Programmer ses applications en Multicore

2023

$ /usr/bin/whoami

- Sébastien VINCENT
- Consultant et formateur indépendant
- Github : https://github.com/s-vincent
- E-mail : sebastien@vincent-netsys.fr

Fonctionnement du cours

- Concentration / focus
- **Poser des questions**
  - si les explications ne sont pas claires
  - pour approfondir

Objectifs / pré-requis de la session

- Objectifs
  - Maîtriser les enjeux de la programmation Multicore
  - Concevoir et développer des applications à base de threads et de processus
  - Maîtriser les modèles de programmation parallèle et les librairies disponibles
  - Déboguer et profiler des applications Multicore
- Pré-requis
  - Bonnes connaissances de C ou de C++
  - Connaissances de base des concepts liés aux applications Multicore

Enjeux de la programmation multicore

- Aujourd'hui nos ordinateurs sont *extrêmement* puissants
- Un ordinateur de supermarché dispose de 2 à 4 coeurs
- Les processeurs haut de gamme peuvent aller jusqu'à 64 coeurs
- Certains serveurs d'entreprise ont parfois plusieurs processeurs...

Enjeux de la programmation multicore (2)

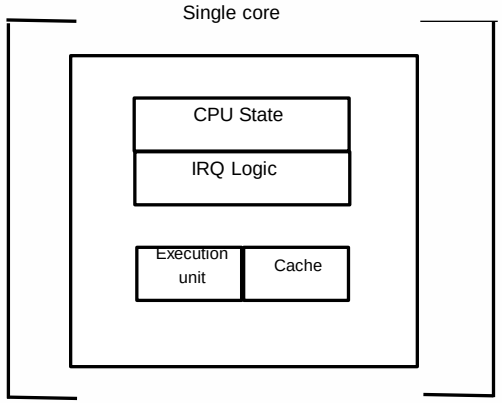
- Malheureusement toute cette puissance n'est pas utilisée à sa juste valeur
- La plupart des applications classiques ne sont pas prévues pour
- Les applications *optimisées* pour un grand nombre de CPU sont entres autres :
  - les jeux vidéos
  - encodage / décodage vidéo
  - calcul scientifique
  - *machine learning* et Intelligence Artificielle
  - *high frequency trading*
  - ...

Architectures processeur

- On trouve plusieurs architectures :
  - mono-processeur
  - *SMT*
  - multi-processeurs
  - multi-coeurs
- Un processeur physique peut contenir un à plusieurs coeurs
- Un coeur peut disposer du SMT (*hyperthreading* chez Intel)

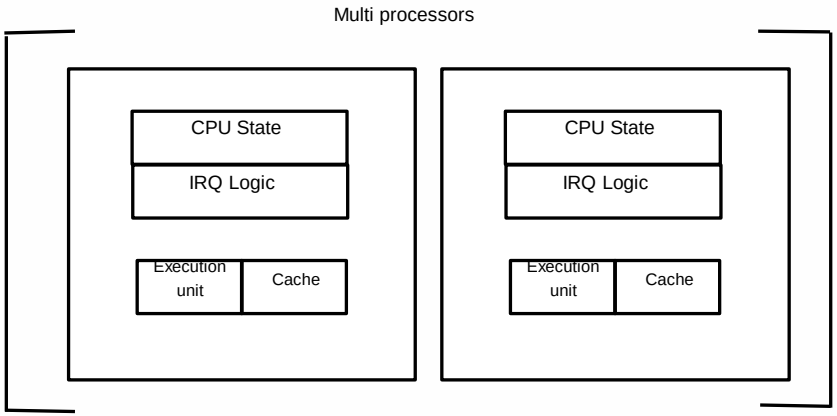
Mono-processeur

- Un unique processeur (à 1 coeur)
- Modèle classique... de moins en moins répandu sur les PC



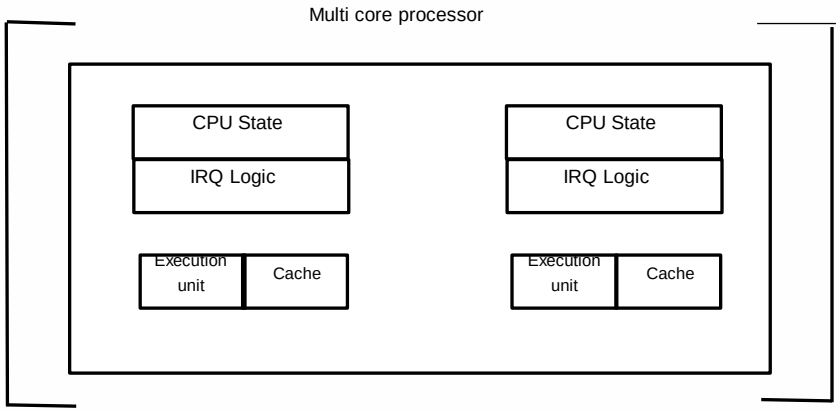
Multi-processeurs

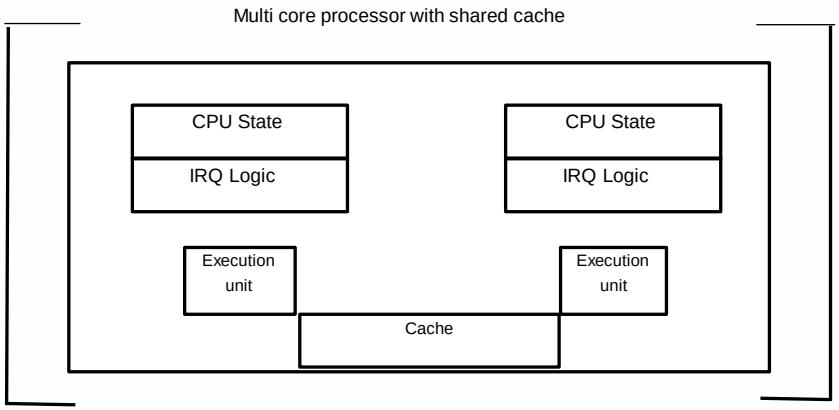
- La carte mère contient plusieurs sockets de processeurs physiques
- Fréquemment répandu dans les serveurs professionnels
- Ces processeurs peuvent être multi-coeurs
- Architecture NUMA et SMP



Processeur multi-coeurs

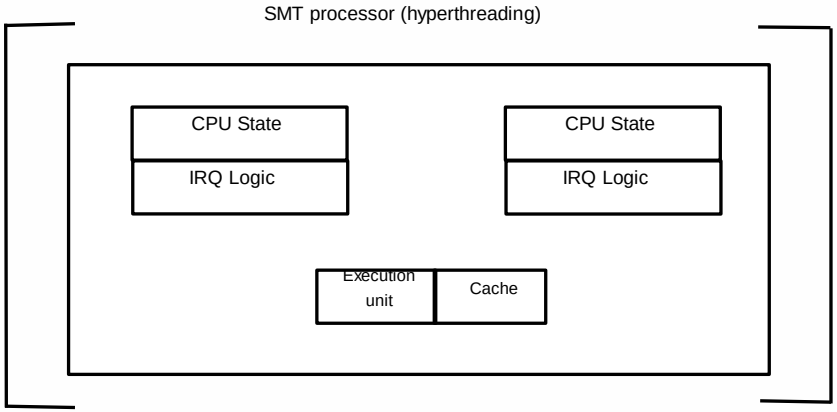
- Architecture actuelle (et ça arrive aussi dans l'embarqué)
- Introduction de plusieurs CPU sur la même puce
- Un CPU X-core est vu comme X CPU virtuels par le système
- Chacun des coeurs peut aussi disposer du SMT
- ... ce qui double encore le nombre de CPU virtuels vu par le système





Simultaneous Multi Threading (SMT)

- *Hyperthreading* : nom commercial d'Intel
- Un coeur est vu comme **deux** CPU virtuels par le système
- Un système d'exploitation verra un processeur à 4 coeurs SMT comme 8 CPU virtuels
- Duplication du CPU state et interrupt logic, partage de l'unité d'exécution et du cache
- Attention :
  - pas forcément intéressant : https://www.agner.org/optimize/blog/read.php?i=6
  - le SMT faciliterait les attaques par canaux auxiliaires visant le CPU (attaques *Spectre*, *Meltdown*, ...)

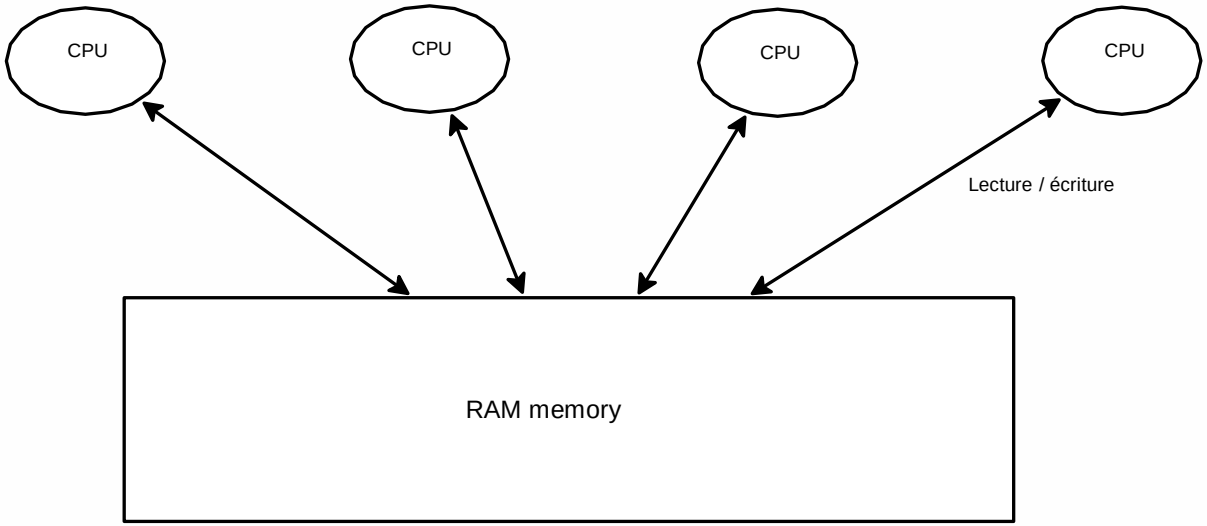


Architecture matérielle

- Traitement concurrent peut se faire sur :
  - le nombre d'instruction en parallèle
  - la dimension des données
- Taxonomie de Flynn
- En 1966, Flynn a défini quatre types :
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple data (SIMD)
  - Multiple Instruction Single Data (MISD)
  - Multiple Instruction Multiple Data (MIMD)

