13. Programmation parallèle avec C++ et les *Threading Building Blocks* d'Intel

13.1 Introduction

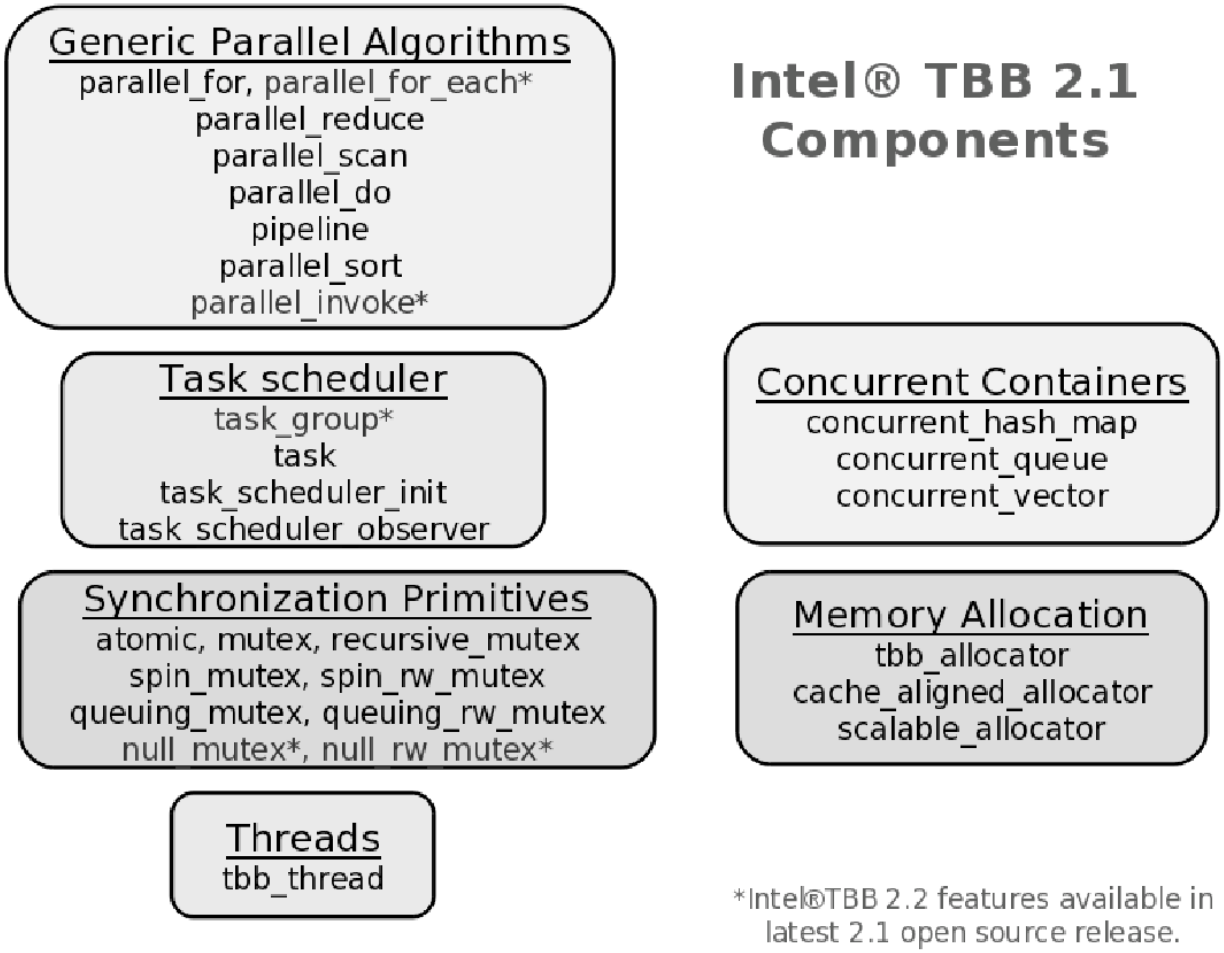

Figure 13.1: Les principaux composants de TBB.

