

INF 560
Calcul Parallèle et Distribué
Cours 4

Eric Goubault

CEA, LIST & Ecole Polytechnique

3 février 2014



E. Goubault

EN TD, VOUS AVEZ VU...

Un calcul de π inefficace sur GPU:

- Coût de communication CPU \leftrightarrow GPU prohibitif par rapport au calcul. Solutions possibles:
 - Plus de calcul...(augmenter n)
 - Moins de communications...(faire la somme des sommes partielles sur le GPU: cf. SCAN après)

```

for d:=1 to log2n do
  forall k in parallel do
    if k ≥ 2d then
      x[out][k] := x[in][k - 2d-1] + x[in][k]
    else
      x[out][k] := x[in][k]
  swap(in, out)
    
```

- Communiquer de façon asynchrone avec le calcul (cudaMemcpyAsync, après)
- Partie séquentielle non négligeable restant sur le CPU, cf. loi d'Amdahl, plus tard... (cf. SCAN également)
- Il ne suffit pas d'avoir beaucoup de processeurs pour faire quelque chose d'efficace...



E. Goubault

- CUDA, en mode "PRAM":
 - exemple du scan (sauf de pointeur)
 - attention au modèle mémoire!
- Les synchronisations en CUDA
 - synchronisation intra-block, device/host
 - opérations atomiques en CUDA (pas de façon commode de synchroniser inter-bloc!)
 - application au calcul non typiquement SIMT et au calcul de π , cf. TD)
- Et en JAVA?



E. Goubault

SCAN: IMPLÉMENTATION CUDA (NAIVE)

(scan_kernel.cu)

```

__global__ void scan(float *g_odata, float *g_idata, int n)
{
  // Dynamically allocated shared memory for scan kernels
  extern __shared__ float temp[];
  int thid = threadIdx.x;
  int pout = 0;
  int pin = 1;
  // Cache the computational window in shared memory
  temp[pout*n + thid] = (thid > 0) ? g_idata[thid-1] : 0;
    
```



E. Goubault

- Ce code nous permet de voir une nouveauté en CUDA...
- Jusqu'ici nous n'avons alloué la mémoire partagée statiquement: `__shared__ float x[100];`
- Quand on ne connaît pas avant l'exécution la taille souhaitée: `extern __shared float x[];` le `extern` est obligatoire...
- ...et il y a une subtilité à l'appel du kernel...(plus loin)



AUTRE SUBTILITÉ: `__SYNCTHREADS()`

Barrière de synchronisation:

- Tous les threads *du même bloc* vont s'attendre à ce point de rendez-vous
- avant de continuer leur exécution
- Permet ici de s'assurer que tous les threads ont fait leur calcul en mémoire partagée avant d'écrire le résultat en mémoire globale



```
for (int offset = 1; offset < n; offset *= 2)
{
    pout = 1 - pout;
    pin = 1 - pout;
    __syncthreads();
    temp[pout*n+thid] = temp[pin*n+thid];
    if (thid >= offset)
        temp[pout*n+thid] += temp[pin*n+thid - offset];
}
__syncthreads();
g_odata[thid] = temp[pout*n+thid];
}
```



IMPLÉMENTATION CUDA

(`scan.cu`)

```
int main( int argc, char** argv)
{
    runTest( argc, argv);
    return 1;
}
void runTest( int argc, char** argv)
{
    // initialize the input data on the host to be integer values
    // between 0 and 1000
    for( unsigned int i = 0; i < num_elements; ++i)
        h_data[i] = floorf(1000*(rand()/((float)RAND_MAX)));
    // compute reference solution
    float* reference = (float*) malloc( mem_size);
    computeGold( reference, h_data, num_elements);
}
```



IMPLÉMENTATION CUDA

```
// allocate device memory input and output arrays
float* d_idata;
float* d_odata[3];
cudaMalloc( (void**) &d_idata, mem_size);
cudaMalloc( (void**) &(d_odata[0]), mem_size);
cudaMalloc( (void**) &(d_odata[1]), mem_size);
cudaMalloc( (void**) &(d_odata[2]), mem_size);
// copy host memory to device input array
cudaMemcpy( d_idata, h_data, mem_size, cudaMemcpyHostToDevice);
```