Architecture mémoire partagée

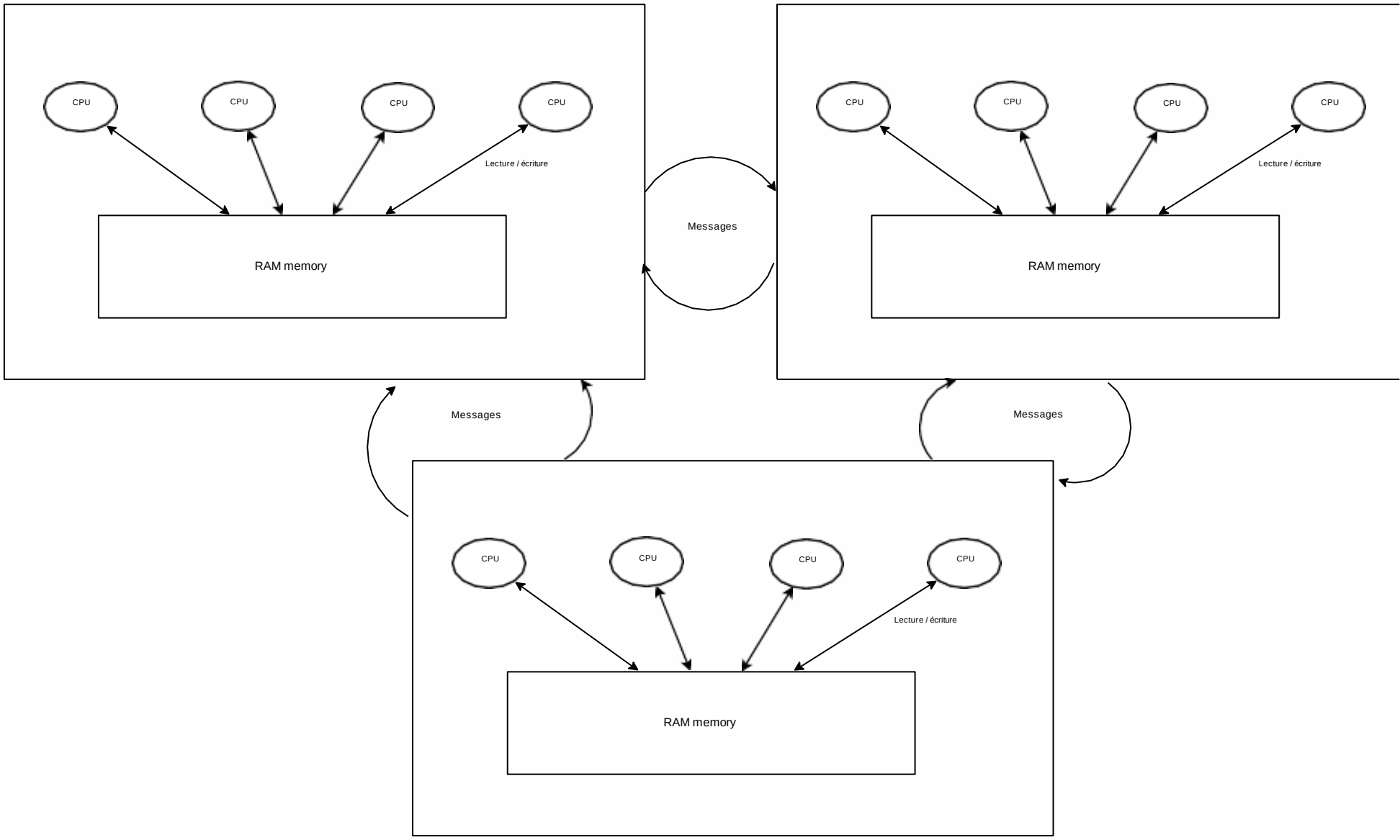
- C'est l'architecture utilisée dans un PC classique
- Tous les processeurs ont accès à la mémoire (RAM)
- Mise en oeuvre du parallélisme simple
- Attention aux conflits d'accès à la RAM (synchronisation, cache, ...)
- Technologies : threads, processus / IPC, OpenMP



Architecture mémoire distribuée

- C'est l'architecture du *grid computing* (modèle multi-machine)
- Chaque processeur dispose de sa mémoire dédiée
- Chaque machine va communiquer avec les autres (envoi de données, résultats, ...)
- Mise en oeuvre complexe car il y a les mécanismes de communication à gérer
- Par contre, il n'y a aucun conflit d'accès à la RAM (sauf s'il y a parallélisation à mémoire partagée sur chaque machine)
- Technologies : Message Passing Interface

Architecture mémoire distribuée (2)



Architecture carte graphique

- Utilisation des capacités de calcul des cartes graphiques (GPU)
- Capacité supérieure au CPU pour **certains** cas d'utilisation
  - Calcul matriciel
  - Calcul cryptographique
- Usages : calcul haute performance, *machine learning*, minage de cryptomonnaie (Bitcoin, Ethereum, ...)
- Technologies : OpenCL, CUDA (NVidia), AMD APP, ROCm (AMD)

Système d'exploitation à temps partagé

- Un système à temps partagé permet de simuler l'exécution de programmes en parallèle
- Chaque programme se voit attribuer un quantum de temps d'exécution sur le ou les processeurs
- Implémentations :
  - Système d'exploitation coopératif
  - Système d'exploitation préemptif

Système d'exploitation coopératif

- Les applications sont responsables de *passer la main* aux autres
- Gare aux applications buggées, boucles infinies, ...
- Exemples :
  - Windows 1.x -> 3.11

Système d'exploitation préemptif

- Le système d'exploitation s'occupe de gérer quel programme est exécuté, qui sera le prochain, ...
  - Concrêtement c'est le travail de l'ordonnanceur (*scheduler*)
- Exemples :
  - Systèmes d'exploitations modernes (Windows NT4+, GNU/Linux, *BSD, ...)

Ordonnanceur

- Le noyau du système d'exploitation dispose d'un ordonnanceur pour gérer l'exécution des processus sur le(s) processeur(s)
- L'ordonnanceur exécute un algorithme équitable pour placer les processus sur le processeur tenant compte de leur nombre et des priorités
- Un processus dispose d'un quantum de temps d'exécution sur le CPU
- Quand un processus a épuisé son temps, on passe au processus suivant
- Ce changement est appelé la commutation de contexte (*context switch*)

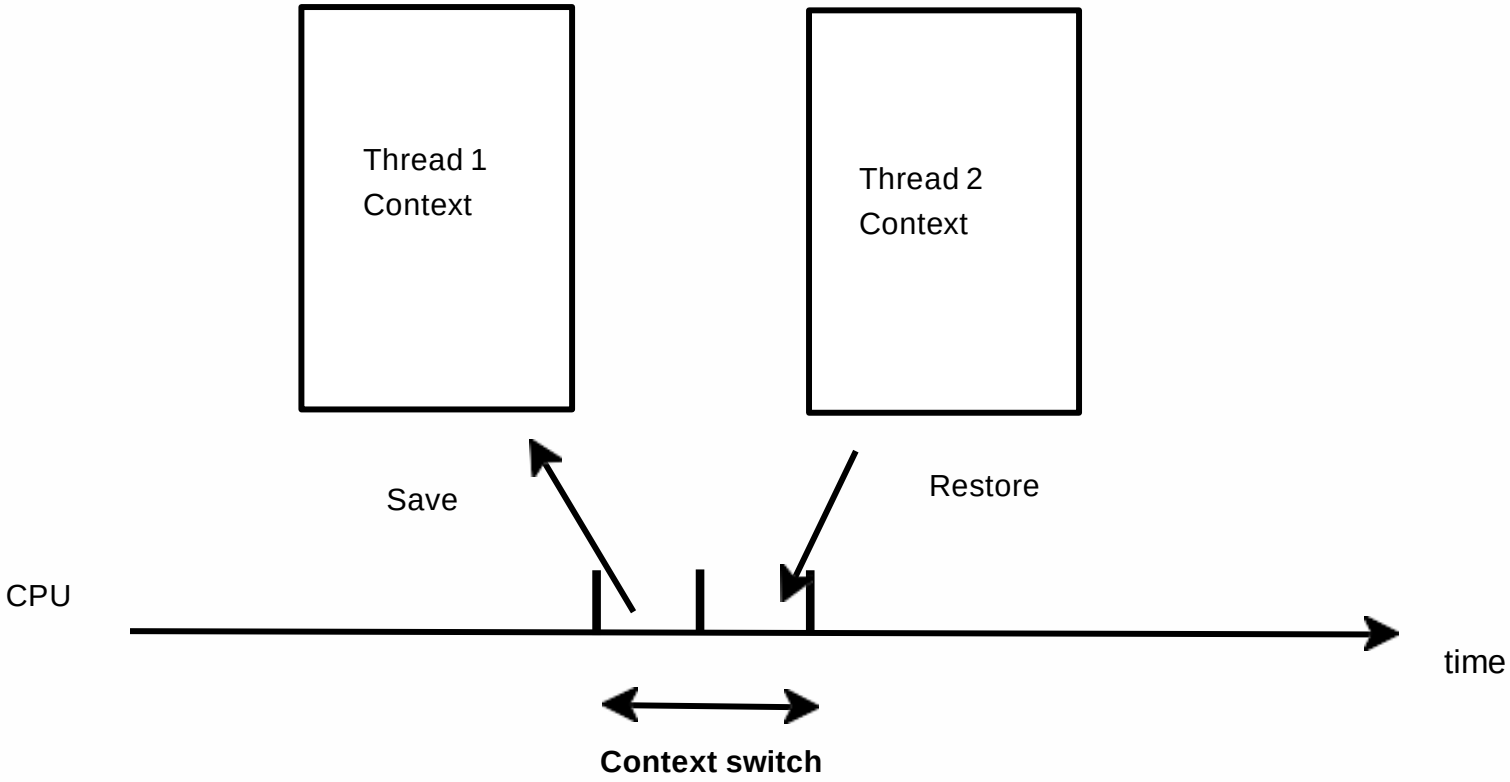
Ordonnanceur (2)

