# Calculateurs à mémoire partagée

Xavier JUVIGNY

ONERA

October 15, 2021

# Plan du cours

# Processus

## Définition

- Créée par l'OS ou un exécutable
- Contient beaucoup d'informations propres :
  - id processus, user id, group id
  - environnement, répertoire de travail
  - instruction du programme, registre, tas, pile, fichiers
  - bibliothèques partagées, signaux action, etc.
- Prends beaucoup de ressources à sa création

# Thread

## Définition

- Exécutable léger créer par un processus;
- Contient peu d'information propre :
  - tas, registres, politique d'exécution
  - variables propres au code exécuté par le thread;
- Possède son propre flot d'instructions;
- Partage ressources processus père avec d'autres threads
- Meurt avec son processus parent;

## Propriétés mémoire partagée

- Changement ressource du père par thread vu par tous ( fermeture fichier,...);
- Deux threads ont deux pointeurs égaux $\Rightarrow$ la même donnée;
- Conflits mémoires possible lors d'accès en écriture par deux threads sur une variable partagée;

# Comparaison performance création processus vs threads

| **Plateforme** | fork() | | | std :: thread | | |
|---|---|---|---|---|---|---|
| | real | user | sys | real | user | sys |
| Intel 2.6Ghz Xeon E5-2670 ( 16 cœurs/nœuds ) | 8.1 | 0.1 | 2.9 | 0.9 | 0.2 | 0.3 |
| Intel 2.8GHz Xeon 5660 ( 12 cœurs/nœuds ) | 4.4 | 0.4 | 4.3 | 0.7 | 0.2 | 0.5 |
| AMD 2.4GHz Opteron ( 8 cœurs/nœuds ) | 17.6 | 2.2 | 15.7 | 1.4 | 0.3 | 1.3 |
| IBM 4.0 GHz Power 6 ( 8 cpus/nœuds ) | 9.5 | 0.6 | 8.8 | 1.6 | 0.1 | 0.4 |

Table: Comparaison création de 50 000 processus ( fork() ) contre création de 50 000 threads ( std::thread )

