11. Programmation parallèle avec `OpenMP/C`

11.1 Introduction

`OpenMP` = «*Open Multi-Processing*».

- Interface de programmation **parallèle** pour architectures **à mémoire partagée**.

- Ensemble de **directives**, routines et variables

- Définie et appuyée par Compaq/Digital, HP, Intel, IBM, Silicon Graphics, Sun, USDE, etc.

- Dernière version = 4.5 (Novembre 2015)

- Fondé sur le modèle *fork/join*

  > **fork–join model***: (computer science) A method of programming on parallel machines in which one or more child processes branch out from the root task when it is time to do work in parallel, and end when the parallel work is done.*

- `OpenMP` est «*excellent for Fortran-style code written in C*» .

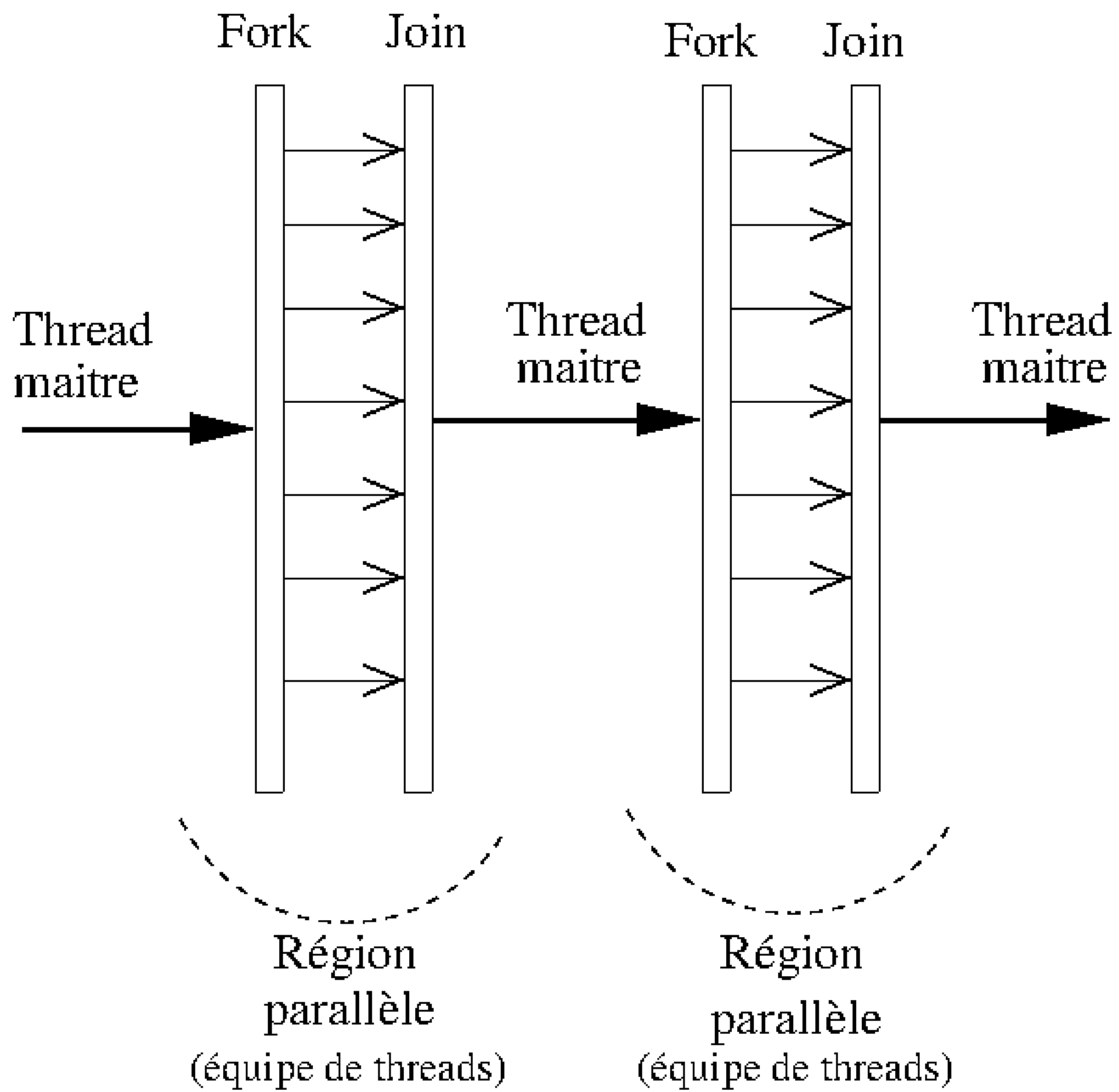

Figure 11.1: Représentation graphique du modèle «*fork/join*» à la OpenMP.