- Certains évènements peuvent endormir un processus avant la fin de son quantum de temps
  - appels systèmes
  - attente de condition / mutex
  - Préemption par un processus plus prioritaire
- Certains évènements peuvent réveiller un processus
  - retour d'appels systèmes
  - interruption (IRQ)
  - exception
  - condition signalée / mutex libéré

Ordonnanceur (3)

- La commutation de contexte :
  - met en pause le processus P1 en cours
  - sauvegarde l'état du processus P1 (pile d'appel, registre, ...)
  - charge les informations du processus P2 à exécuter (pile, ...) ainsi que son espace d'adressage mémoire
  - reprise exécution du processus P2
- Le temps d'une commutation a un coût et est incompressible !
- Une commutation peut se produire n'importe où dans le code (assembleur) d'un processus
- Attention : certaines fonctions d'assignations / opérations peuvent prendre plusieurs instructions assembleurs !

Ordonnanceur (4)



Formation Programmer ses applications en Multicore

Parallélisation ou concurrence

- Il faut se poser la question si nous voulons un traitement parallèle ou un traitement concurrent !
- Parallélisation : répartition du travail en plusieurs parties
  - calcul, map-reduce, ...
- Concurrence : opérations sur des entités indépendantes
  - serveur réseau, *dispatcher*, interface graphique, ...
- Selon le cas, la limite entre les deux peut être très mince...

Programmation concurrente : briques logicielles

- Multiprocessus
- Multithreading
  - API POSIX (pthreads)
  - API Windows
  - Standard C11 / C17 / C18
  - Standard C++11 / C++14 / C++17 / C++20
  - Librairie boost::thread (C++)
- Note : il existe une librairie qui implémente une grande partie de l'API threads POSIX en utilisant l'API Windows (voir https://sourceware.org/pthreads-win32/conformance.html)
- Les threads C11 ne sont pas supportés par Visual Studio mais il y a une émulation (voir https://gist.github.com/yohhoy/2223710)

Programmation parallèle : briques logicielles

- Extensions basées sur les threads / processus :
  - OpenMP
  - Message Passing Interface
  - Autres (Intel TBB, Microsoft Parallel FX, ...)
- GPGPU
  - OpenCL
  - CUDA (NVidia)
  - AMD APP / ATI Stream / ROCm (AMD)
  - OpenACC

Programmation parallèle : loi d'Amdahl

- Facteur d'augmentation de performance par rapport à un programme exécuté séquentiellement
- Loi : $S = \frac{1}{1-p+\frac{p}{n}}$
- S : augmentation des performances
- p : pourcentage d'activités parallélisables
- n : nombre de coeurs
  - Exemple : programme tournant sur quatre coeurs dont 95% des activités sont parallélisables :
  - $S = \frac{1}{1-0.95+\frac{0.95}{4}} = 3,47 => 3,47$ plus de performances

Programmation parallèle : loi d'Amdahl (2)

- Limite théorique : $S = \frac{1}{1-p}$
  - Exemple : programme tournant sur un nombre de processeurs infini dont 95% des activités sont parallélisables :
  - $S = \frac{1}{1-0.95} = 20 => 20x$ plus de performance au maximum !
- Ce modèle est théorique et ne prends pas en compte le temps d'attente, de synchronisation, la commutation de contexte, ...

Multiprocessus ou multithread ou GPU ?

- Thread :
  - + création rapide
  - + commutation rapide
  - - programmation plus complexe / synchronisation
  - - si un thread crash, tout le programme crash
- Processus :
  - + isolation entre processus
  - + accès à plus grande zone mémoire
  - - création lente
  - - commutation et IPC couteuses
- GPGPU :
  - + Performance excellente pour des cas spécifiques
  - - Programmation et mise en oeuvre complexe
  - - Cas d'utilisation très restreint (calcul, matrices, crypto, IA, ...)

Outils

- GNU/Linux, *BSD : htop, valgrind / helgrind, gdb, gprof, taskset, hwloc-ls, lscpu, cat /proc/cpuinfo
- Windows : taskmgr.exe, performance monitor, Process Explorer, Profiler Visual Studio, ProcDump, WinDbg, DebugDiag2

Formation Programmer ses applications en Multicore

Les threads

- Aussi appelé processus léger
- Un processus contient au minimum 1 thread et peut en avoir plusieurs
- Contrairement aux processus, les threads partagent le même espace d'adressage : plus besoin d'IPC !
- La pile d'appel, les registres ainsi que les variables globales de type *thread local storage*) sont propres au thread
- Attention aux accès / modification d'une zone mémoire par plusieurs threads !
- Il y a toujours besoin de synchronisation
- La commutation de contexte entre thread est plus rapide car il n'y a pas de changement de l'espace d'adressage

Les threads (2)

- Nécessite de la rigueur pour l'implémentation afin d'éviter les problèmes de synchronisation :
  - Accès à une même zone mémoire en parallèle
  - Situation de compétition (*race condition*)
  - Interblocage (*deadlock*)
  - Interblocage actif (*livelock*)
  - Opération non-atomique
  - Inversion de priorité
- Sur certains systèmes (GNU/Linux), un thread est vu comme un processus (avec un PID, ...) mais la création et la gestion est moins lourde (utilise l'appel système clone() au lieu de fork())

Les threads (3)

- La programmation *multi-thread* est plus compliquée que l'équivalent séquentiel.
- Il faut veiller à rendre son code *thread-safe*...
- ... ou indiquer dans la documentation qu'il n'est pas *thread-safe* (alors c'est à l'appelant de gérer cela)

Types de threads

- Au niveau de la machine, il y a les *hardware threads*
  - unité physique de parallélisation
  - CPU X-core : X *hardware threads*
- Au niveau du noyau, il y a les *kernel threads*
  - Le *kernel thread* sera ensuite exécuté sur l'un des *hardware threads* déterminé par l'ordonnanceur
  - L'unité pour l'ordonnanceur du noyau est le *kernel thread* (et non le processus)
  - Pour un processus disposant d'un certain quantum de temps, tous ses threads auront le même quantum de temps !
- Au niveau applicatif, il y a les *user threads*

Modèles de threads

- Il y a trois modèles de threads
  - 1:1 : un thread d'une application est associé à un kernel thread ordonnançable
    - Modèle simple utilisé par GNU/Linux, *BSD, Windows
  - M:1 : plusieurs threads applicatifs sont associés à un seul *kernel thread*
    - la librairie de thread s'occupe d'ordonnancer les threads applicatifs et de les placer sur un *kernel thread*
    - deux threads applicatifs ne pourront jamais tourner en même temps
    - Exemple : GNU Portable Threads
  - M:N : plusieurs threads applicatifs peuvent être associés à plusieurs kernel threads

API threads : création

- POSIX :

```
int pthread_create(pthread_t* id, const pthread_attr_t* attr,
                   void* (*func)(void*), void* arg);
int pthread_join(pthread_t id, void** ret);
int pthread_detach(pthread_t thread);
```

- Windows :

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD dwCreationFlags, LPDWORD lpThreadId);
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
uintptr_t _beginthread(void(*start_address)(void*), unsigned stack_size,
  void* arglist);
uintptr_t _beginthreadex(void* security, unsigned stack_size,
  unsigned (*start_address)(void*), void* arglist, unsigned initflag,
  unsigned* thrdaddr);
```

API threads : création (2)

- C11 :

```
int thrd_create(thrd_t* id, int (*func)(void*), void* arg);
int thrd_join(thrd_t id, int* res);
int thrd_detach(thrd_t thr);
```

- C++11 :

```
template<class Function, class... Args > std::thread(Function&& f,
  Args&&... args);
```

Méthodes :

```
get_id();
native_handle();
join();
detach();
static hardware_concurrency();
```

- Boost :

```
template <class F,class A1,class A2,...>
boost::thread(F f,A1 a1,A2 a2,...);
```

Méthodes :

```
get_id();
native_handle();
detach();
join();
static hardware_concurrency();
```

API threads : arrêt

- Normalement un thread doit s'arrêter tout seul (retour de fonction ou exit)
- Arrêter un thread depuis un autre est une mauvaise idée !
- Des objets de synchronisation peuvent ne pas être déverrouillés, la mémoire non désallouée, ...
- Certaines API rendent possible d'indiquer un bloc de code où le thread ne pourra pas être annulée (une fois sortie il le sera par contre)
- POSIX :

```
int pthread_cancel(pthread_t thread);
int pthread_setcancelstate(int state, int* oldstate);
void pthread_cleanup_push(void (*routine)(void*), void* arg);
void pthread_cleanup_pop(int execute);
```

- Windows :

```
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Synchronisation

- Vu que l'espace d'adressage est commun, il faut des moyens de synchroniser les threads (attente d'un résultat, accès à une variable, ...)
- Sinon il y a un risque de comportements indéfinis ou résultats erronnés
- Il y a plusieurs possibilités :
  - sémaphore et mutex (MUTual EXclusion)
  - verrou lecture / écriture
  - variable atomique
  - condition
  - barrière

Mutex

- Similaire à une sémaphore avec un compteur de 1
- Protège l'accès à une zone de code
- Un seul thread peut y accéder, les autres sont bloqués
- Il est possible de mettre un *timeout* pour éviter d'être bloqué indéfiniment et faire autre chose
- On peut tenter de verrouiller un mutex : la fonction renvoie *false* immédiatement s'il est déjà pris
- Un même thread peut verrouiller plusieurs fois le même mutex si ce dernier est configuré pour (i.e. mutex récursif)
- Remarque sous Windows : bien que l'on puisse utiliser CreateMutex et OpenMutex avec des threads, on leur préfèrera l'utilisation de CRITICAL_SECTION car beaucoup moins couteuse (CreateMutex va créer un objet noyau) !