# Création et terminaison de threads

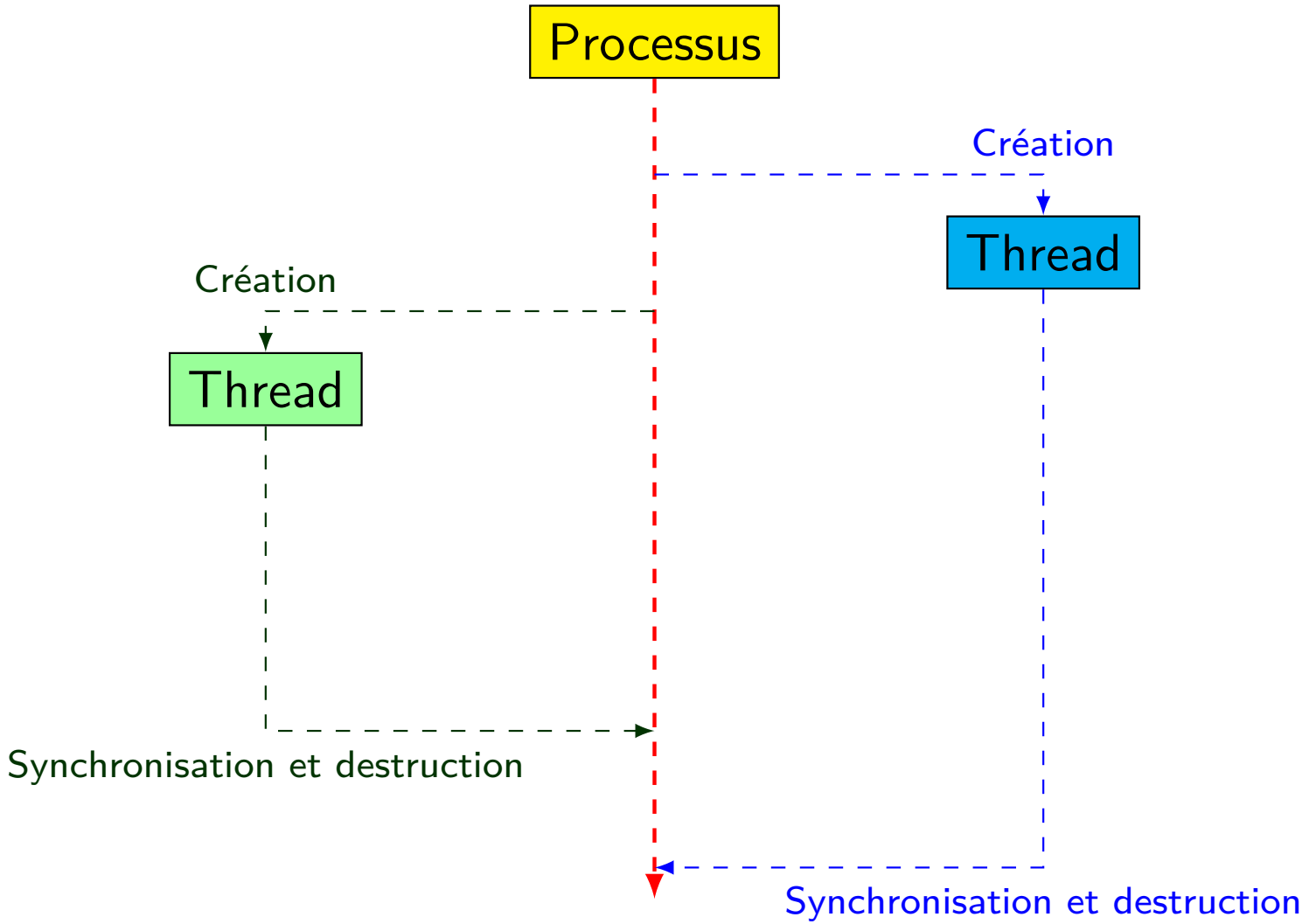


Figure: Exemple de création, synchronisation et terminaison de thread

# Ajustement du nombre de threads et Hyperthreading

## Définition de l'hyperthreading

- Deux fois plus de décodeur d'instruction que d'unités de calcul;
- Deux threads sur une unité de calcul peuvent se partager les ressources de cette unité ( par exemple calcul sur les entiers et calcul sur les réels ).

## Performance de l'hyperthreading

- Au mieux, en prenant deux fois plus de threads que d'unité de calcul, une accélération de 30% par rapport à un calcul fait avec un nombre de thread égal au nombre d'unité de calcul.

Pour connaître le nombre d'unité de calcul, attention aux informations données par vos OS. En général, le nombre de CPUs affichés est en fait le nombre de décodeur d'instructions. Pour connaître le nombre d'unité de calcul, chercher le nombre de cœurs disponibles.

# Exemple de sortie de lscpu

```
 Architecture: x86_64
Mode(s) opératoire(s) des processeurs :32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) par cœur : 2
Cœur(s) par socket : 2
Socket(s): 1
Nœud(s) NUMA : 1
Identifiant constructeur :GenuineIntel
Famille de processeur :6
Modèle : 42
Model name: Intel(R) Core(TM) i7-2620M CPU  2.70GHz
Révision : 7
...
```

# Concurrence d'accès aux données

## Définition

On dit que des threads sont en concurrence d'accès aux données (Datarace condition) quand :

- Un thread lit une variable partagée "pendant" qu'un deuxième thread la modifie;
- Les deux threads tentent de modifier "simultanément" la même variable.

## Phénémologie

- Valeurs "aléatoires", voire incohérentes, de certaines variables partagées;
- Plantage aléatoire du code sur des accès mémoires;

# Section séquentielle

## Cas de sections séquentielles

- On garde une "zone" du programme afin que les threads ne puissent passer que un par un ou qu'un seul thread y passe;
- Cette zone peut être complexe, comme modifier les valeurs d'un tableau, modifier un dictionnaire, une liste, une pile, etc.
- Généralement, mécanisme relativement lourd et lent.
- Exemples : mutex en pthreads, critical en OpenMP, etc.

# Opération atomique

## Cas des opérations atomiques

- La section séquentielle se réduit à une seule opération simple sur un type de base ( entier, réels ,etc. )
- Mécanisme léger et non pénalisant;
- Exemples : atomic sur pthread et OpenMP.

# Affinité

## Définition

Le système d'exploitation attache un thread à une unité de calcul spécifique : on parle d'affinité

## Intérêt

- Par défaut, un thread s'exécute sur n'importe quel unité de calcul, et peut migrer durant son exécution;
- Lorsqu'il migre, on pert les données en mémoire cache;
- Sur architecture NUMA, on peut même changer de socket !
- On peut contrôler l'affinité avec l'utilitaire `taskset` (Linux) sinon API différent selon bibliothèque et OS utilisé !