E. Goubault

IMPLÉMENTATION CUDA

```
// setup execution parameters
// Note that these scans only support a single thread-block worth of data,
// but we invoke them here on many blocks so that we can accurately compare
// performance
dim3 grid(256, 1, 1);
dim3 threads(num_threads*2, 1, 1);

unsigned int numIterations = 100;
for (unsigned int i = 0; i < numIterations; ++i)
{
    scan<<< grid, threads, 2 * shared_mem_size >>>
        (d_odata[0], d_idata, num_elements);
}
cudaThreadSynchronize();
...
```



E. Goubault

ALLOCATION DE LA MÉMOIRE PARTAGÉE

- Il faut passer, au moment de l'appel au kernel, la taille à allouer en mémoire partagée par multiprocesseur
- `kernel <<<griddim, blockdim, sharedmemsize>>>`
- Attention, on n'alloue qu'un bloc mémoire contigu comme cela, charge au programmeur de le découper en les données dont il a besoin...



E. Goubault

VISIBILITÉ DES LECTURES ET ÉCRITURES

QUELS SONT LES RÉSULTATS POSSIBLES DU CODE SUIVANT?

```
__device__ int X = 1;

__device__ void writeX() {
    X = 10;
}

__device__ void readX(int *a, int *b) {
    *a = X;
    *b = X;
}
```

Le thread 0 fait `writeX()`, le thread 1 fait `readX()`



E. Goubault

VISIBILITÉ DES LECTURES ET ÉCRITURES

- Avec le modèle mémoire habituel, à *cohérence séquentielle* (ex. JAVA), on a soit
 - *a=1 et *b=1, si le thread 1 s'exécute entièrement avant le thread 0
 - *a=10 et *b=10, si le thread 0 s'exécute entièrement avant le thread 1
 - *a=1 et *b=10, si le thread 1 est interrompu entre ses deux lectures par l'exécution du thread 0



E. Goubault

SOLUTION: MEMORY FENCES

- `void __threadfence_block();`
- `void __threadfence();`
- `void __threadfence_system();`
- ces instructions "forcent" la visibilité de l'écriture/lecture, pour les threads du bloc, de la carte, et du système (plusieurs cartes, si CUDA capability ≥ 2.0) respectivement

SOLUTION

Insérer `thread_fence_block()` entre les deux lectures, dans `readX()` si les deux threads sont dans le même bloc, sinon `thread_fence()` si ils sont dans le même device, sinon `thread_fence_system()`.



E. Goubault

VISIBILITÉ DES LECTURES ET ÉCRITURES

- Avec le modèle mémoire habituel, à *cohérence séquentielle* (ex. JAVA), on a soit
 - *a=1 et *b=1, si le thread 1 s'exécute entièrement avant le thread 0
 - *a=10 et *b=10, si le thread 0 s'exécute entièrement avant le thread 1
 - *a=1 et *b=10, si le thread 1 est interrompu entre ses deux lectures par l'exécution du thread 0
- Avec le modèle mémoire "faible" (multi-coeurs, GPUs...) on peut en plus avoir *a=10 et *b=1, par exemple si la première lecture de X (dans *a) se fait, dans la "vue" du thread 1, après la deuxième (dans *b), même si elle est "déclenchée" après a priori.



E. Goubault

PLUS GÉNÉRALEMENT: SYNCHRONISATION EN CUDA

- Même problème qu'en JAVA, à cause de la mémoire partagée...
- Pas de sémaphores, mais quelques fonctions de synchronisation:
 - intra-bloc: `__syncthreads();`
 - entre le GPU et le CPU (très inefficace: attend la fin de l'appel au kernel): `cudaThreadSynchronize();` (mais aussi les `cudaMemcpy();` par effet de bord)
 - les fonctions "atomiques" (ininterruptibles)
- Ces fonctions atomiques font des opérations "read-modify-write" sans interférence par d'autres threads
- S'appliquent à la mémoire globale et à la mémoire partagée



E. Goubault

QUELQUES FONCTIONS ATOMIQUES CUDA

- `int atomicAdd(int *address, int val);` (ajoute val à la valeur pointée à l'adresse address)
- `int atomicSub(int *address, int val);` (retire val à la valeur pointée à l'adresse address)

Ces deux fonctions retournent les valeurs avant modification.