13.1 Introduction

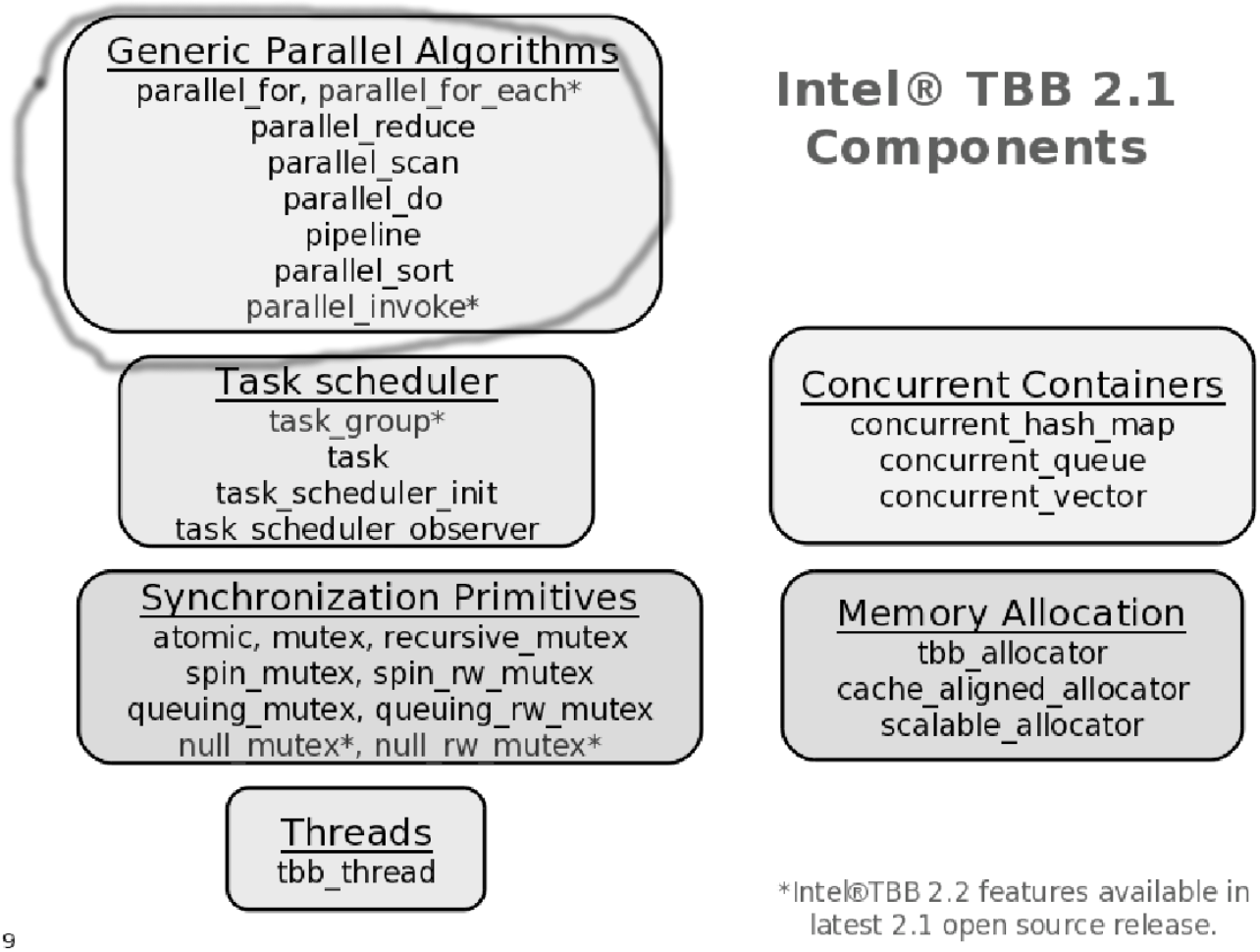

Figure 13.1: Les principaux composants de TBB.