# Application "Memory bound"

## Définition

- Quand la vitesse d'exécution de l'application limitée par la vitesse d'accès aux données

## Explication

- La mémoire vive plus lente que le traitement des données par un CPU;
- Mémoire cache et mémoire interlacée permet d'accélérer l'accès aux données;
- Algorithmes pas toujours possible d'optimiser en mémoire cache ou avec mémoire interlacée;
- Plus le nombre d'unités de calcul grand, plus difficile de ne pas être memory bound.

# Application "cpu bound"

## Définition

- Quand la vitesse d'exécution de l'application est limitée à la vitesse de traitement du CPU

## Explication

- Programme bien optimisé pour l'architecture mémoire du calculteur;
- Toujours chercher à être le plus "cpu bound".
- Pas toujours possible !

# Quand peut-on être cpu bound ?

## Avec la mémoire cache

- Quand le nombre de données accédées $<$ nombre d'opérations effectuées;
- Pas de saut mémoire non réguliers pour la preamption de cache;
- Avoir un algorithme local en temps et espace : accèder dans le délai le plus bref aux mêmes données le plus possible.

## Avec la mémoire interlacée

- Quand on accède à des données successives sur des bancs mémoires différents;
- Ou accès à très peu de variables différentes pour stocker sur des registres.

# Exemple suite itérative

```
double u = 56547;
for ( unsigned int iter = 0; iter < 1023; ++iter ) {
    u = (u%2 == 0 ? u/2 : (3*u+1)/2);
}
```

Mémoire cache

cpu bound

Mémoire interlacée

cpu bound

# Exemple opération vectorielle

```
unsigned long N = 1000000UL;
std::vector<double> u(N), v(N), w(N);
for ( int i = 0; i < N; ++i ) {
    u[i] = std::max(v[i],w[i]);
}
```

Mémoire cache

memory bound

Mémoire interlacée

cpu bound

# Opération matricielle

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j )
    for ( int k = 0; k < N; ++k )
      C[i+j*N] += A[i+k*N]*B[k+j*N];
```

Mémoire cache

memory bound

Le code peut être optimiser !

Mémoire interlacée

memory bound

Le code peut être optimiser !

# Opération matricielle ( suite )

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
for ( int k = 0; k < N; ++k )
  for ( int j = 0; j < N; ++j )
    for ( int i = 0; i < N; ++i )
      C[i+j*N] += A[i+k*N]*B[k+j*N];
```

Mémoire cache

memory bound/cpu bound

Mieux mais peut être encore optimiser !

Mémoire interlacée

cpu bound

Code optimisé !

# Opération matricielle ( suite... )

```
std::vector<double> A(N*N);
std::vector<double> B(N*N);
std::vector<double> C(N*N);
...
const int szBloc = 127;
// On saucissone les boucles en i et j en plusieurs morceaux
// de taille szBloc :
for ( int kb = 0; kb < N; kb += szBloc )
  for ( int jb = 0; jb < N; jb += szBloc )
    for ( int ib = 0; ib < N; ib += szBloc )
      for ( int k = kb; k < kb+szBloc; ++k )
        for ( int j = jb; j < jb+szBloc; ++j )
          for ( int i = ib; i < ib+szBloc; ++i )
            C[i+j*N] += A[i+k*N]*B[k+j*N];
```

Mémoire cache

cpu bound

Code optimisé !

Mémoire interlacée

cpu bound

Code un peu moins optimisé !

# Padding pour la mémoire interlacée

```
// Calcul la transposée de la matrice A et la range dans B
for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j ) {
     B[i+j*N] = A[j+i*N];
  }
```

- Accès des données de $A$ optimisé ( lecture linéaire )
- Pour $B$, plus délicat :
  - Si $N$ = multiple du nombre de voies : pire des accès pour $B$ !
  - Si $N$ = multiple du nombre de voies + 1 : Accès pour $B$ optimal !

# Padding pour la mémoire interlacée

```
const int nbMemVoies = ...;
int padding_N = N+(nbMemVoies + 1 - (N%nbMemVoies))%nbMemVoies;
std::vector B(N*padding_N);
...
for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j ) {
     B[i+j*padding_N] = A[j+i*N];
  }
```

- Accès optimisé pour $A$ et $B$ !