L'API OpenMP 4.5 pour C/C++ contient un grand nombre de directives et constructions :

http://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

On va voir les principales directives, qui forment l'essence d'OpenMP — si vous les comprenez, vous pourrez vous débrouiller avec le reste!

11.2 Directives et opérations de base

- Pragma de base = `omp parallel` :

```
#pragma omp parallel
{
  printf( "Hello du thread %d\n",
          omp_get_thread_num() );
  ...
  printf( "...\n" );
}
```

- Règle de base = une variable allouée **avant** le début d'une région parallèle est **partagée** par les *threads*.

```c
int x = 0;  // Variable partagee entre threads

#pragma omp parallel
{
  // Acces, non protege?!, a une var. partagee
  x = x + 1;

  // Variable y locale a chaque thread.
  int y = x + 1;

  ...
}
```

```c
int x = 0;

#pragma omp parallel private( x )
{
  // Copie locale de x.
  x = x + 1;

  // Variable y locale a chaque thread.
  int y = x + 1;

  ...
}
```

- Différentes façons d'indiquer le nombre de *threads* :

```
#  pragma omp parallel num_threads(10)
```

```
omp_set_num_threads(10);
```

```
export OMP_NUM_THREADS=10
```

Dans tous les cas = «**suggestion**»

- Clauses de **partage de travail** :
  Rôle = distribuer le travail à faire **entre** les *threads* d'une équipe.

- Répartition des itérations d'un `for` (*loop splitting*)

```
#pragma omp for
for( ... ) {
   ...
}
```

Note : Ne crée pas une équipe de *threads* — répartit les itérations entre les *threads* déjà actifs!

– Distribution par sections

```
#pragma omp sections
{

#    pragma omp section
     { ... code pour 1ere section,
           executee par un thread ... }

#    pragma omp section
     { ... code pour 2e section,
           executee par un autre thread ... }
     ...

}
```

- Création dynamique de tâches : Voir plus loin.

Deux petits exemples

Premier exemple :

```
void foo( int n, int nb_threads ) {
  printf( "foo( %d, %d )\n", n, nb_threads );

  omp_set_num_threads( nb_threads );

# pragma omp parallel
  for( int i = 0; i < n; i++ ) {
    int id = omp_get_thread_num();
    printf( "i = %d: id = %d\n", i, id );
  }
}
```

Quel est l'effet d'un appel à `foo( 5, 3 )`?

Exercice 11.1: Effet d'un appel à `foo( 5, 3 )`.

Deuxième exemple :

```c
void foo( int n, int nb_threads ) {
  printf( "foo( %d, %d )\n", n, nb_threads );

  omp_set_num_threads( nb_threads );

# pragma omp parallel
# pragma omp for
  for( int i = 0; i < n; i++ ) {
    int id = omp_get_thread_num();
    printf( "i = %d: id = %d\n", i, id );
  }
}
```

Quel est l'effet d'un appel à `foo( 5, 3 )`?

Exercice 11.2: Effet d'un appel à `foo( 5, 3 )`.

– Abréviation :

```
# pragma omp parallel
# pragma omp for
  for( ... )
```

Même effet :

```
# pragma omp parallel for
  for( ... )
```

– Boucle utilisée pour effectuer une *réduction*

```
#  pragma omp for reduction( <op>:  var )
```

Opérations possibles : `+`, `*`, `-`, `&`, `|`, `&&`, `||`,`^`, `min`, `max`.

Remarque : Depuis la version 4.0, il est aussi possible d'utiliser une opération de réduction **définie par le programmeur**.