API Mutex

- POSIX :

```
int pthread_mutex_init(pthread_mutex_t* mtx, const pthread_mutexattr_t* attr);
int pthread_mutex_destroy(pthread_mutex_t* mtx);
int pthread_mutex_lock(pthread_mutex_t* mtx);
int pthread_mutex_trylock(pthread_mutex_t* mtx);
int pthread_mutex_unlock(pthread_mutex_t* mtx);
```

- Windows :

```
void InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
void LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

API Mutex (2)

C11 :

```
int mtx_init(mtx_t* mtx, int type);
void mtx_destroy(mtx_t* mtx);
int mtx_lock(mtx_t* mtx);
int mtx_unlock(mtx_t* mtx);
int mtx_trylock(mtx_t* mtx);
int mtx_timedlock(mtx_t* mtx, const struct timespec* ts);
```

C++11 :

```
class std::mutex
```

Méthodes :

```
void lock();
void unlock();
bool try_lock();
```

```
class std::timed_mutex
```

Méthodes :

```
bool try_lock_for(const std::chrono::duration<Rep,Period>& timeout);
bool try_lock_until(const std::chrono::time_point<Clock,Duration>& time);
```

Boost :

```
class boost::mutex
```

Méthodes :

```
void lock();
void unlock();
bool try_lock();
```

Mutex : *deadlock*

- Problème possible avec les mutex
- Exemple :
  - Un thread T1 enchaine le verrouillage des mutex M1 puis M2
  - Un thread T2 enchaine le verrouillage des mutex M2 puis M1
  - T1 se fait préempter avant l'acquisition de M2
  - T2 acquiert M2 mais ne peut pas acquérir M1 (pris par T1)
  - T1 est actif mais ne peut pas acquérir M2 (pris par T2)
  - Le système est bloqué (0% de CPU cependant)
- Solution : s'il y a besoin de plusieurs mutex, il faut les verrouiller **dans l'ordre** par **tous** les threads.
- Solution C++**17**, utiliser les *std::scoped_lock*

Mutex : *livelock*

- Analogie de deux personnes face à face dans un corridor :
  - P1 bouge sur sa droite
  - P2 bouge sur sa gauche
  - et ainsi de suite...
- Exemple :
  - T1 acquiert M1
  - T2 acquiert M2
  - Au même moment :
    - T1 veut acquérir M2 mais échoue alors il relâche M1
    - T2 veut acquérir M1 mais échoue alors il relâche M2
  - T1 acquiert M2
  - T2 acquiert M1
  - et ainsi de suite...
- Le système est bloqué mais chacun des threads est actif
- Solution : s'il y a besoin de plusieurs mutex, il faut les verrouiller dans l'ordre par tous les threads

Mutex : inversion de priorité

- Analogie d'une Porsche 911 derrière une Renault 5 sur une route étroite
  - la 911 ne peut dépasser la R5, et la R5 ne peut pas la laisser passer
  - la 911 doit attendre derrière la R5 même si elle plus puissante !
- En informatique :
  - Priorité : T3 > T2 > T1
  - T1 acquiert un mutex
  - T3 se réveille et préempte T1
  - T3 tente d'acquérir le mutex, il ne peut pas et se rendort
  - T1 reprend son exécution (toujours avec le mutex verrouillé)
  - T2 préempte T1
  - T2 fini son travail
  - T1 fini son travail et relâche le mutex
  - T3 s'exécute

Mutex : inversion de priorité (2)

- Problème : la tâche de haute priorité est retardé et cela peut avoir des effets dramatiques !
- Exemple : arrêt d'urgence centrale nucléaire, la mission *Pathfinder* (https://degeeter.pagesperso-orange.fr/inv_fr.htm)
- Solutions :
  - Architecturer l'application en interdisant la prise de mutex par des threads de priorité différente
  - Dans le cas où plus de deux threads, utiliser des mutex à héritage de priorité

Verrou lecture / écriture

- On va différencier le verrouillage pour la lecture et l'écriture
- On pourra avoir plusieurs lecteurs en parallèle ou un seul écrivain (et pas de lecteurs)
- Cas d'utilisation : accès en lecture important comparé aux accès en écriture !
- Ces verrous ont un coût d'exécution plus important que les mutex
  - Il faut que le code soit plus conséquent que l'incrémentation d'une variable !

API verrou lecture / écriture

- POSIX :

```
int pthread_rwlock_init(pthread_rwlock_t* mtx,
  const pthread_rwlockattr_t* attr);
int pthread_rwlock_destroy(pthread_rwlock_t*  mtx);
int  pthread_rwlock_rdlock(pthread_rwlock_t*  mtx);
int pthread_rwlock_tryrdlock(pthread_rwlock_t* mte);
int  pthread_rwlock_wrlock(pthread_rwlock_t*  mtx);
int pthread_rwlock_trywrlock(pthread_rwlock_t* mtx);
int pthread_rwlock_unlock(pthread_rwlock_t* mtx);
```

- Windows :

```
void  InitializeSRWLock(PSRWLOCK  SRWLock);
void  AcquireSRWLockShared(PSRWLOCK SRWLock);
BOOLEAN  TryAcquireSRWLockShared(PSRWLOCK  SRWLock);
void  AcquireSRWLockExclusive(PSRWLOCK  SRWLock);
BOOLEAN TryAcquireSRWLockExclusive(PSRWLOCK  SRWLock);
```

API verrou lecture / écriture (2)

- C++17 :

```
class std::shared_mutex
```

Méthodes :

```
void lock();
void unlock();
bool try_lock();
void lock_shared();
void unlock_shared();
bool try_lock_shared();
```

- Boost :

```
class boost::shared_mutex
```

Méthodes :

```
void lock();
void unlock();
bool try_lock();
void lock_shared();
void unlock_shared();
bool try_lock_shared();
```

Spinlock

- Un thread va s'endormir si un mutex est déjà pris
- Il va laisser son quantum de temps à quelqu'un d'autre avant d'être à nouveau ordonnancer
- Un *spinlock* va tenter d'acquérir le verrou pendant tout le quantum de temps du thread !
- Cela peut être intéressant si le temps de verrouillage d'une ressource est très faible

API *spinlock*

- POSIX :

```
int  pthread_spin_lock(pthread_spinlock_t*  lock);
int pthread_spin_trylock(pthread_spinlock_t* lock);
int  pthread_spin_unlock(pthread_spinlock_t* lock);
```

- Boost :

```
class boost::spinlock
```

Méthodes :

```
void lock();
void unlock();
```

Variable atomique

- Mythes sur le C et C++
  - Incrémenter un entier est atomique : FAUX
  - Assigner une valeur à un entier est atomique : FAUX
  - Une variable *volatile* rend les opérations atomiques : FAUX
  - Une variable de type sig_atomic_t est atomique : uniquement pour la gestion des signaux !
- A retenir : le standard C et C++ ne garantissent nullement l'atomicité des variables *classiques* !
- Seules les variables dont le type est atomic_X ou std::atomic<X> ont, d'après le standard, des opérations atomiques !

Variable atomique (2)

- L'ordonnanceur peut préempter un thread et donc le mettre en pause après une instruction assembleur
- Une opération *simple* peut prendre plus d'une instruction assembleur : incrémenter un entier sur une architecture x86 prend **trois instructions**
- Cela peut engendrer des effets de bords gênants !
- L'utilisation des variables atomiques permet d'être sûr d'éviter ça
- Les variables atomiques sont moins couteux qu'un mutex !

Variable atomique (3)

```
$ cat increment.c
int myvar = 0;
void increment(void)
{
  myvar++;
}

$ gcc -c test.c
$ objdump -d increment.o
increment.o:     format de fichier elf64-x86-64   Déassemblage de la section .text :  0000000000000000 <increm
   0:        55                              push   %rbp
   1:        48 89 e5                        mov    %rsp,%rbp
   4:        8b 05 00 00 00 00               mov    0x0(%rip),%eax         # a <increment+0xa>
   a:        83 c0 01                        add    $0x1,%eax
   d:        89 05 00 00 00 00               mov    %eax,0x0(%rip)         # 13 <increment+0x13>
  13:        90                              nop
  14:        5d                              pop    %rbp
  15:        c3                              retq
```

API atomique

► C11 :