E. Goubault

AUTRES FONCTIONS ATOMIQUES CUDA

- `unsigned int atomicDec(unsigned int* address, unsigned int val);`
- `int atomicAnd(int* address, int val);`
- `int atomicOr(int* address, int val);`
- `int atomicXor(int* address, int val);`



E. Goubault

AUTRES FONCTIONS ATOMIQUES CUDA

- `int atomicExch(int* address, int val);` stocke val dans address
- `int atomicMin(int* address, int val);`
- `int atomicMax(int *address, int val);`
- `unsigned int atomicInc(unsigned int* address, unsigned int val);` incrémente address de 1 si la valeur stockée précédemment est inférieure ou égale à val



E. Goubault

COMPARE AND SWAP

Fonction très utile:

- `int atomicCAS(int* address, int compare, int val);`
- compare la valeur à l'adresse address à compare et écrit val à l'adresse address seulement si cette valeur est égale à compare
- permet de coder des modifications atomiques de valeurs de variables, sans les sémaphores binaires



E. Goubault

EXEMPLE D'UTILISATION

AMÉLIORATION DU CALCUL DE π

```

__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;

__device__ float calculateTotalSum(float *result) {
// En general, saut de pointeur sur BLOCKSIZE<N threadproc
}

__device__ float calculatePartialSum(const float *array, int N) {
// Saut de pointeur sur N thread proc
}

--global__ void sum(const float* array, unsigned int N, float* result) {
// Chaque bloc calcule une somme partielle de array
float partialSum = calculatePartialSum(array, N);
if (threadIdx.x == 0) {
// Thread 0 de chaque bloc stocke la somme partielle en mem globale
result[blockIdx.x] = partialSum;
// Pour rendre visible au dernier block
__threadfence();
// Thread 0 de chaque bloc signale qu'il a effectue le calcul
unsigned int value = atomicInc(&count, gridDim.x);
// Thread 0 de chaque bloc determine si son bloc est le dernier
isLastBlockDone = (value == (gridDim.x - 1)); }
    
```



E. Goubault

EXEMPLE D'UTILISATION

SUITE

```

// Pour que chaque thread lise la valeur correct de isLastBlockDone
__syncthreads();
if (isLastBlockDone) {
// Dernier bloc somme les sommes dans result[0 .. gridDim.x-1]
float totalSum = calculateTotalSum(result);
if (threadIdx.x == 0) {
// Thread 0 du dernier bloc stocke la somme globale en mem globale
// et reinitialise count pour une nouvelle execution du noyau
result[0] = totalSum;
count = 0; } } }
    
```

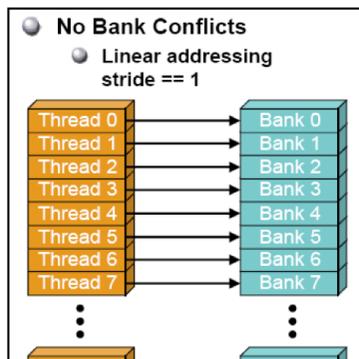


E. Goubault

POUR ALLER PLUS LOIN

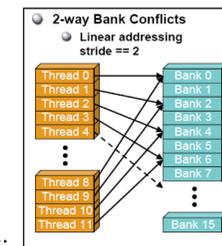
OPTIMISATION ACCÈS MÉMOIRE: "BANK CONFLICT"?

- La mémoire d'une machine parallèle à mémoire partagée est généralement partitionnée en bancs sur lesquelles un seul read ou un seul write (de mots 32 bits ici) est effectué à tout instant (2 cycles en général)
- Cf. PRAM EREW...



E. Goubault

"BANK CONFLICT"?



- Cas de conflit:
- Perte de performance: sérialisation des accès mémoire...
- Cas particulier: tous les threads d'un warp lisent la même adresse en même temps...pas de conflit!
- Ex. les cartes G80:
 - ont 16 banks, faits de mots 32 bits contigus
 - Donc # bank=adresse % 16
 - attention à l'alignement des données! (`__ALIGN(X)`)



E. Goubault

EXEMPLE DE “BANK CONFLICT”

Code standard:

```
--global-- void test(int *gpA)
{
  __shared__ int sa[16];
  sa[0]=3; // bank conflict if blocksize > 1
  gpA[0]=sa[0]; // bank conflict again
}
```



E. Goubault

COMMUNICATION ASYNCHRONE

(à partir d'ici compiler avec `nvcc -arch sm_30` pour avoir accès aux fonctions “avancées”, “CUDA capability 3.0”)

- `cudaMemcpyAsync` pour des copies asynchrones (potentiellement, c.-à-d. non bloquantes):
 - mais il faut verrouiller des bouts de mémoire, et les déverrouiller explicitement (`HostRegister`, `HostUnregister` sur l'hôte, à partir de la version 4 de CUDA)
 - ou alors associer les `cudaMemcpy` à des “CUDA streams”: plusieurs “copy engines” (si l'architecture le permet) permettent de faire plusieurs transferts mémoires en parallèle, et en parallèle de l'exécution. On verra au cours 6/7 l'intérêt algorithmique de ces “recouvrements” communication/calcul.