11.3 Autres directives de synchronisation

- Région critique :

```
# pragma omp critical
```

- Exécution *unique* : c'est le premier *thread* arrivé qui exécute l'instruction :

```
#  pragma omp single
```

- Exécution unique par le *thread* maître :

```
#  pragma omp master
```

- Barrière explicite de synchronisation pas dans une région de partage de travail :

```
#  pragma omp barrier
```

- Accès **atomique** à une variable simple:

```
# pragma omp atomic
  x = x  <op>  <expr>
```

Valeurs possibles pour *<op>* :
`+`, `*`, `-`, `&`, `|`, `&&`, `||`, `^`, `<<`, `>>`

Note : Les opérations **atomiques** arithmétiques et logiques peuvent être mises en oeuvre, sur les machines modernes, **sans utiliser de verrous**!

Note :

- Les formes abrégées sont permises :
  «`x++`», «`x += 1`», etc.

- On peut spécifier le mode d'accès :
  `read`, `write`, `update`, `capture`.

  Mode par défaut = `update`

  L'utilisation de `read/write` assure l'atomicité, peu importe la taille des mots de la machine

- Mode `capture` :
  «`v = x++`» ≡ «`{v = x; x += 1;}`»

- Instructions explicites de synchronisation.

```
omp_init_lock( omp_loc_t *lock )
omp_set_lock( omp_loc_t *lock )
omp_unset_lock( omp_loc_t *lock )
omp_test_lock( omp_loc_t *lock )
omp_init_destroy( omp_loc_t *lock )
```

11.4 Clauses de distribution du travail entre les *threads* dans les boucles

Permettent de spécifier de quelle façon les diverses itérations d'une boucle **sont réparties entre les *threads***.

```
schedule( static [,chunk] )

schedule( dynamic [,chunk] )

schedule( guided [,chunk] )

schedule( runtime )

schedule( auto )
```

- `guided` : répartition dynamique entre les *threads*, mais avec un comportement qui varie en cours d'exécution.

  «*guided self-scheduling*»

  > *Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies the minimum size chunk to use.*
  >
  > *https://software.intel.com/en-us/articles/openmp-loop-schedul*

- `runtime` : définie par l'intermédiaire de la variable d'environnement `OMP_SCHEDULE`.

- `auto` : décision laissée au compilateur!

11.5 Création dynamique de tâches

Depuis la **version 3.0**, il est possible de créer dynamiquement des tâches

```
#   pragma omp task shared(r_foo1)
    r_foo1 = foo1( ... ); // Tache appel a foo1
#   pragma omp task shared(r_foo2)
    r_foo2 = foo2( ... ); // Tache appel a foo2
    .
    .
    .
#   pragma omp taskwait
    r = bar(r_foo1,...,r_foo2) // Attente taches
```

11.6 Exemples

Les exemples qui suivent sont adaptés de deux articles provenant du site Web de Sun/Oracle :

- «*Introducing OpenMP: A Portable, Parallel Programming API for Shared Memory Multiprocessors*»
- «*OpenMP Support in Sun Studio Compilers and Tools*»

11.6.1 Directive `parallel`

```c
int main(void)
{
  omp_set_dynamic(0);
  omp_set_num_threads(10);

# pragma omp parallel
  {
    /* Obtain thread ID. */
    int tid = omp_get_thread_num();

    /* Print thread ID. */
    printf("Hello World from thread = %d\n", tid);
  }
}
```

11.6.2 Directive for

```c
int main(void)
{
  float a[N], b[N], c[N];

  omp_set_dynamic(0);
  omp_set_num_threads(20);

  /* Initialize arrays a and b. */
  for( int i = 0; i < N; i++ ) {
      a[i] = i * 1.0;
      b[i] = i * 2.0;
  }

  /* Compute values of array c in parallel. */
# pragma omp parallel for
  for( int i = 0; i < N; i++ ) {
    c[i] = a[i] + b[i];
  }
  printf("%f\n", c[10]);
}
```

11.6.3 Directive sections

```
int square(int n) { return n*n; }

int main(void)
{
  int x, y, z, xs, ys, zs;
  omp_set_dynamic(0);
  omp_set_num_threads(3);
  x = 2;  y = 3;  z = 5;

# pragma omp parallel sections
  {
#   pragma omp section
    { xs = square(x);
      printf("id = %d, xs = %d\n", omp_get_thread_
    }
#   pragma omp section
    { ys = square(y);
      printf("id = %d, ys = %d\n", omp_get_thread_
    }
#   pragma omp section
    { zs = square(z);
      printf("id = %d, zs = %d\n", omp_get_thread_
    }
  }
}
```

11.6.4 Directive `for` avec réduction

Version Séquentielle

```
int sum = 0;

for( int i = 0; i < n; i++ ) {
  sum += some_complex_long_function(a[i]);
}
```

Avec section critique

```
int sum = 0;

# pragma omp parallel for shared(sum, a, n)
for( int i = 0; i < n; i++ ) {
  int value = some_complex_long_function(a[i]);

# pragma omp critical
  sum += value;
}
```

Est-ce que cette solution sera efficace?