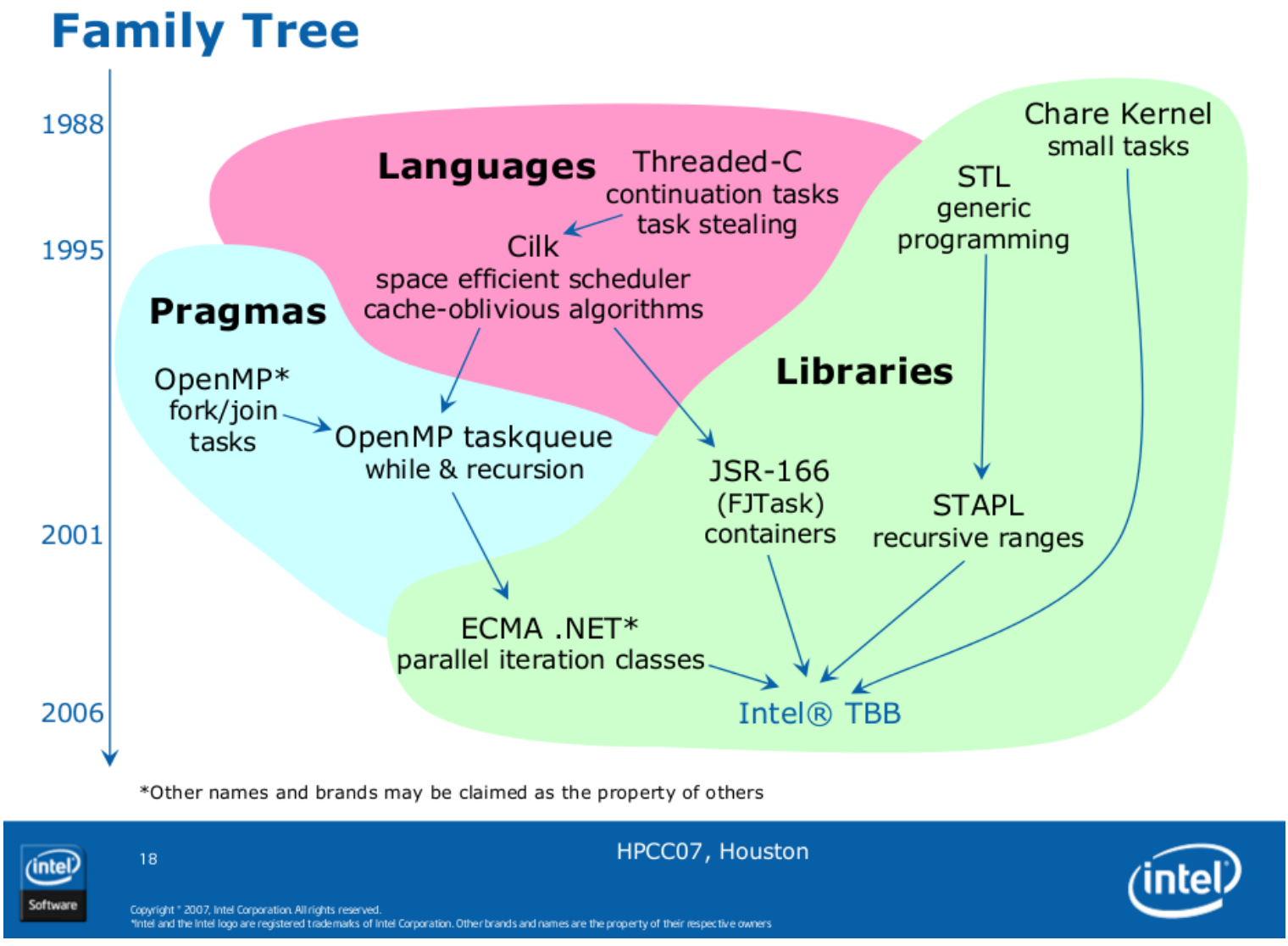

Figure 13.2: Historique de TBB (selon Intel).