E. Goubault

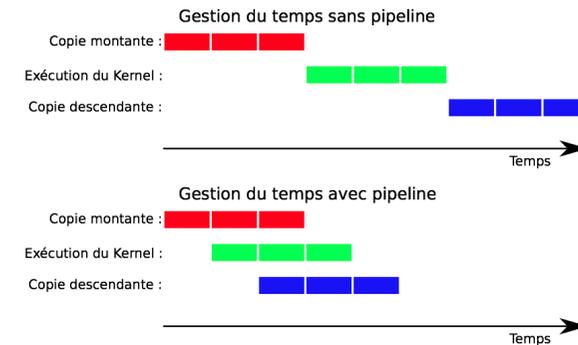
POUR ALLER PLUS LOIN

- `cudaMemcpyToArray(struct cudaArray * dstArray, size_t dstX, size_t dstY, const void * src, size_t count, enum cudaMemcpyKind kind)`
- `cudaMemcpy2D(void * dst, size_t dpitch, const void * src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)`
- Depuis la carte pour allouer dans la mémoire du CPU: `cudaMallocHost(void **pointer, size_t size)`, `cudaFreeHost`



E. Goubault

COMMUNICATION ASYNCHRONE (SUITE)



- Implémentés au niveau logique par les streams: permettent de s'assurer de l'ordre dans lequel on lancera la copie de l'entrée du programme, puis le lancement du kernel, puis la recopie vers la mémoire hôte.



E. Goubault

EXEMPLE DE STREAM

```
// Create Streams
cudaStream_t streamA, streamB;
cudaStreamCreate(&streamA);
cudaStreamCreate(&streamB);

// Create Event
cudaEvent_t sync;
cudaEventCreate(&sync);

// Uploads
cudaMemcpy(pDeviceA, pHostA, sizeInByteA, cudaMemcpyHostToDevice, streamA);
cudaMemcpy(pDeviceA, pHostA, sizeInByteA, cudaMemcpyHostToDevice, streamB);

// Kernels
kernelA<<<BpG, TpB, 0, streamA>>>(pDeviceA);
kernelB<<<BpG, TpB, 0, streamB>>>(pDeviceB);

// Downloads
cudaMemcpy(pHostA, pDeviceA, sizeInByteA, cudaMemcpyDeviceToHost, streamA);
cudaMemcpy(pHostB, pDeviceB, sizeInByteB, cudaMemcpyDeviceToHost, streamB);
```



E. Goubault

CUDA 5.5/6.0 - CUDA CAPABILITY 3.5

Sur les cartes les plus récentes (Tesla), on a des possibilités encore bien meilleures:

- Communication directe entre la mémoire de plusieurs device (et espace d'adressage virtuel pour les devices, et le host)
- Parallélisme dynamique: création dynamique de processus, avec des grilles propres...

```
--global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

--global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```



E. Goubault

POUR ALLER PLUS LOIN: ÉVÉNEMENTS

CUDA EVENT

- Attaché à un CUDA stream
- Intérêt: opération de synchronisation permettant de faire attendre le CPU jusqu'à ce que l'évènement (event) ait été exécuté par le stream.
- `cudaEventCreate`, `cudaEventRecord` pour l'attacher à un stream, `cudaEventSynchronize` pour synchroniser
- Peut être une synchronisation globale: il faut l'attacher au contexte d'exécution CUDA qui est le stream 0

```
// Synchronize on CUDA context
cudaEventRecord(sync, 0);
cudaEventSynchronize(sync);
```

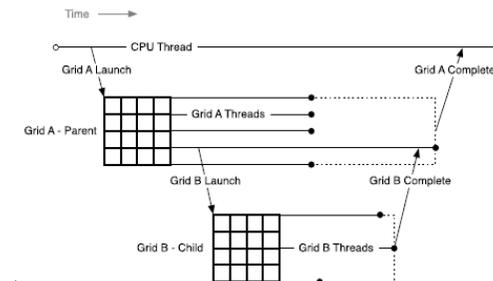
```
// Destroy Event
cudaEventDestroy(sync);
```