Exercice 11.3: Utilisation de `critical`.

Avec opération atomique

```
int sum = 0;

# pragma omp parallel for shared(sum, a, n)
for( int i = 0; i < n; i++ ) {
  int value = some_complex_long_function(a[i]);

# pragma omp atomic
  sum += value;
}
```

Est-ce que cette solution sera efficace?

Exercice 11.4: Utilisation d'`atomic`.

Autre version équivalente (la plus simple et courte)

```
int sum = 0;

#pragma omp parallel for reduction(+: sum)
for( int i = 0; i < n; i++ ) {
  sum += some_complex_long_function(a[i]);
}
```

Est-ce que cette solution sera efficace?

Exercice 11.5: Utilisation de `reduction`.

11.6.5 Directives task et taskwait

```
int fibo( int n )
{
  if ( n <= 1 ) {
    return 1;
  } else {
    int r1, r2;

#   pragma omp task shared(r1)
    // Le shared est obligatoire!
    // (les regles sont differentes pour cette directive)
    r1 = fibo( n-1 );

#   pragma omp task shared(r2)
    r2 = fibo( n-2 );

#   pragma omp taskwait
    return r1 + r2;
  }
}
```

```
int main( int argc, char *argv[] ) {
  assert( argc >= 3 );
  int n = atoi( argv[1] );
  int nb_threads = atoi( argv[2] );

  omp_set_dynamic(0);
  omp_set_num_threads( nb_threads );

# pragma omp parallel
  {
#   pragma omp single
    printf( "fibo(%d) = %d\n", n, fibo(n) );
  }
  return( 0 );
}
```

Que se passe-t-il si on omet la clause «`#pragma omp single`»?

Exercice 11.6: Utilisation `single`.

11.A Modèles avec *fork/join* ou *threads* explicites vs. implicites

Comparaisons OpenMP, `PRuby`, `Ruby`, `Java` vs. `C` :

- `fork` : généralement explicite — pour lancer le *thread*!

- `join` : explicite ou implicite

- Code du *thread* : λ-expression (`PRuby`/`Ruby`, `Java`), fonction explicite (`C`) ou code arbitraire (OpenMP)

Exemple

Soit le code séquentiel suivant :

```
def f1( x ); ...; end
def f2( x ); ...; end


r = Array.new(N)

r.each_index do |k|
  r[k] = f1(k) + f2(k)
end
```

On veut paralléliser ce code.

Il s'agit d'un problème *embarrassingly parallel*, et on veut définir une solution à granularité (très) fine.

MPD

(Avec *thread* «explicite», *join* implicite.)

```
procedure f1_f2( int k ) returns int r
{
  r = f1(k) + f2(k)
}



int r[N]

co [k = 0 to N-1]    # Co-begin/co-end.
  r[k] = f1_f2(k)
oc
```

$\Rightarrow$ Il est nécessaire, en MPD, d'introduire une procédure auxiliaire qui sera exécutée par le *thread* ☹

PRuby

(Avec *thread* explicite (λ-expression), *join* implicite.)

```
r = Array.new(N)

PRuby.pcall( 0...N,
             lambda do |k|
               r[k] = f1(k) + f2(k)
             end
           )
```

OpenMP/C

(Avec *thread* implicite, *join* implicite.)

```
omp_set_num_threads(N);

int* r = (int*) malloc(N * sizeof(int));

# pragma omp parallel for schedule( static, 1 )
for ( int k = 0; k < N; k++ ) {
  r[k] = f1(k) + f2(k);
}
```

PRuby (bis)

(Avec *thread* explicite (bloc), *join* explicite.)

```
r = Array.new(N)

futures = (0...N).map do |k|
  PRuby.future { f1(k) + f2(k) }
end

r.each_index do |k|
  r[k] = futures[k].value
end
```