# Exécution et terminaison d'un thread

Plus simple qu'avec les posix threads :

```
#include <iostream>
#include <thread>

// Cette fonction sera appelée par un thread
void call_from_thread() {
  std::cout << "Hello, World" << std::endl;
}

int main() {
  // Exécution d'un thread
  std::thread t1(call_from_thread);

  //Join the thread with the main thread
  t1.join();

  return 0;
}
```

# Exécution d'un thread avec une lambda fonction

L'utilisation des fonctions lambdas en C++ 2011 simplifie beaucoup l'utilisation des threads :

```
#include <iostream>
#include <thread>

int main() {
  // Exécution d'un thread
  std::thread t1([] () { std::cout << "Hello, World" << std::endl; } );
  //Join the thread with the main thread
  t1.join();

  return 0;
}
```

# Exécution d'un groupe de threads

```
static const int num_threads = 10;

void call_from_thread() {
  std::cout << "Hello, World" << std::endl;
}

int main() {
  std::thread t[num_threads];
  // Exécution d'un groupe de threads
  for (int i = 0; i < num_threads; ++i) {
    t[i] = std::thread(call_from_thread);
  }
  std::cout << "Launched from the main\n";
  //Joint les threads avec le thread principal
  for (int i = 0; i < num_threads; ++i) {
    t[i].join();
  }

  return 0;
}
```

# Lancement d'un thread par CPU

```
int main(int argc, const char** argv) {
  unsigned num_cpus = std::thread::hardware_concurrency();
  std::cout << "Créer " << num_cpus << " threads\n";
  // mutex permettant un accès ordonnées à std::cout pour les threads
  std::mutex iomutex;
  std::vector<std::thread> threads(num_cpus);
  for (unsigned i = 0; i < num_cpus; ++i) {// Lambda function for thread
    threads[i] = std::thread([&iomutex, i] () {
      {
// std::lock_guard permet de bloquer un mutex pendant la durée d'un bloc
// d'instruction. Ici cela permet de bloquer le mutex uniquement durant
// l'affichage à l'aide de std::cout
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << "Thread #" << i << " is running\n";
      }
// On simule un travail important en mettant le thread un peu en pause
      std::this_thread::sleep_for(std::chrono::milliseconds(200));
    });
  }
  for (auto& t : threads) { t.join(); }
  return 0;
}
```

# Section protégée par mutex

- Pour des sections de code complexes;
- Gestion preneuse de ressources CPU;

```
std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;

void save_page(const std::string &url) {// simulate a long page fetch
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";
    g_pages_mutex.lock();
    g_pages[url] = result;
    g_pages_mutex.unlock();
}
int main() {
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join(); t2.join();
    g_pages_mutex.lock();
    for (const auto &pair : g_pages)
        std::cout << pair.first << " => " << pair.second << '\n';
    g_pages_mutex.unlock();
}
```

# Atomicité

- Pour des opérations simples sur des types de base;
- Très léger en ressources CPU.

```
# include <atomic>
void compLineMandelbrot( int i, std::vector<unsigned>& img  ) {
// Calcul 1 ligne de l'image de mandelbrot...
}
void compMandelbrot( int W, int H ) {//  mandelbrot multi-threadé :
  volatile std::atomic<int> num_line(-1);
  std::vector<unsigned> img(W*H);
  unsigned num_cpus = std::thread::hardware_concurrency();
  std::vector<std::thread> threads;
  for ( int c = 0; c < num_cpus-1; ++c ) {
    threads.push_back(std::thread([W,H,maxIter,&num_line,&img]()
      {while(num_line<H) {num_line ++;
        compLineMandelbrot(W,H,maxIter,num_line,img); } } ) );
  }
  while ( num_line < H ) {num_line ++;compLineMandelbrot(num_line,img);}
  for ( auto& t : threads ) t.join();
}
```

# Cas d'interblocage avec les mutex

```
if ( pid == 0 ) {
  S.lock();
  Q.lock();
  ...
  Q.unlock();
  S.unlock();
else if ( pid == 1 ) {
  Q.lock();
  S.lock();
  ...
  S.unlock();
  Q.unlock();
}
```

Interblocage impossible avec l'atomicité !

# Vue d'ensemble d'OpenMP

## Principe

- Intégré aux compilateurs : option de compilation (`-fopenmp` ) + directives de compilation # pragma omp ;
- Définition de régions parallèles avec déclaration variables partagées/privées
- Possibilité d'alterner entre régions séquentielles et parallèles.

## Région parallèle

- Peut être une boucle parallèle : répartition statique ou dynamique
- Plusieurs sections de code en parallèle;
- Même section de code exécutée par plusieurs threads.
- Contrôle du nombre de thread par directive ou variable d'environnement `OMP_NUM_THREADS`
- Affinité via variable d'environnement : dépend du compilateur.