```
// Destroy Streams
cudaStreamDestroy(streamA);
cudaStreamDestroy(streamB);
```



E. Goubault

CUDA 5.5/6.0 - CUDA CAPABILITY 3.5



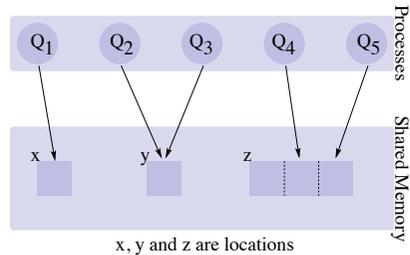
POUR EXPÉRIMENTER... (ET LES PROJETS EN CUDA):

- Machine du LIX: 2 device chacun de 2496 coeurs
- Accessible depuis les salles TD à l'IP 129.104.252.237
- Me montrer un programme CUDA qui marche puis m'envoyer un mail pour avoir accès (ensuite ce sera login et mot de passe LDAP)



E. Goubault

CAS PARTICULIER DE PRIMITIVES D'EXCLUSION MUTUELLE



PRIMITIVES CLASSIQUES (DIJKSTRA 1968...)

- Sémaphore binaires ou mutex, ou verrou: permettant l'accès exclusif à des ressources partagées. Résoud le problème d'exclusion mutuelle (synchronized JAVA).
- Généralisation: sémaphores à compteur (verrou pouvant être détenu par jusqu'à n processus mais pas $n + 1$).
- Moniteurs: mécanisme de signalisation élémentaire.



E. Goubault

LA BANQUE...

```
public class Banque implements Runnable {
    Compte nom;

    Banque(Compte n) {
        nom = n;
    }

    public void Liquide (int montant)
        throws InterruptedException {
        if (nom.retirer(montant)) {
            Thread.currentThread().sleep(50);
            Donne(montant);
            Thread.currentThread().sleep(50);
            ImprimeRecu();
            Thread.currentThread().sleep(50);
        }

    public void Donne(int montant) {
        System.out.println(Thread.currentThread().
        getName()+"Voici_vous_" + montant + "_euros.");
    }

    public void ImprimeRecu() {
        if (nom.solde() > 0)
            System.out.println(Thread.currentThread().
            getName()+"_Il_vous_reste_" + nom.solde() + "_euros.");
        else
            System.out.println(Thread.currentThread().
            getName()+"_Vous_etes_fauchés!");
    }
}
```



E. Goubault

SÉMAPHORES? REVENONS AU JAVA ET AUX COMPTES...

```
public class Compte {
    private int valeur;

    Compte(int val) {
        valeur = val;
    }

    public int solde() {
        return valeur;
    }

    public void depot(int somme) {
        if (somme > 0)
            valeur+=somme;
    }

    public boolean retirer(int somme)
        throws InterruptedException {
        if (somme > 0)
            if (somme <= valeur) {
                Thread.currentThread().sleep(50);
                valeur -= somme;
                Thread.currentThread().sleep(50);
                return true;
            }
        return false;
    }
}
```



E. Goubault

```
public void run() {
    try {
        for (int i=1;i<10;i++) {
            Liquide(100*i);
            Thread.currentThread().sleep(100+10*i);
        }
    } catch (InterruptedException e) {
        return;
    }
}

public static void main(String[] args) {
    Compte Commun = new Compte(1000);
    Runnable Mari = new Banque(Commun);
    Runnable Femme = new Banque(Commun);
    Thread tMari = new Thread(Mari);
    Thread tFemme = new Thread(Femme);
    tMari.setName("Conseiller_Mari");
    tFemme.setName("Conseiller_Femme");
    tMari.start();
    tFemme.start();
}
}
```



E. Goubault

UNE EXÉCUTION