```
atomic_int myvar;
atomic_long  mylong;
...
void atomic_init(volatile A* obj, A arg);
void  atomic_store(volatile  A* obj,  A arg);
A atomic_load(const volatile  A* obj);
A atomic_fetch_add(volatile  A* obj,  A arg);
A atomic_fetch_sub(volatile  A* obj, A arg);
A atomic_fetch_or(volatile  A* obj,  A arg);
A atomic_fetch_xor(volatile  A* obj,  A arg);
Aatomic_fetch_and(volatile A* obj, Aarg);
_Bool  atomic_compare_exchange_weak(volatile  A* obj,  A* expected,  A desired);
_Bool  atomic_compare_exchange_strong(volatile  A* obj,  A* expected,  A desired);
```

► C++11

```
template<class T> struct atomic;
template<> struct atomic<integral>;
template<class T> struct  atomic<T*>;
```

Méthodes :

```
operator=(T desired)
store(T  desired);
T load()  const;
T  exchange(T  desired);
T  fetch_add(T_arg);
T fetch_sub(T_arg);
T   fetch_or(T_arg);
T fetch_xor(T_arg);
T fetch_and(T_arg);
```

Algorithme *lock-free*

- Il est possible avec les variables atomiques de réaliser des algorithmes sans besoin de verrouillage (mutex)
- Les algorithmes *lock-free* sont assez complexes et ne devraient être utilisés qu'en dernier recours pour gagner les quelques nanosecondes qui manquent !
- Lire :
  - https://preshing.com/20120612/an-introduction-to-lock-free-programming/
  - https://nullprogram.com/blog/2014/09/02/

Programmer ses applications en Multicore

- TP 1

Condition

- Un thread T1 va attendre sur un objet condition
- Un thread T2 va notifier cette condition ce qui va réveiller T1
- L'attente est non-active (le thread est endormi) et ne consomme pas de CPU
- L'usage nécessite un mutex pour garantir l'ordre à la fois pour l'attente et pour la notification
- *Spurious wakeup*
  - une attente peut se débloquer sans que la condition ne soit signalée
  - Toujours tester un prédicat (via la fonction qui va bien ou en utilisant une boucle)

Condition (2)

- Pseudo-code d'un thread attendant une notification :

```
lock(mtx)
while(trigger != true)
{
  // implicitely unlock mtx in wait_condition
  wait_condition(cnd, mtx)
  // implicitely lock mtx after function call
}
trigger = false
unlock(mtx)
```

- Pseudo-code d'un thread notifiant une condition :

```
lock(mtx)
trigger = true
signal_condition(cnd)
unlock(mtx)
```

API condition

- POSIX :

```
int pthread_cond_init(pthread_cond_t* cnd, const pthread_condattr_t* attr);
int pthread_cond_destroy(pthread_cond_t* cnd);
int pthread_cond_signal(pthread_cond_t* cnd);
int pthread_cond_broadcast(pthread_cond_t* cnd);
int pthread_cond_wait(pthread_cond_t* cnd, pthread_mutex_t* mtx);
int pthread_cond_timedwait(pthread_cond_t* cnd, pthread_mutex_t* mtx,
  const struct timespec* abstime);
```

- Windows :

```
void InitializeConditionVariable(PCONDITION_VARIABLE ConditionVariable);
BOOL SleepConditionVariableCS(PCONDITION_VARIABLE ConditionVariable,
  PCRITICAL_SECTION CriticalSection, DWORD dwMilliseconds);
void WakeConditionVariable(PCONDITION_VARIABLE ConditionVariable);
void WakeAllConditionVariable(PCONDITION_VARIABLE ConditionVariable);
```

API condition (2)

► C11 :

```
int  cnd_init(cnd_t*  cnd);
void cnd_destroy(cnd_t* cnd);
int cnd_signal(cnd_t* cnd);
int cnd_broadcast(cnd_t*  cnd);
int cnd_wait(cnd_t*  cnd, mtx_t*  mtx);
int cnd_timedwait(cnd_t*  cnd, mtx_t*  mtx, const struct timespec*  ts);
```

► C++11 :

```
class std::condition_variable;
```

Méthodes :

```
void notify_all();
void notify_one();
void  wait(std::unique_lock<std::mutex>&  lock);
void wait(std::unique_lock<std::mutex>& lock, Predicate pred);
void  wait_for(std::unique_lock<std::mutex>& lock,
  const std::chrono::duration<Rep,  Period>&  rel_time);
void  wait_for(std::unique_lock<std::mutex>&  lock,
  const std::chrono::duration<Rep, Period>& rel_time, Predicate pred);
void  wait_until(std::unique_lock<std::mutex>& lock,
  const  std::chrono::time_point<Clock,  Duration>&  abs_time);
void  wait_until(std::unique_lock<std::mutex>&  lock,
  const  std::chrono::time_point<Clock,  Duration>&  abs_time,  Predicate  pred);
```

API condition (3)

- Boost :

```
class boost::condition_variable;

Méthodes :
void notify_all();
void notify_one();
void wait(boost::unique_lock<boost::mutex>& lock);
void wait(boost::unique_lock<boost::mutex>& lock, predicate_type pred);
void timed_wait(boost::unique_lock<boost::mutex>& lock,
  boost::system_time const& abs_time);
void timed_wait(boost::unique_lock<boost::mutex>& lock, =
  boost::system_time const& abs_time, predicate_type pred);
```

Condition : *thundering herd*

- Problème possible si beaucoup de threads attendent une même condition pour accéder à une ressource (socket, fichier, ...)
- Dès que cette condition est notifiée, il y a reprise de l'activité de tous les threads et ils vont entrer en compétition pour acquérir le mutex
- Ceci peut consommer beaucoup de ressources CPU !

Barrière

- Les barrières sont utilisées pour garantir que les threads sont bien arrivés à un certains point !
- Les threads vont être bloqués à la barrière tant que le nombre exact n'est pas atteint
- Dès que le nombre exacte est atteint, tous les threads vont reprendre l'exécution
- Cas d'utilisation : initialisation asynchrone de différentes ressources avant de continuer
- Analogie : point de rendez-vous

API barrière

- POSIX :

```
int pthread_barrier_init(pthread_barrier_t* barrier,
  const pthread_barrierattr_t* attr, unsigned count);
int pthread_barrier_destroy(pthread_barrier_t* barrier);
int pthread_barrier_wait(pthread_barrier_t* barrier);
```

- Windows :

```
BOOL InitializeSynchronizationBarrier(LPSYNCHRONIZATION_BARRIER lpBarrier,
    LONG lTotalThreads, LONG lSpinCount);
BOOL DeleteSynchronizationBarrier(LPSYNCHRONIZATION_BARRIER lpBarrier);
BOOL EnterSynchronizationBarrier(LPSYNCHRONIZATION_BARRIER lpBarrier,
  DWORD dwFlags);
```

API barrière (2)

► C++20 :

```
class std::barrier;
```

Méthodes :

```
void arrive_and_wait() const;
arrival_token arrive()  const;
void  wait(arrival_token&&) const;
```

```
class std::latch;
```

Méthodes :

```
void  count_down_and_wait()  const;
void  count_down() const;
void  wait()  const;
```

► Boost :

```
class boost::barrier;
```

Méthodes :

```
bool  wait()  const;
void  count_down_and_wait();
```

```
class boost::latch;
```

Méthodes :

```
void  wait()  const;
bool try_wait()  const;
void  count_down();
void  count_down_and_wait();
void  reset(size_t);
```

Thread Local Storage

- Les threads peuvent avoir besoin de variables globales mais dont les valeurs sont **différentes** d'un thread à l'autre
- Cas d'utilisation : simplicité d'utilisation de variables globales sans verrou

Thread Local Storage (2)

- POSIX :

```
int pthread_key_create(pthread_key_t* key, void (*dstr_function) (void*));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void* ptr);
void* pthread_getspecific(pthread_key_t key);
```

- Windows :

```
DWORD TlsAlloc();
BOOL TlsFree(DWORD dwTlsIndex);
BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);
LPVOID TlsGetValue(DWORD dwTlsIndex);
```

Thread Local Storage (3)

- C11 :

```
attribut thread_local (exemple : thread_local int  mytls = 0;)
int tss_create(tss_t* tss_key, tss_dtor_t destructor);
void tss_delete(tss_t tss_id);
void* tss_get(tss_t tss_key);
int tss_set(tss_t tss_id, void* val);
```

- C++11 :

```
attribut thread_local (exemple : thread_local int mytls = 0;)
```

- Boost :

```
class boost::thread_specific_ptr<T>;
```

Once

- Once permet de garantir qu'une séquence d'initialisation ne sera effectuée qu'une seule fois (même si appelée par plusieurs threads)
- POSIX :

```
int pthread_once(pthread_once_t* ctrl, void (*init)(void));
```

- Windows :

```
BOOL InitOnceExecuteOnce(PINIT_ONCE InitOnce, PINIT_ONCE_FN InitFn,
  PVOID Context, LPVOID* Parameter);
```

- C11 :

```
void call_once(once_flag* flag, void (*init)(void));
```

- C++11 :

```
template<class Callable, class... Args>
void call_once(std::once_flag& flag, Callable&& f, Args &&... args);
```

- Boost :

```
template<typename Function>
inline void boost::call_once(Function func, boost::once_flag& flag);
```

Programmer ses applications en Multicore

- TP 2

Affinité du processeur

- Fixer un thread spécifique sur un ou plusieurs CPU
- *CPU pinning* peut réduire les problèmes de rechargement de cache et donc améliorer les performances
- Cela peut compliquer le travail de l'ordonnanceur
- En principe il faut connaître la topologie des CPU
  - attention, la topologie varie d'un CPU à un autre, d'une carte mère à une autre, ...
- Usages : calcul intensif (matrice, ...), dédié un coeur pour des interruptions matériels (embarqué, temps-réel, ...)
- Il faut faire des tests et *benchmarks* !

Affinité du processeur (2)

- GNU/Linux