# Déclaration d'une région parallèle

```
# include <iostream>
# include <omp.h>

int main() {
  float a;
  int p;
  a = 92290; p = 0;
# pragma omp parallel
  { // Début de la région parallèle
    p = omp_in_parallel();
    std::cout << " a = " << a << " et p = " << p << std::endl;
  }
  return 0;
}
```

## Sortie du programme

```
a = 92290 et p = 1
a = 92290 et p = 1
```

# Déclaration d'une région parallèle

```
# include <iostream>
# include <omp.h>

int main() {
  float a = 92290;
# pragma omp parallel default(none) private(a) shared(std::cout)
  { // Début de la région parallèle
    a = a + 290;
    std::cout << " a = " << a << std::endl;
  }
  return 0;
}
```

## Sortie du programme

```
a = 290
a = 290
```

# Déclaration d'une région parallèle

```
# include <iostream>
# include <omp.h>

int main() {
  float a = 92000;
# pragma omp parallel default(none) firstprivate(a) shared(std::cout)
  { // Début de la région parallèle
    a = a + 290;
    std::cout << " a = " << a << std::endl;
  }
  return 0;
}
```

## Sortie du programme

```
a = 92290
a = 92290
```

# Étendue d'une région parallèle

```
# include <iostream>
# include <omp.h>

void function() {
    double a = 92290;
    a += omp_get_thread_num();
    std::cout << "a = " << a << std::endl;
}
int main() {
# pragma omp parallel
{
  function();
}
  return 0;
}
```

## Sortie du programme

```
a = 92290
a = 92291
```

# Étendue d'une région parallèle

```
void function(double& a, double& b) {
    b = a + omp_get_thread_num();
    std::cout << "b = " << b << std::endl;
}

int main()
{
  double a = 92290, b;
# pragma omp parallel shared(a) private(b)
  {
      function(a,b);
  }
  return 0;
}
```

## Sortie du programme

```
b = 92290
b = 92291
```

# Allocation dynamique et région parallèle

```
# include <iostream>
# include <omp.h>

int main() {
    int nbTaches, i, deb, fin, rang, n = 1024;
    double* a;
#   pragma omp parallel
    { nbTaches = omp_get_num_threads(); }
    a = new double[n*nbTaches];
#   pragma omp parallel default(none) private(deb,fin,rang,i) \
                        shared(a,n,std::cout)
    {
        rang = omp_get_thread_num();
        deb  = rang*n; fin = (rang+1)*n-1;
        for ( i = deb; i <= fin; i++ ) a[i] = 92290. + double(i);
        std::cout << "Rang : "<<rang<< " A[" << deb << "] = " << a[deb]
                  << ", A[" << fin << "] = " << a[fin] << std::endl;
    }
    delete [] a;
    return 0;
}
```

# Réduction en OpenMP

```
int main() {
    int s = 0;
#   pragma omp parallel default(none) reduction(+:s)
    {
        s = omp_get_thread_num()+1;
    }
    std::cout << "s = " << s << std::endl;
    return EXIT_SUCCESS;
}
```

Cette application affichera `s = 3` sur deux threads.

# Division des boucles parallèles

## Parallélisation boucle

- Se fait à l'aide d'une clause `for`;
- Boucles infinies non parallélisables ( do ... while );
- Trois modes de parallélisation via `schedule` : Statique, dynamique ou guidé ( non abordé ici ).
- Par défaut, répartition statique
- Synchronisation globale à la fin de la boucle, sauf si clause `nowait` spécifié.

# Division des boucles parallèles

Parallélisation de Mandelbrot avec équilibre des tâches :

```
/* Calcul de l'ensemble de Mandelbrot en OpenMP */
std::vector<int> compMandelbrot( int W, int H, int maxIter ) {
    std::vector<int> img(W*H);
#   pragma omp parallel for schedule(dynamic)
    for ( int i = 0; i < H; ++i )
        compLineMandelbrot( W,H, maxIter, i, img );
  return img;
}
```