- La notion fondamentale de TBB = «**tâche**» — et non «*thread*»

Philosophie : On expose autant de parallélisme que possible, et on laisse TBB «*choose how much of that parallelism is actually exploited*» .

- TBB met l'accent sur le **parallélisme de données**, tout en permettantle **parallélisme de contrôle** et le **parallélisme de flux**

- TBB repose sur la décomposition *récursive* diviser-pour-régner dichotomique

- TBB est une **bibliothèque** en C++:
  - Aucun compilateur spécial ;
  - Fonctionne sur n'importe quel système ayant un compilateur C++.

- TBB utilise des **templates** pour exprimer les patrons de programmation parallèle : de boucles, de flux, récursif , etc.

13.2 Quelques éléments de C++

Pour comprendre les *Threading Building Blocks*, il faut connaitre certains éléments plus avancés de C++, par ex., les *templates*, les λ-expressions, mais aussi le passage de paramètre par référence.

13.2.0 Passage de paramètres par valeur vs. par référence

Call by reference (also referred to as pass by reference) is an evaluation strategy where a function receives an implicit reference to a variable used as argument, rather than a copy of its value.

This typically means that the function can modify (i.e. assign to) the variable used as argument—something that will be seen by its caller.

`https://en.wikipedia.org/wiki/Evaluation_strategy`

Quel mécanisme est utilisé pour le passage des arguments en Java?

Indice : Voici quelques mécanismes utilisés dans divers langages de programmation :

- Par valeur (*copy-in*) ;
- Par résultat (*copy-out*) ;
- Par valeur/résultat (*copy-in/copy-out*) ;
- Par référence ;
- Par nécessité (*call-by-need*) ;
- Par nom (*call-by-name*) ;

	Val.	**Rés.**	**Val./Rés.**	**Réf.**	***Need***	**Nom**
ALGOL-60	X					X
FORTRAN				X		
Pascal	X		X			
C	X					
C++	X			X		
Ada	X	X	X			
Haskell					X	

Exercice 13.0: Passage des arguments en Java.

1. Pour chacun des programmes C++ ci-bas, indiquez ce qui sera affiché si on compile (avec `g++`) puis on exécute.

   Dans tous les cas, on suppose qu'une instruction «`#include <cstdio>`» est présente au début du fichier.

2. Pour le programme `pgm5.cpp`, est-ce que les deux versions de la fonction `incinc` produisent toujours le même effet?

3. En `FORTRAN`, tous les paramètres sont toujours passés par référence.
   Qu'est-ce qui était imprimé en `FORTRAN-77` par le programme suivant — dans l'appel à `PRINT`, «`*`» dénote la sortie standard (*stdout*) :

```
FUNCTION inc( x )
  x = x + 1
END

inc( 0 )

PRINT *, "0 = ", 0
```

Exercice 13.1: Passage par valeur vs. par référence.

pgm1.cpp

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

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

int main( int argc, char *argv[] )
{
  int x = 0;

  inc( x ); printf( "%d\n", x );

  inc( &x ); printf( "%d\n", x );
}
```

pgm2.cpp

```
void inc( int& x ) { x += 1; }

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

int main( int argc, char *argv[] )
{
  int x = 0;

  inc( x ); printf( "%d\n", x );

  inc( &x ); printf( "%d\n", x );
}
```

pgm3.cpp

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

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

int main( int argc, char *argv[] )
{
  int x = 0;

  inc( x ); printf( "%d\n", x );

  inc( x ); printf( "%d\n", x );
}
```

pgm4.cpp

```
void inc( int& x ) { x += 1; }

int main( int argc, char *argv[] )
{
  int x = 0;

  inc( 1 ); printf( "%d\n", x );
}
```