```
int sched_getcpu();
int sched_setaffinity(pid_t pid, size_t cpusetsize, const cpu_set_t* mask);
int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t* mask);
int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize,
  const cpu_set_t* cpuset);
int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize,
  cpu_set_t* cpuset);
```

- Windows :

```
DWORD GetCurrentProcessorNumber();
BOOL SetProcessAffinityMask(HANDLE hProcess, DWORD_PTR dwProcessAffinityMask);
BOOL GetProcessAffinityMask(HANDLE hProcess, PDWORD_PTR lpProcessAffinityMask,
  PDWORD_PTR lpSystemAffinityMask);
DWORD_PTR SetThreadAffinityMask(HANDLE hThrd, DWORD_PTR dwThreadAffinityMask);
```

- Outils sous GNU/Linux : taskset, hwloc-bind, hwloc-ls
- Outil sous Windows : taskmgr.exe, hwloc-bind, hwloc-ls

Hwloc

- Hwloc est une librairie portable pour déterminer la topologie matériel d'une machine
- Il permet aussi de placer un thread sur un CPU en particulier de manière portable
- Il est intéressant de placer des tâches sur des CPU "proches" pour minimiser les temps de transfert / migration
- Dans les architecture multi-processeurs, on peut également déterminer sur quel processeur physique est attaché un périphérique précis (GPU, carte réseau haut débit)
- Les tâches qui envoient des données sur ces périphériques auront de meilleures performances si elles sont placés sur le CPU attaché au périphérique

Priorité des processus / threads

- On peut spécifier la priorité d'un processus (héritée par tous ses threads) ainsi qu'un thread en particulier
- POSIX :

```
int nice(int prio);
int setpriority(int which, id_t who, int prio);
int getpriority(int which, id_t who);
int pthread_setschedparam(pthread_t thread, int policy,
  const struct sched_param* param);
int pthread_getschedparam(pthread_t thread, int* policy,
  struct sched_param* param);
```

- Windows :

```
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
DWORDGetPriorityClass(HANDLE hProcess);
BOOL SetThreadPriority(HANDLE hThread, int nPriority);
int GetThreadPriority(HANDLE hThread);
```

Priorité des processus / threads (2)

- Choix épineux pour augmenter ou diminuer la priorité
- Cela ne doit pas être choisi au hasard
- Eviter d'avoir des threads hautement prioritaires qui ne rendent pas la main
- Les histoires de priorités interviennent souvent dans les applications orientées systèmes et temps-réel... rarement dans une application métier

Passer la main à un autre processus

- Dans certains cas, un thread / processus peut laisser la main à un autre
- Attention si le thread laissant la main est toujours celui qui a la plus haute priorité, il va être sélectionné !
- En général ce genre de chose peut arriver avec les applications temps-réel... rarement dans les applications métiers

Passer la main à un autre processus (2)

- POSIX :

```
int pthread_yield(void);
int sched_yield(void);
```

- Windows :

```
BOOLSwitchToThread();
Sleep(0);
```

- C11 :

```
void thrd_yield(void);
```

- C++11 :

```
std::this_thread::yield();
```

- Boost :

```
boost::this_thread::yield();
```

Programmer ses applications en Multicore

- TP 3
- TP 4

Formation Programmer ses applications en Multicore

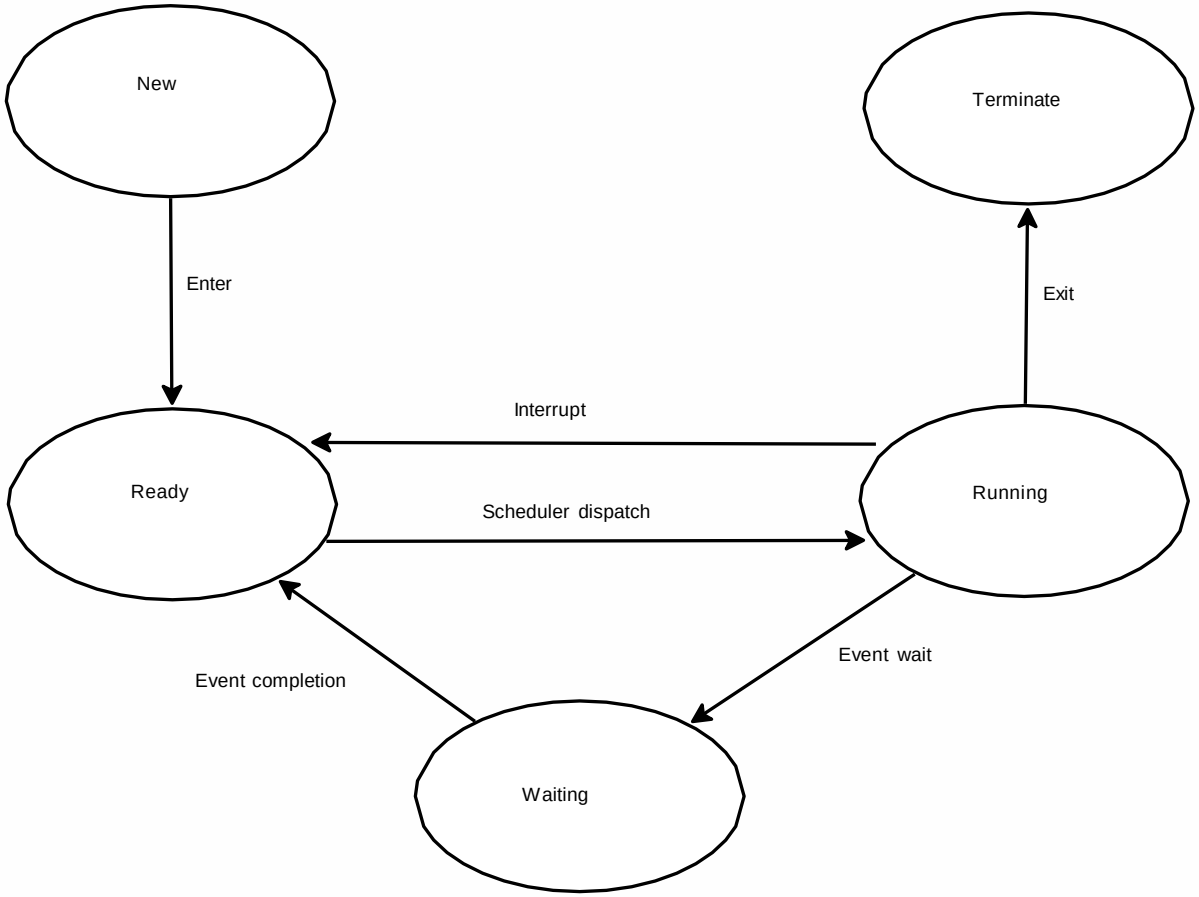
Processus

- Un programme lancé est un processus
- Un processus dispose d'un espace d'adressage en mémoire et d'une pile d'appel (*stack*)
- L'espace d'adressage du processus est isolé des autres sauf cas particuliers (IPC / mémoire partagée)
- Il dispose également d'une priorité d'exécution
- Dans l'embarqué on trouve fréquemment des processus avec des priorités temps-réel
  - Les processus avec des priorités temps-réel s'exécutent avant les autres

Processus (2)

- Un processus a différents états :
  - Nouveau
  - Prêt
  - En attente
  - En cours d'exécution
  - Terminée

Processus (3)


New
Terminate
Enter
Exit
Interrupt
Ready
Running
Scheduler dispatch
Event completion
Event wait
Waiting

API processus Unix

- Appel système : pid_t fork()
- Duplication d'un processus à l'identique
- On peut exécuter une partie de code spécifique ou exécuter un autre programme (appel système execve())
- Le code retour peut valoir :
  - -1 : erreur, impossible de dupliquer le processus, vérifier la valeur d'*errno* pour connaître la raison
  - 0 : on est le processus fils
  - > 0 : on est le processus père, la valeur est l'identifiant du processus (PID)

API processus Unix (2)

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

int main(int argc, char** argv)
{
  pid_t pid = fork();

  if(pid == -1) /* erreur */
  {
    printf("Erreur fork (%d)\n", errno);
  }
  else if(pid == 0) /* fils */
  {
    printf("Je suis le fils !\n");
  }
  else /* père */
  {
    printf("Je suis le père, %u est mon fils\n", pid);
  }

  printf("Processus %uquitte\n", getpid());
  return 0;
}
```

API processus Windows

- Appel système :

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

API processus Windows (2)

```
#include <cstdlib>

#include <iostream>

#define WINDOWS_LEAN_AND_MEAN
#include <Windows.h>

int main(int argc, char** argv)
{
        PROCESS_INFORMATION processInfo;
        STARTUPINFOA si;
        bool ret = false;
        char cmdLine[] = "c:\\windows\\notepad.exe";

        ZeroMemory(&si, sizeof(STARTUPINFO));
        si.cb = sizeof(STARTUPINFO);
        ret = CreateProcessA(cmdLine, NULL, NULL, NULL, false, 0, NULL, NULL, &si,
          &processInfo);

        if (!ret)
        {
                std::cout << "Error CreateProcess" << std::endl;
                return EXIT_FAILURE;
        }

        std::cout << "Launched" << std::endl;
        Sleep(5000);

        WaitForSingleObject(processInfo.hProcess, INFINITE);
        CloseHandle(processInfo.hProcess);
        CloseHandle(processInfo.hThread);

        std::cout << "Program exit" << std::endl;
        return EXIT_SUCCESS;
}
```

Inter-Process Communication

- Mécanismes permettant l'échange de données et la synchronisation entre processus
- Les traitements IPC sont couteux !
- Sous Windows : il faut renseigner le paramètre *lpName* des fonctions pour pouvoir l'utiliser pour synchroniser des processus
- Il est aussi possible d'utiliser ces mécanismes pour synchroniser des **threads**

Inter-Process Communication (2)