```
% java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Femme: Voici vos 300 euros.
Conseiller Femme: Vous etes fauches!
Conseiller Mari: Vous etes fauches! ...
```

RÉSULTAT...

- Le mari a retiré 600 euros du compte commun,
- La femme a retiré 600 euros du compte commun,
- qui ne contenait que 1000 euros au départ!



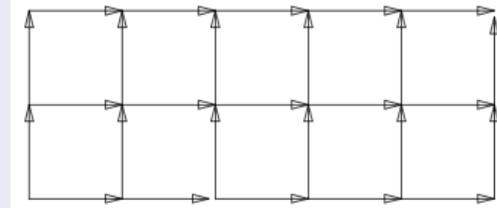
E. Goubault

EXPLICATION

("Sémantique")

- L'exécution de plusieurs threads se fait en exécutant une action insécable ("atomique") de l'un des threads, puis d'un autre ou d'éventuellement du même etc.
- Tous les "mélanges" possibles sont permis

SÉMANTIQUE PAR ENTRELACEMENTS



E. Goubault

EXPLICATION ET SOLUTION

EXPLICATION

Si les 2 threads `tMari` et `tFemme` sont exécutés de telle façon que dans `retirer`, chaque étape soit faite en même temps:

- Le test pourra trouvé être satisfait par les deux threads en même temps,
- qui donc retireront en même temps de l'argent.

UNE SOLUTION

- Rendre "atomique" le fait de retirer de l'argent,
- Se fait en déclarant "synchronisée" la méthode `retirer` de la classe `Compte`:

```
public synchronized boolean retirer(int somme)
```



E. Goubault

MAINTENANT...

```
% java Banque
Conseiller Mari: Voici vos 100 euros.
Conseiller Mari: Il vous reste 800 euros.
Conseiller Femme: Voici vos 100 euros.
Conseiller Femme: Il vous reste 800 euros.
Conseiller Mari: Voici vos 200 euros.
Conseiller Mari: Il vous reste 400 euros.
Conseiller Femme: Voici vos 200 euros.
Conseiller Femme: Il vous reste 400 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Voici vos 300 euros.
Conseiller Mari: Il vous reste 100 euros.
Conseiller Femme: Il vous reste 100 euros.
Conseiller Mari: Il vous reste 100 euros...
```

RÉSULTAT...

- Le mari a tiré 600 euros,
- La femme a tiré 300 euros,
- et il reste bien 100 euros dans le compte commun.



E. Goubault

REMARQUE

- `synchronized` qualifie une méthode, mais est en fait un verrou au niveau de l'objet sur lequel s'applique la méthode
- dans le cas d'une méthode `static`, le verrou s'applique à la classe (i.e. toutes ses instances sont verrouillées)



E. Goubault

PLUS PRÉCISÉMENT

- Chaque objet JAVA `o` comporte une liste de threads en attente
- un thread y rentre en exécutant `o.wait()`
- un thread en sort si un `o.notify()` est exécuté par un autre thread, et que ce soit celui-ci dans la liste qui est libéré (en général, c'est le premier en attente qui est libéré)
- on doit absolument protéger l'accès à cette liste d'attente par un `synchronized`, le plus sûr étant un `synchronized(o)` (mais pas forcément, ce peut être sur la méthode faisant le `wait()` par exemple, si bien étudié...)



E. Goubault

MONITEURS: `wait()` ET `notify()` EN JAVA

Chaque objet fournit un verrou, mais aussi un mécanisme de mise en attente (forme primitive de communication inter-threads; similaire aux variables de conditions ou aux moniteurs):

- `void wait()` attend l'arrivée d'une condition sur l'objet sur lequel il s'applique (en général `this` - mais pas seulement! voir après). Doit être appelé depuis l'intérieur d'une méthode ou d'un bloc `synchronized`, (il y a aussi une version avec `timeout`): on y revient aussi après!
- `void notify()` notifie un thread en attente d'une condition, de l'arrivée de celle-ci. De même, dans `synchronized`.
- `void notifyAll()` même chose mais pour tous les threads en attente sur l'objet.



E. Goubault

UN EXEMPLE (FAUX...)

Considérons :

```
class buffer1 {
    Object data = null;

    public synchronized void push(Object d) {
        try { if (data != null) wait();
        } catch (Exception e) { System.out.println(e); return; }
        data = d;
        System.out.println("Pushed_" + data);
        try { if (data != null) notify();
        } catch (Exception e) { System.out.println(e); return; } }

    public Object pop() {
        try { if (data == null) wait();
        } catch (Exception e) { System.out.println(e); return null; }
        Object o = data;
        System.out.println("Read_" + o);
        data = null;
        try { if (data == null) notify();
        } catch (Exception e) { System.out.println(e); return null; }
        return o; } }
```



E. Goubault

LES THREADS PRODUCTEUR/CONSOMMATEUR

