une classe A est l'ensemble des définitions de m dans la classe A et dans la classe parente.
- Lorsqu'une sous-classe redéfinit une méthode présente dans la classe parente avec le même type, la nouvelle définition cache (override) la précédente. Sinon elle ne fait que la surcharger.

Exemple

```
class Compte extends Thread {
   int valeur;

   Compte(int val) {
      valeur = val;
   }

   public void run() {
      try {
```



```
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
         }
      } catch (InterruptedException e) {
           return;
        }
   }

   public static void main(String[] args) {
      new Compte(1).start();
      new Compte(2000).start();
   }
}
```


Exécution

L'exécution du programme **main** donne quelque chose comme,

```
> java Compte
1 2000 2000 1 1 1 1
^C
>
```


Remarques

- Les entiers `valeur` sont distincts dans les deux threads qui s'exécutent. C'est une variable locale au thread.
- Si on avait déclaré `static int valeur`, cette variable aurait été partagée par ces deux threads (durée de vie=celle des threads).
- Si on avait déclaré `Integer valeur` ("classe enveloppante" de `int`), cette classe aurait été partagée par tous les threads et aurait une durée de vie éventuellement supérieure à celle des threads

Complément JAVA: les exceptions

- Les exceptions sont des objets qui sont créés lors d'erreurs, que l'on peut récupérer et traiter dans une section de code dédiée,
- Quand le programme détecte une situation anormale, il lève une exception, "Throw",
- Qui peut être capturée dans un bloc d'essai "try ... ",
- Et récupérée pour traitement dans un bloc "catch(...) ... ".

Arrêt

- méthode `stop()` (dangereux et n'existe plus en JAVA 2),
- méthode `sleep(long)` (suspension du thread pendant un certain nombre de nanosecondes)

Mais...

- Pas d'héritage multiple en JAVA, donc on ne peut pas hériter de `Thread` et d'une autre classe en même temps,
- Il est alors préférable d'utiliser l'interface `Runnable`.

L'interface `Runnable`

- L'interface `Runnable` représente du code exécutable,
- Elle ne possède qu'une méthode, `public void run();`
- Par exemple la classe `Thread` implémente l'interface `Runnable`.
- Mieux encore, on peut construire une instance de `Thread` à partir d'un objet qui implémente l'interface `Runnable` par le constructeur: `public Thread(Runnable target);`

Complément JAVA: Interfaces et implémentations

Les interfaces sont des spécifications des implémentations.

```
interface Printable {
  void print (); }
```

- L'interface `Printable` n'est pas une classe. Elle n'a pas d'implémentation propre. Mais l'interface peut être implémentée (ie. respectée) par différentes classes.
- Une classe peut implémenter plusieurs interfaces.

```
class Printable_point extends Point implements Printable {
  Printable_point () { super(); }
  public void print () { System.out.println(x); } }
```


Utilisation

La déclaration d'une interface I dans la définition d'une classe A:

- effectue des vérifications de typage.
- déclare le type des objets de la classe A un sous-type du type de l'interface I.
- Cela permet par exemple, de voir tous les objets imprimables avec un type commun Printable et de leur envoyer des messages.
- La relation de sous-typage Une classe ne peut hériter que d'une seule classe, mais elle peut simultanément implémenter une ou plusieurs interfaces.
- La relation d'héritage est un arbre, mais la relation de sous-typage est un graphe (treillis)

Le sous-typage

En JAVA, le sous-typage est par nom.

- Une déclaration de classe class A extends B implements B1, ... Bk déclare un nouveau type A, et le déclare sous-type de B et de chaque Bi.
- Une déclaration d'interface interface A extends B1, ... Bk déclare un nouveau type A, et le déclare sous-type de chaque Bi.
- La relation de sous-typage est fermée par reflexivité et transitivité.

Les casts

- Le cast d'un objet change son type statique, mais ne change pas sa représentation (valeur dynamique).
- Un cast se note (A) (v) où A est le type cible et v l'argument.
- Lorsque le type statique de v est un sous-type de A, alors le cast est statique, et il réussit toujours.
- Lorsque le type statique de v est un sur-type de A, alors il est possible que le type dynamique de v soit en fait plus précis et un sous-type de A. Le cast est autorisé, mais il nécessite un test à l'exécution qui peut échouer.
- Tout cast qui peut réussir à l'exécution est permis.

Les conversions

- Un cast (conversions) effectue une conversion de type explicite.
- Certaines conversions de types sont implicites. Par exemple, une affectation d'une valeur à une variable de type A effectue une conversion implicite vers le type A pourvu que la valeur soit un sous-type de B. Les conversions implicites ne peuvent pas échouer à l'exécution.
- Les règles de conversion sont complexes (voir The Java Language Specification pour une description complète...)

Exemple

```
class Compte implements Runnable {
   int valeur;

   Compte(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
         }
```


Exemple

```
      } catch (InterruptedException e) {
           return;
        }
   }

   public static void main(String[] args) {
      Runnable compte1 = new Compte(1);
      Runnable compte2 = new Compte(2000);
      new Thread(compte1).start();
      new Thread(compte2).start();
   }
}
```


Déterminer la terminaison d'un thread

- `isAlive()` renvoie un booléen indiquant si un thread est encore vivant (après que start() se soit exécuter, et avant la fin normale ou imposée par stop()),

Déterminer la terminaison d'un thread

- `void join()` (comme sous Unix) attend la terminaison d'un thread.
- Egalement: `void join(long timeout)` attend au maximum `timeout` millisecondes.
- Par exemple:

```
public class ... extends Thread {
   ...
   public void stop() {
      t.shouldRun=false;
      try {
         t.join();
      } catch (InterruptedException e) {}   } }
```


Nommer les threads

- `void setName(String name)` nomme un thread: utile essentiellement pour le debugging (`toString()` renvoie ce nom)
- `String getName()` renvoie le nom du thread.

Informations sur les threads

- `static Thread currentThread()` renvoie la référence au Thread courant c'est-à-dire celui qui exécute `currentThread()`,
- `static int enumerate(Thread[] threadArray)` place tous les threads existant (y compris le `main()` mais pas le thread ramasse-miettes) dans le tableau `threadArray` et renvoie leur nombre.
- `static int activeCount()` renvoie le nombre de threads.

Exemple

```
class Compte3 extends Thread {
   int valeur;

   Compte3(int val) {
      valeur = val;
   }

   public void run() {
      try {
         for (;;) {
            System.out.print(valeur + " ");
            sleep(100);
         }
```

```
      } catch (InterruptedException e) {
           return; } }

   public static void printThreads() {
      Thread[] ta = new Thread[Thread.activeCount()];
      int n = Thread.enumerate(ta);
      for (int i=0; i<n; i++) {
         System.out.println("Le thread "+ i + " est
                            " + ta[i].getName());
      } }

   public static void main(String[] args) {
      new Compte3(1).start();
      new Compte3(2000).start();
      printThreads(); } }
```


EXÉCUTION

```
% java Compte3
1 2000 Le thread 0 est main
Le thread 1 est Thread-2
Le thread 2 est Thread-3
1 2000 1 2000 1 2000 1 2000 2000
1 1 2000 2000 1 1 2000 2000 1 1
2000 2000 1 1 2000 2000 1 1 2000
2000 1 1 2000 2000 1 1 ^C
%
```