- Sémaphore
- File de messages
- Mémoire partagée
- Event Windows
- Socket (Unix, TCP ou UDP)
- Tube nommé ou non
- Pour les sémaphores, files de messages et mémoires partagée, il y a deux normes sous Unix : System V et POSIX
  - macOS ne supporte pas les files de messages POSIX

Sémaphore

- Objet avec un compteur d'unité dont on connait le maximum
- Le compteur peut être initialisé à 0 ou plus
- Un processus peut prendre ou déposer une ou plusieurs unités
- Si la sémaphore ne contient pas assez d'unité le processus est bloqué
- Dès qu'il y a suffisamment d'unité les processus bloqués se débloquent de manière atomique

Sémaphore (2)

- Une sémaphore avec une unité max de 1 unité est équivalent à un mutex
  - Sous Windows, il existe CreateMutex / OpenMutex pour ce cas !
- Cas d'usages : sections critiques, limiter le nombre de processus à exécuter un code
- Problèmes résolus avec les sémaphores : dîner des philosophes, lecteurs / écrivains, producteurs / consommateurs

API Sémaphore

POSIX

```
int sem_init(sem_t* sem, int pshared, unsigned int value);
int sem_destroy(sem_t* sem);
int sem_post(sem_t* sem);
int sem_wait(sem_t* sem);
int sem_trywait(sem_t* sem);
int sem_timedwait(sem_t* sem, const struct timespec* abs);
```

System V

```
int semget(key_t key, int nb, int flag)
int semctl(int semid, int semnum, int cmd, ...);
int semop(int semid, struct sembuf* ops, size_t nb_ops);
```

Windows

```
HANDLE CreateSemaphoreA(LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount, LONG lMaximumCount, LPCSTR lpName);
BOOL CloseHandle(HANDLE hObject);
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG lReleaseCount,
  LPLONG lpPreviousCount);
DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);
```

API Sémaphore (2)

C++20

```
template<std::ptrdiff_t LeastMaxValue> class counting_semaphore;
template<std::ptrdiff_t  LeastMaxValue> class  binary_semaphore
```

Méthodes :

```
void acquire();
void release(std::ptrdiff_t update = 1);
bool try_acquire();
template<class Rep, class Period> bool try_acquire_for(
  const std::chrono::duration<Rep, Period>& rel_time);
template<class Clock, class Duration> bool try_acquire_until
  const std::chrono::time_point<Clock, Duration>& abs_time);
```

Boost

```
class boost::interprocess::interprocess_semaphore(int initial)
```

Méthodes :

```
void post();
void wait()
bool try_wait();
bool timed_wait(const boost::posix_time::ptime& tm);
```

File de messages

- Rien à voir avec MPI
- Un processus peut envoyer un message (structure) via une file
- Un processus peut recevoir le(s) message(s) contenu(s) dans une file
- Attention à la limite de taille de messages sur certaines implémentations

API file de messages

POSIX :

```
mqd_t mq_open(const char* name, int flag);
int mq_close(mqd_t fd);
int mq_unlink(const char* name);
ssize_t mq_receive(mqd_t fd, char* msg, size_t msg_len, unsigned int* prio);
int mq_send(mqd_t fd, const char* msg, size_t msg_len, unsigned int prio);
```

System V :

```
int msgget(key_t key, int flag)
int msgctl(int msqid, int cmd, struct msqid_ds* buf);
int msgsnd(int msqid, const void* msg, size_t msg_len, int flag);
ssize_t msgrcv(int msqid, void* msg, size_t msg_len, long type, int flag);
```

Windows :

```
HANDLE CreateMailslotA(LPCSTR lpName, DWORD nMaxMessageSize,
  DWORD lReadTimeout, LPSECURITY_ATTRIBUTES lpSecurityAttributes);
BOOL GetMailslotInfo(HANDLE hMailslot, LPDWORD lpMaxMessageSize,
  LPDWORD lpNextSize, LPDWORD lpMessageCount, LPDWORD lpReadTimeout);
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);
BOOL ReadFile(HANDLE hFile,LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
```

Mémoire partagée

- Rend disponible une zone mémoire (structure, ...) à d'autres processus
- Utiliser les sémaphores pour *protéger* les accès concurrents à la zone !
- Une fois la zone mémoire ouverte, il faut la rendre disponible dans le processus avec un appel système correspondant de l'API (mmap(), shmat(), ...)

API mémoire partagée

POSIX

```
int shm_open(const char* name, int flag, mode_t mode);
int shm_unlink(const char* name);
void* mmap(void* addr, size_t length, int prot, int flags, int fd,
  off_t offset);
int munmap(void* addr, size_t length);
```

System V

```
int shmget(key_t key, int nb, int flag)
int shmctl(int shmid, int cmd, struct shmid_ds* buf);
void* shmat(int shmid, const void* addr, int flag);
void* shmdt(const void* addr);
```

Windows

```
HANDLE CreateFileMapping(HANDLE hFile,
   LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect,
   DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCSTR lpName);
  HANDLE OpenFileMappingA(DWORD dwDesiredAccess, BOOL bInheritHandle,
   LPCSTR lpName);
LPVOID MapViewOfFile(HANDLE hFileMappingObject, DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, SIZE_T dwNumberOfBytesToMap);
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);
```

API mémoire partagée (2)

- Boost

```
class boost::interprocess::shared_memory_object();
class boost::interprocess::shared_memory_object(open_only_t, const char* name,
mode_t mode);  class template<typename MemoryMappable>
boost::interprocessmapped_region(const MemoryMappable&, mode_t, offset_t = 0,
  std::size_t = 0, const void* = 0, map_options_t = default_map_options);
class  boost::interprocess::mapped_region()
```

Méthodes :

```
void* get_address();
size_t get_size();
```

Event Windows

- Similaire au condition des threads
- Un processus P1 attend sur un évènement
- Un autre processus P2 (ou thread) active l'évènement
- P1 est notifié et se réveille

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpEventAttributes,  BOOL bManualReset,
   BOOL bInitialState,  LPCTSTR lpName);
BOOL SetEvent(HANDLE  hEvent);
BOOL  ResetEvent(HANDLE  hEvent);
DWORD  WaitForSingleObject(HANDLE  hHandle,  DWORD  dwMilliseconds);
```

Autres IPC

- Signaux
  - Unix : signal(), sigaction(), kill()
- Socket local (AF_UNIX) ou socket réseau (AF_INET sur localhost)
- Tubes
  - Unix : pipe(), mkfifo()
  - Windows : CreateFile (nom de fichier \\.\pipe\myname), SetNamedPipeHandleState

Formation Programmer ses applications en Multicore

OpenMP

- Open Multi-Processing
- Standard (2020 : version 5.0)
- Disponible pour les langages C, C++ et Fortran
- OpenMP est basé sur les threads
- API + instructions préprocesseur (#pragma)
  - Des #defines indique si OpenMP est supporté
  - Si le compilateur supporte OpenMP alors la parallélisation se fait automatiquement

OpenMP (2)

- Le pattern utilisé par OpenMP est *fork and join*
- Création d'un pool de threads, calcul et synchronisation puis destruction des threads
- OpenMP permet aussi de protéger une variable / section de l'accès concurrent
- Possibilité de spécifier des variables partagées et d'autres privées au thread de travail

Exemple

```
#pragma omp parallel for
for(int i = 0; i < 5000; i++)
{
  somme += i;
}
```

Instructions et API

- Instructions préprocesseurs

```
omp parallel for
ompbarrier
omp single
ompcritical
ompatomic
omp flush
ompsection
omp ordered
```

- API

```
int omp_get_num_threads();
int omp_get_thread_num();
int omp_in_parallel()
int omp_get_max_threads();
int omp_get_num_procs();
int  om_get_dynamic();
int omp_get_ntested();
double omp_get_wtime();
double omp_get_wstick();
void omp_set_num_threads(int);
void omp_set_nested(int);
```

Variables et synchronisation

- Une variable peut être partagée ou privée dans un bloc OpenMP
- Une variable privée est "recopiée" automatiquement
- Attention aux accès en écriture des variables partagées !
- De nombreux mécanismes de synchronisation existent
  - #pragma omp critical
  - #pragma omp atomic
  - #pragma omp barrier

Bonnes pratiques

- Rappel : la création de threads est couteuse et la synchronisation également !
  - En cas de parallisation successive, avoir plusieurs *#pragma omp for* au sein d'un bloc *#pragma omp parallel*
- Si le nombre de calcul / données est trop faible, une implémentation séquentielle peut être plus rapide
- Les *benchmarks* sont obligatoires pour déterminer à partir de quelle quantité de données / threads la parallélisation est utile
- Attention à l'ordonnancement sur deux vCPU d'un même coeur (SMT)

Bonnes pratiques (2)

- Algorithme qui ne prend pas en compte le cache CPU
- Un tableau est accédé par ligne
- Pour itérer sur un tableau 2D, utiliser :

```
for(int i = 0 ; i < max ; i++)
{
  for(int j = 0 ; j < max ; j++)
  {
    sum += array[i][j]
  }
}
```

- Plutôt que :

```
for(int j = 0 ; j < max ; j++)
{
  for(int i = 0 ; i < max ; i++)
  {
    sum += array[i][j]
  }
}
```

Bonnes pratiques (3)

- *False sharing*
  - la ligne de cache est modifié par chaque thread
  - chaque thread va devoir la relire en mémoire = perte de temps
- En cas de modification d'un tableau partagé, *padder* chaque entrée avec la valeur du cache

```
int array[nb_threads][cache_line_size];