# Parallélisation boucles avec réduction

```
  const int n = 5;
  int i, s = 0, p = 1, r = 1;

# pragma omp parallel for reduction(+:s) reduction(*:p,r)
  for ( i = 1; i <= n; ++i ) {
      s += 1;
      p *= 2;
      r *= 3;
  }
  std::cout << "s = "<<s<< ", p = " <<p<< ", r = " <<r<< std::endl;
```

ce qui affichera à l'exécution :

```
s = 5, p = 32, r = 243;
```

# Sections parallèles

```
# pragma omp parallel private(i)
{
#  pragma omp sections nowait
   {
#     pragma omp section
      for (int i = 0; i < 10; ++i)
        std::cout << "Thread one ! ---> " << i << std::endl;
#     pragma omp section
      for (int i = -10; i < 0; i += 2)
        std::cout << "Thread two ! ---> " << i << std::endl;
    }
}
```

# clause single

Section uniquement exécutée que par un thread...

```
#pragma omp parallel default(none) private(a,rang)
{
  a = 92290.;
# pragma omp single
  {
      a = -92290.;
  }
  rang = omp_get_thread_num();
  std::cout << "rang " << rang << " : a = " << a << std::endl;
}
```

Sur deux threads :

```
rang 0 : a = 92290.
rang 1 : a = -92290.
```

# clause master

Section uniquement exécutée par le thread principal :

```
#pragma omp parallel default(none) private(a,rang)
{
  a = 92290.;
# pragma omp master
  {
      a = -92290.;
  }
  rang = omp_get_thread_num();
  std::cout << "rang " << rang << " : a = " << a << std::endl;
}
```

```
rang 0 : a = -92290.
rang 1 : a = 92290.
```

# Synchronisation

```
  double *a, *b;
  int i, n=5;
# pragma omp parallel
  {
# pragma omp single
    { a = new double[n]; b = new double[n]; }
# pragma omp master
    { for (i = 0; i < n; i++)
        a[i] = (i+1)/2.; }
# pragma omp barrier
# pragma for schedule(static)
    for (i=0; i < n; i++) b[i] = 2.*a[i];
# pragma omp single nowait
    { delete [] a; }
  }
  printf("B equal : ");
  for (i = 0; i < n; i++) printf("%7.5lg\t",b[i]);
  printf("\n");
```

# Atomicité et section critique

Même notions que pour les threads :
Atomicité :

```
  int counter = 100, rank;
# pragma omp parallel private(rank)
  {
    rank = omp_get_thread_num();
#   pragma omp atomic
    counter += 1;
    printf("Rank : %d, counter = %d\n",rank,counter);
  }
  printf("Final counter : %d\n",counter);
```

Section critique :

```
  int s = 0,p = 1;
# pragma omp parallel
  {
#   pragma omp critical
    {
      s += 1; p *= 2;
    }
  }
  printf ("s = %d, p = %d\n", s, p);
```

# Vue d'ensemble

- Bibliothèque template C++ mise en œuvre par Intel;
- License apache;
- Boucles parallèles
- Algorithmes de réduction;
- Gestion graphe de tâches en parallèle;
- Gestion concurrence entre threads;
- Politique d'allocation pour optimisation cache en multithread;
- Conteneurs ( à la STL ) supportant la concurrence.

# Exemple de code TBB

En reprenant notre éternel Mandelbrot :

```
class CompMandelbrot {
public:
  CompMandelbrot( unsigned W, unsigned H, int maxIter, int* pt_img ) :
    m_W(W), m_H(H), m_maxIter(maxIter), m_pt_img(pt_img)  {}
  void operator()( const tbb::blocked_range<int>& r ) const {
    double scaleX = 3./(m_W-1);
    double scaleY = 2.25/(m_H-1);
    for ( int i = r.begin(); i != r.end(); ++i ) {
        ...
    } }
private:
  unsigned m_W, m_H, m_maxIter;
  int* m_pt_img;
};
std::vector<int> compMandelbrot( int W, int H, int maxIter ) {
  std::vector<int> img(W*H);
  tbb::parallel_for(tbb::blocked_range<int>(0,H),
                    CompMandelbrot(W,H,maxIter,(int*)img.data()));
  return img;
}
```

# C++ 2017

## Aperçu

- Parallélisation d'algorithmes de la STL;
- Dans la norme, pas besoin d'outils externes !

```
std::vector<int> v = ...
std::sort(vec.begin(), vec.end());// standard sequential sort
using namespace std::experimental::parallel;
sort(seq, v.begin(), v.end());// explicitly sequential sort
sort(par, v.begin(), v.end());// permitting parallel execution
sort(vec, v.begin(), v.end());// permitting vectorization as well
// sort with dynamically-selected execution
size_t threshold = ...
execution_policy exec = seq;
if(v.size() > threshold)
{
exec = par;
}
sort(exec, v.begin(), v.end());
```