```
class Prod extends Thread {
    buffer1 buf;

    public Prod(buffer1 b) {
        buf = b; }

    public void run() {
        while (true) {
            buf.push(new Integer(1)); } } }

class Cons extends Thread {
    buffer1 buf;

    public Cons(buffer1 b) {
        buf = b; }

    public void run() {
        while (true) {
            buf.pop(); } } }
```



E. Goubault

LE MAIN...

Construit 1 producteur et 2 consommateurs qui se partagent un buffer:

```
public class essaimon0 {
    public static void main(String[] args) {
        buffer1 b = new buffer1();
        new Prod(b).start();
        new Cons(b).start();
        new Cons(b).start(); } }
```



E. Goubault

CORRECTION...

PREMIÈRE ERREUR...

```
is010046:Cours04new Eric$ javac essaimon0.java
is010046:Cours04new Eric$ java essaimon0 | more
Pushed 1
Read 1
java.lang.IllegalMonitorStateException from pop-notify
java.lang.IllegalMonitorStateException from pop-wait
java.lang.IllegalMonitorStateException from pop-wait
...
```

Pas de `synchronized` dans la méthode `pop()`: non repéré à la compilation, mais à l'exécution!



E. Goubault

On rajoute `synchronized` dans la méthode `pop()`, puis:

```
is010046:Cours04new Eric$ javac essaimon0.java
is010046:Cours04new Eric$ java essaimon0 | more
Pushed 1
Read 1
Pushed 1
Read 1
Pushed 1
Read 1
Read null
Read null
...
```

Quel est le problème?



E. Goubault

- wait() et notify() s'appliquent ici sur this qui est toujours le même et unique buffer
- Supposons que data=null (pas de push par exemple, ou un pop effectué)
- Quand le premier consommateur exécute pop(), il voit data=null et fait wait() : **il suspend sa prise de verrou sur le buffer**
- Le deuxième consommateur peut donc s'exécuter, voit data=null et fait wait() : **il suspend sa prise de verrou sur le buffer**
- Le producteur n'est pas bloqué, voit data=null et fait data=1 puis notify, débloquent ainsi l'un des deux consommateurs, disons le premier
- Le premier consommateur lit alors 1 puis fait data=null et notify() et termine, libérant son verrou sur le buffer
- Le deuxième consommateur est alors débloqué et fait read null...



E. Goubault

EXÉCUTION

```
is010046:Cours04new Eric$ javac essaimon1.java
is010046:Cours04new Eric$ java essaimon1 | more
Pushed 1
Read 1
Pushed 1
Read 1
Pushed 1
Read 1
...
```

(correct!)



E. Goubault

UNE SOLUTION POSSIBLE...

Utiliser deux objets, associé au buffer, qui signale "vide" et un autre qui signale "plein" (variables de conditions...):

```
Object full = new Object();
Object empty = new Object();
Object data = null;
public void push(Object d) {
    synchronized(full) {
        try { if (data != null) full.wait();
        } catch (Exception e) { System.out.println(e); return; } }
    data = d;
    System.out.println("Pushed_" + data);
    synchronized(empty) {
        try { if (data != null) empty.notify();
        } catch (Exception e) { System.out.println(e); return; } }
}

public Object pop() {
    synchronized(empty) {
        try { if (data == null) empty.wait();
        } catch (Exception e) { System.out.println(e); return null; } }
    Object o = data;
    System.out.println("Read_" + o);
    data = null;
    synchronized(full) {
        try { if (data == null) full.notify();
        } catch (Exception e) { System.out.println(e); return null; } }
    return o; } }
```



E. Goubault

SÉMAPHORES

(en fait déjà dans java.util.concurrent.Semaphore)

```
public class Semaphore {
    int n;
    String name;

    public Semaphore(int max, String S) {
        n = max;
        name = S;
    }

    public synchronized void P() {
        while (n == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {}
        }
        n--;
        System.out.println("P(" + name + ")");
    }

    public synchronized void V() {
        n++;
        System.out.println("V(" + name + ")");
        notify();
    }
}
```

Bien sûr ici, this.wait et this.notify sont ce que l'on souhaite... Remarquez le while (n==0) dans void P().



E. Goubault

SECTION CRITIQUE

```
public class essaiPV extends Thread {
    static int x = 3;
    Semaphore u;

    public essaiPV(Semaphore s) {
        u = s;
    }

    public void run() {
        int y;
        // u.P();
    }
}
```



E. Goubault

RÉSULTATS

SANS P, V