#pragma ompparallel for shared(arrray, nb_threads)
for(int i =0; i <nb_threads ; i++)
{
  array[i][0] =i;
}
```

Tasks

- Avec OpenMP > 3.0, on peut utiliser des tâches
- Une tâche est liée à un thread en particulier
- Une tâche peut faire autre chose que du calcul (I/O, recherche, ...)

Méthodes *fine grain* et *coarse grain*

- *fine grain* : parallélisation automatique
- *coarse grain* : distribution manuelle de la charge de travail et synchronisation manuelle des threads
- L'approche *coarse grain* peut être envisagée pour du code complexe où l'on recherche la performance au détriment de la simplicité du code

Conclusion OpenMP

- Mise en oeuvre relativement facile avec les #pragma
- Bien tester et penser à la synchronisation
- Faire attention aux problèmes potentiels liés au cache CPU

Programmer ses applications en Multicore

- TP 5

Formation Programmer ses applications en Multicore

Message Passing Interface

- API standardisée en C et Fortran
- Il existe une interface C++ (binding) mais dépréciée en faveur de Boost.MPI
- De nombreuses implémentations propriétaires et open-sources
  - Open MPI
  - MPICH2
  - MS-MPI
  - Intel MPI
  - ...
- Répartir la charge de travail sur des machines distinctes (grille de calcul ou via socket réseau) ou sur des processus (via communication IPC)
- Possible de coupler MPI avec d'autres technologies (threads, OpenMP, OpenCL, ...)

Message Passing Interface (2)

- MPI utilise un compilateur spécifique (*mpicc* sous Unix)
- Il faut utiliser *mpiexec* pour lancer le programme
- Une configuration et des variables d'environnement peuvent être utilisées
- La configuration sert à déterminer le nombre de processus, les IP des machines, ...

Initialisation API

- Initialiser MPI :

```
int MPI_Init(int* argc, char*** argv);
int MPI_Init_thread(int* argc, char*** argv,
          int required, int* provided);
int MPI_Finalize();
```

- Utiliser MPI_Init_thread si le programme est multithreadé
- required peut prendre MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED, MPI_THREAD_SERIALIZED ou MPI_THREAD_MULTIPLE
- Attention certaines implémentations de MPI ne les supportent pas tous

Initialisation API (2)

- Pour avoir des informations sur le contexte MPI :

```
int MPI_Comm_size(
  MPI_Comm communicator,
  int* size);
int MPI_Comm_rank(
  MPI_Comm communicator,
  int* rank);
int MPI_Get_processor_name(
  char* name,
  int* name_length);
```

Types de communicateur MPI

- Les communicateurs MPI regroupent un nombre de noeuds (processus ou machines)
- MPI_COMM_WORLD = tous les noeuds
- On peut découper ce MPI_COMM_WORLD en sous-groupes

```
int MPI_Comm_split(
   MPI_Comm comm,
   int color,
   int key,
   MPI_Comm* newcomm);

int MPI_Comm_free(MPI_Comm* comm);
```

Types de communicateur MPI (2)

- On peut aussi créer des communicateurs MPI

```
int MPI_Comm_dup(
    MPI_Comm comm,
    MPI_Comm* newcomm);
int MPI_Comm_create(
    MPI_Comm comm,
    MPI_Group group,
    MPI_Comm* newcomm);
int MPI_Group_incl(
    MPI_Group group,
    int n,
    const int ranks[],
    MPI_Group* newgroup);
```

Types de variable MPI

- MPI_BYTE
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_LONG_LONG
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_LONG_LONG
- MPI_FLOAT
- MPI_DOUBLE
- MPI_LONG_DOUBLE

API communication point-à-point bloquante

- A l'instar des sockets réseaux, il y a une fonction d'envoi et de réception de données :

```
int MPI_Send(
  void* data,
  int count,
  MPI_Datatype datatype,
  int destination,
  int tag,
  MPI_Comm communicator);
int MPI_Recv(
  void* data,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm communicator,
  MPI_Status* status);
```

API communication point-à-point bloquante (2)

- MPI_Status permet de connaitre certains éléments :
  - rang de l'émetteur
  - tag du message
  - longueur du message

```
int MPI_Get_count(
  MPI_Status* status,
  MPI_Datatype datatype,
  int* count);
```

API communication point-à-point bloquante (3)

- Le récepteur peut connaitre la taille du message qui va être reçu (i.e. avant un MPI_Recv)
- Cas d'usage : taille variable de message et allocation mémoire pour le buffer de réception

```
int MPI_Probe(
  int source,
  int tag,
  MPI_Comm comm,
  MPI_Status* status);
```

API communication point-à-point non-bloquante

- Il est possible d'envoyer / recevoir des données de manière asynchrone tout en faisant des calculs en attendant

```
int MPI_Isend(void* buffer, int count, MPI_Datatype datatype,
  int dest, int tag, MPI_Communicator comm,MPI_Request* request);
int MPI_Irecv(void* buffer, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Communicator comm, MPI_Request* request);

int MPI_Wait(MPI_Request* request, MPI_Status* status);
int MPI_Waitany(int count, MPI_Request array_of_requests[],
  int* index, MPI_Status* status);

int MPI_Test(MPI_Request* request, int* flag, MPI_Status* status);
int MPI_Testany(int count, MPI_Request array_of_requests[], int* index,
int* flag, MPI_Status* status);
```

API de synchronisation

- Il y a des mécanismes de synchronisation avec MPI
- La barrière est utilisé pour attendre que tous les processus soit au même niveau avant de repartir

```
int MPI_Barrier(MPI_Communicator comm);
int MPI_Ibarrier(MPI_Communicator comm, MPI_Request* request);
```

API communication collective

- Il y a des API pour envoyer des données à tous les noeuds en un seul appel
- Méthodes :
  - One-to-many : Broadcast et Scatter
  - Many-to-one : Gather
  - Many-to-many : Allgather, Alltoall

API communication collective : broadcast

- Envoi / réceptionne les mêmes données à tous les noeuds
- L'émetteur est désigné par le paramètre *root*
- Les autres sont récepteurs

```
int MPI_Bcast(
  void* data,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm communicator);
```

API communication collective : scatter

- Découpe les données et en envoi une partie à chacun des noeuds

```
int MPI_Scatter(
  void* send_data,
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  int root,
  MPI_Comm communicator);
```

API communication collective : Gather

- Envoi / réceptionne le résultat du calcul des données
- Le récepteur est désigné par le paramètre *root*

```
int MPI_Gather(
  void* send_data,
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  int root,
  MPI_Comm communicator);
```

API communication collective : Allgather

- Tout le monde envoi ses données et reçoit les données des autres

```
int MPI_Allgather(
  void* send_data,
  int send_count,
  MPI_Datatype send_datatype,
  void* recv_data,
  int recv_count,
  MPI_Datatype recv_datatype,
  MPI_Comm communicator);
```

API communication collective : Alltoall

- Tout le monde envoi ses données à tout le monde

```
int MPI_Alltoall(
  const void* sendbuf,
  int sendcount,
  MPI_Datatype sendtype,
  void* recvbuf,
  int recvcount,
  MPI_Datatype recvtype,
  MPI_Comm comm);
```

API communication collective : reduce

- Pour les opérations de réduction "simples",
- Avec MPI, on a MPI_MAX / MIN / SUM / PROD / LAND / LOR / BAND / BOR / MAXLOC / MINLOC

```
int MPI_Reduce(
  void* send_data,
  void* recv_data,
  int count,
  MPI_Datatype datatype,
  MPI_Op op,
  int root,
  MPI_Comm communicator);
int MPI_Allreduce(
  void* send_data,
  void* recv_data,
  int count,
  MPI_Datatype datatype,
  MPI_Op op,
  MPI_Comm communicator);
```

Programmer ses applications en Multicore

- TP 6

Formation Programmer ses applications en Multicore

Programmer ses applications en Multicore

- TP 7

Conclusion

- On utilise majoritairement les threads pour rentabiliser l'utilisation du ou des processeurs
- Il y a plusieurs API de programmation pour les threads
- Les standards C11, C++11/14/17/20 sont à privilégier s'il y a des besoins de portabilité
- Pour le calcul pure, OpenMP et MPI sont des choix potentiels
- Certains traitements peuvent être également déchargés sur les GPU
- Faire attention à la synchronisation (mutex, sémaphore, ...) et à l'architecture globale de l'application

Derniers mots

- Ces dernières années beaucoup d'efforts ont été fait pour faire entrer les éléments de programmation parallèle dans le standard C++
- Programmer en multithread / multiprocessus permet d'optimiser le rendement des ordinateurs
- La programmation parallèle / concurrente amène des problèmes liés à la synchronisation et l'accès aux données
- Il faut bien penser la conception en amont avant de se lancer dans le code
- Il faut toujours faire des tests et des *benchmarks* pour vérifier que l'on gagne bien en performance !
- Dans certains cas, l'utilisation des GPU est à considérer !

Liens intéressants

- Thread C11 https://fr.cppreference.com/w/c/thread
- Thread C++ https://fr.cppreference.com/w/cpp/thread
- Livre *C++ Concurrency in Action* d'Anthony Williams
- Livre *Développement système sous Linux* de Christophe Blaess
- Livre *Solutions temps réel sous Linux* de Christophe Blaess
- OpenMP https://www.openmp.org/
- Reference OpenMP https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf
- How to get good performance by using OpenMP http://akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf
- Open MPI https://www.open-mpi.org/
- MPI tutorial https://mpitutorial.com/tutorials/
- Managing hardware localities https://www.open-mpi.org/projects/hwloc/tutorials/20150605-PATC-hwloc-tutorial.pdf
- Reference OpenCL 1.2 https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf
- An Introduction to the OpenCL Programming Model https://cims.nyu.edu/~schlacht/OpenCLModel.pdf
- Counting Nanoseconds: Microbenchmarking C++ Code https://www.youtube.com/watch?v=Czr5dBfs72U

Merci pour votre attention.

Questions ?