OpenMP/C

(Avec *thread* implicite, *join* implicite.)

```
omp_set_num_threads(N);

int* r = (int*) malloc(N * sizeof(int));

# pragma omp parallel for schedule( static, 1 )
for ( int k = 0; k < N; k++ ) {
  r[k] = f1(k) + f2(k);
}
```

PRuby (ter)

(Avec *thread* implicite, *join* implicite.)

```
PRuby.nb_threads = N

r = Array.new(N)

r.peach_index( static: 1 ) do |k|
  r[k] = f1(k) + f2(k)
end
```

OpenMP/C

(Avec *thread* implicite, *join* implicite.)

```
omp_set_num_threads(N);

int* r = (int*) malloc(N * sizeof(int));

# pragma omp parallel for schedule( static, 1 )
for ( int k = 0; k < N; k++ ) {
  r[k] = f1(k) + f2(k);
}
```

Java (avec `Future` et lambda-expression)

(Avec *thread* explicite, *join* explicite.)

```
ExecutorService pool
  = Executors.newCachedThreadPool();

int[] r = new int[N];

Future<Integer>[] fs = new Future[N];
for( int k = 0; k < N; k++ ) {
    final int kf = k;
    fs[k] = pool.submit(
        () -> f1(kf) + f2(kf)
    );
}

for( int k = 0; k < N; k++ ) {
    try {
      r[k] = fs[k].get();
    } catch( Exception e ){...};
}

pool.shutdown();
```

C/Pthreads

(Avec *thread* explicite, *join* explicite.)

```
void *f1_f2( void *arg )
{
  int k = (int) arg;
  int resultat = f1(k) + f2(k);

  pthread_exit( (void*) resultat );
}

pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)

int *r = (int *) malloc(N * sizeof(int));

pthread_t *trIds
   = (pthread_t *) malloc(N * sizeof(pthread_t));
for ( int k = 0; k < N; k++ ) {
  pthread_create( &trIds[k], &attr,
                  f1_f2, (void *) k );
}

for ( int k = 0; k < N; k++ ) {
  pthread_join( trIds[k], (void *) &r[k] );
}
```

Langage	`fork`	`join`	Code du *thread*
`PRuby`	Explicite	Implicite	lambda/bloc
`Ruby`	Explicite	Explicite	bloc
`Java`	Explicite	Explicite	lambda
`C`	Explicite	Explicite	fonction
`OpenMP`	Région	Implicite	code arbitraire

11.A.1 Comparaisons du temps pour appeler une méthode vs. créer/activer un *thread*