```
% java essaiPV
Thread-2: x=4
Thread-3: x=4
```

AVEC P, V

```
% java essaiPV
P(X)
Thread-2: x=4
V(X)
P(X)
Thread-3: x=5
V(X)
```



E. Goubault

```
try {
    Thread.currentThread().sleep(100);
    y = x;
    Thread.currentThread().sleep(100);
    y = y+1;
    Thread.currentThread().sleep(100);
    x = y;
    Thread.currentThread().sleep(100);
} catch (InterruptedException e) {}
System.out.println(Thread.currentThread().
    getName()+" : _x="+x);
// u.V();
}
public static void main(String[] args) {
    Semaphore X = new Semaphore(1,"X");
    new essaiPV(X).start();
    new essaiPV(X).start();
}
}
```



E. Goubault

RESSOURCES PARTAGÉES ET PRIORITÉS

Attention, il est possible de tomber dans le problème d'inversion de priorité:

- Soit T_1 une tâche prioritaire par rapport à T_2 , a et b deux objets partagés par T_1 et T_2 (protégés par exemple par deux sémaphores de même nom)
- Supposons l'exécution suivante:

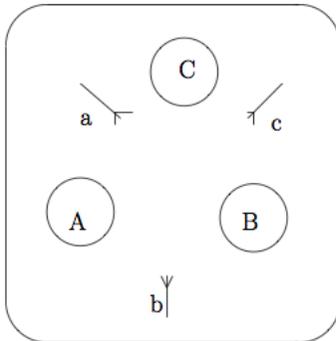
T_1	T_2
Pa	-
(bloqué)	Pb
(obtient le verrou a)	(obtient le verrou b)
Pb	-
(bloqué)	-

- Ainsi, c'est T_2 qui s'exécute alors qu'il est de priorité moindre...



E. Goubault

POINTS MORTS ET PHILOSOPHES QUI DINENT



```
public class Phil extends Thread {
    Semaphore LeftFork;
    Semaphore RightFork;

    public Phil(Semaphore l, Semaphore r) {
        LeftFork = l;
        RightFork = r;
    }
}
```



E. Goubault

```
public void run() {
    try {
        Thread.currentThread().sleep(100);
        LeftFork.P();
        Thread.currentThread().sleep(100);
        RightFork.P();
        Thread.currentThread().sleep(100);
        LeftFork.V();
        Thread.currentThread().sleep(100);
        RightFork.V();
        Thread.currentThread().sleep(100);
    } catch (InterruptedException e) {}
}

}

public class Dining {
    public static void main(String[] args) {
        Semaphore a = new Semaphore(1, "a");
        Semaphore b = new Semaphore(1, "b");
        Semaphore c = new Semaphore(1, "c");
        Phil Phil1 = new Phil(a, b);
        Phil Phil2 = new Phil(b, c);
        Phil Phil3 = new Phil(c, a);
        Phil1.setName("Kant");
        Phil2.setName("Heidegger");
        Phil3.setName("Spinoza");
        Phil1.start();
        Phil2.start();
        Phil3.start();
    }
}
```



E. Goubault

RÉSULTAT

```
% java Dining
Kant: P(a)
Heidegger: P(b)
Spinoza: P(c)
^C
```

INTERBLOCAGE

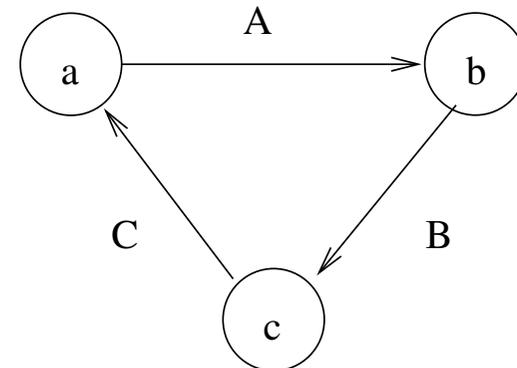
- Une exécution possible n'atteint jamais l'état terminal: quand les trois philosophes prennent en même temps leur fourchette gauche.
- Peut se résoudre avec un "ordonnanceur" extérieur (exercice classique).
- On pouvait s'en apercevoir sur le "request graph"... ou le "progress graph" (voir poly)



E. Goubault

REQUEST GRAPH

Pa.Pb.Vb.Va|Pb.Pa.Va.Vb|Pc.Pa.Va.Vc:



E. Goubault