Question : Si on compare le temps requis pour faire un appel de méthode direct vs. un appel via un *thread*, combien de fois plus long est l'appel via un *thread*?

```ruby
# Programme Ruby.

N = ARGV[0] ? ARGV[0].to_i : 100_000

def inc( x )
  x + 1
end

def executer( bm, nom_methode )
  GC.start; GC.disable

  bm.report( nom_methode )do
    N.times { yield }
  end

  GC.enable
end

puts "Temps pour faire #{N} appels..."

Benchmark.bm(30) do |bm|
  x = 0
  executer bm, "d'une methode simple" do
    x = inc( x )
  end

  x = 0
  executer bm, "d'un Thread style future" do
    x = Thread.new { x + 1 }.value
  end
end
```

```
= MAC BOOK =============

Ruby MRI
--------
Temps pour faire 100_000 appels...
                                      user     system      total        real
d'une methode simple              0.020000   0.000000   0.020000 (  0.014103)
d'un Thread style future          1.290000   2.780000   4.070000 (  3.078120)


=> Thread 218 fois plus long que methode!?


JRuby
-----
Temps pour faire 100_000 appels...
                                      user     system      total        real
d'une methode simple              0.200000   0.010000   0.210000 (  0.067000)
d'un Thread style future         16.530000   9.610000  26.140000 ( 17.413000)
d'un ForkJoin::Task               1.530000   0.990000   2.520000 (  1.371000)


=> Thread 259 fois plus long que methode!?
=> Thread 10 fois plus long que ForkJoin::Task!?
=> ForkJoin::Task 21 fois plus long que methode!?
```

```ruby
class FJFuture < ForkJoin::Task
  def initialize( body )
    @body = body
  end

  def call
    @r = @body.call
  end

  def value
    join
    @r
  end
end


##################

  pool = ForkJoin::Pool.new
  x = 0
  executer bm, "d'un ForkJoin::Task" do
    f = pool.submit FJFuture.new( lambda { x + 1 } )
    x = f.value
  end
```

```
= JAPET ===========================================================

Ruby MRI
--------
Temps pour faire 100000 appels...
                                     user     system      total        real
d'une methode simple             0.020000   0.000000   0.020000 (  0.015892)
d'un Thread style future         2.350000   5.230000   7.580000 (  6.809706)


=> Thread 428 fois plus long que methode!?


JRuby
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.320000   0.010000   0.330000 (  0.048000)
d'un Thread style future       246.240000  19.970000 266.210000 ( 51.394000)


=> Thread 1000 fois plus long que methode!?
```

```java
// Programme Java.
class MethodeVsThread {
  private int x = 0;

  void inc() { x = x + 1; }

  public static void main( String[] args ) {
    ...

    // Appels de methodes.
    MethodeVsThread o = new MethodeVsThread();
    tempsDebut = System.currentTimeMillis();
    o.x = 0;
    for ( int i = 0; i < N; i++ ) {
      o.inc();
    }
    tempsFin = System.currentTimeMillis();

    // Creation de threads.
    tempsDebut = System.currentTimeMillis();
    o.x = 0;
    for ( int i = 0; i < N; i++ ) {
      Thread t = new Thread( () -> { o.inc(); } );
      t.start();
      try { t.join(); } catch( Exception e ){};
    }
    tempsFin = System.currentTimeMillis();

    // Creation de Futures avec pool de threads.
    final ExecutorService pool = Executors.newCachedThreadPool();
    tempsDebut = System.currentTimeMillis();
    o.x = 0;
    for ( int i = 0; i < N; i++ ) {
      Future<?> t = pool.submit( () -> { o.inc(); } );
      try { t.get(); } catch( Exception e ){};
    }
    tempsFin = System.currentTimeMillis();
    ...
  }
}
```

```
Temps pour 1000000 appels de methodes (ms)
      9


Temps pour 1000000 creation/activation de threads (ms)
  55981
```

=> 6000 fois plus long

```
Temps pour 1000000 creation/activation de taches (ms)
  11334
```

=> 1000 fois plus long

11.B Exercices

11.B.1 Traitement d'une liste chainée

```
typedef struct Noeud {
  struct Noeud *suivant;
  long valeur;
} Noeud;

void foo( Noeud* pt ) {
  // On traite le noeud et sa valeur.
  ...
  // On imprime une trace.
  printf( "foo( %p ): valeur = %d\n", pt, pt->valeur );
}

...

// tete = reference vers une liste avec 6 elements.
for( Noeud* pt = tete; pt != NULL; pt = pt->suivant ) {
  foo( pt );
}
```

L'exécution produit le résultat suivant :

```
foo( 0xe900d0 ): valeur = 5
foo( 0xe900b0 ): valeur = 4
foo( 0xe90090 ): valeur = 3
foo( 0xe90070 ): valeur = 2
foo( 0xe90050 ): valeur = 1
foo( 0xe90030 ): valeur = 0
```

On veut paralléliser la boucle `for`. Quels résultats produiront chacune des séries d'annotations ci-bas.

1.
```
#pragma omp parallel for
for( Noeud* pt = tete; pt != NULL; pt = pt->suivant )
  foo( pt );
}
```

2.
```
#pragma omp parallel
for( Noeud* pt = tete; pt != NULL; pt = pt->suivant )
  #pragma omp task
  foo( pt );
  #pragma omp taskwait
}
```

3.
```
#pragma omp parallel
#pragma omp single
for( Noeud* pt = tete; pt != NULL; pt = pt->suivant )
  #pragma omp task
  foo( pt );
  #pragma omp taskwait
}
```

4.
```
#pragma omp parallel
#pragma omp single
for( Noeud* pt = tete; pt != NULL; pt = pt->suivant )
  #pragma omp task
  foo( pt );
}
#pragma omp taskwait
```

Exercice 11.7: Parallélisation du traitement des éléments d'une liste chainée.