pgm5.cpp

```
void incinc( int* x_, int* y_, int n  ) {
  int x = *x_, y = *y_; // Copy-in (=> copies locales).
  for( int i = 0; i < n; i++ ) {
    x++; y++;  // Traitement sur copies locales.
  }
  *x_ = x; *y_ = y;  // Copy-out (=> met a jour argument
}

void incinc( int& x, int& y, int n  ) {
  for( int i = 0; i < n; i++ ) {
    x++; y++;
  }
}

int main( int argc, char *argv[] )
{
  int x, y;

  x = y = 0;
  incinc( &x, &y, 10 ); printf( "%d %d\n", x, y );
  x = y = 0;
  incinc( x, y, 10 ); printf( "%d %d\n", x, y );
}
```

13.2.1 Les *templates* génériques

Les **templates** permettent de définir des algorithmes, opérations ou types **génériques**, c'est-à-dire, paramétrisés par des types.

Programme C++ 13.1 Une procédure générique pour `echanger` le contenu de deux variables.

```
template<typename T>
void echanger( T& x, T& y ) {
  T tmp = x;
  x = y;
  y = tmp;
}

// Exemples d'utilisation: Avec types explicites.
int x, y;
...
echanger<int>( x, y );
// void echanger( int&, int& );

float a, b;
...
echanger<float>( a, b ); // void echanger( float&,


// Exemples d'utilisation: Sans types,
//            => inference des types par le compilat

echanger( x, y );
// => void echanger( int&, int& );

echanger( a, b );
// => void echanger( float&, float& );
```

Programme C++ 13.1 Une utilisation incorrecte de `echanger`...

```
template<typename T>
void echanger( T& x, T& y ) {
  T tmp = x;
  x = y;
  y = tmp;
}

// Autre exemple d'utilisation avec types implicites.
int x = 1;
float a = 2.0;
...
echanger( x, a );

...
```

```
$ /usr/bin/g++ -m64 -ltbb -std=c++11 echanger.cpp -o echanger
```

Programme C++ 13.1 Une utilisation incorrecte de `echanger`...

```
template<typename T>
void echanger( T& x, T& y ) {
  T tmp = x;
  x = y;
  y = tmp;
}

// Autre exemple d'utilisation avec types implicites.
int x = 1;
float a = 2.0;
...
echanger( x, a );

...
```

```
$ /usr/bin/g++ -m64 -ltbb -std=c++11 echanger.cpp -o echanger
echanger.cpp: In function 'int main(int, char**)':
echanger.cpp:32:18: erreur: no matching function for call to 'echanger(int&, float&)'
   echanger( x, a );
                  ^
echanger.cpp:32:18: note: candidate is:
echanger.cpp:4:6: note: template<class T> void echanger(T&, T&)
 void echanger( T& x, T& y ) {
      ^
echanger.cpp:4:6: note: template argument deduction/substitution failed:
echanger.cpp:32:18: note: deduced conflicting types
                          for parameter 'T' ('int' and 'float')
   echanger( x, a );
                  ^
```

Que fait le programme ci-bas?

```
#include <string>

template <int N>
struct Foo
{
  enum { val = N * Foo<N-1>::val };
};

template <>
struct Foo<0>
{
  enum { val = 1 };
};

int main( int argc, char *argv[] )
{
  const int N = 10;
  printf( "%d => %d\n", N, Foo<N>::val );
  return 0;
}

----------------------------------------

$ /usr/bin/g++ -m64 -ltbb -std=c++11 foo.cpp -o foo
$ foo
10 => 3628800
```

Exercice 13.2: Une utilisation non-triviale — et quelque peu étonnante! — des templates.

13.2.2 Les lambda-expressions

Un **objet-fonction** — appelé aussi un **foncteur** — est **un objet possédant une méthode «`operator()`».**

Une λ-expression est une expression qui génère un **foncteur anonyme**.

La syntaxe générale :

```
[capture]( parametres_formels ) -> type_du_resultat {
  corps
}
```

On peut utiliser `auto` pour le type implicite d'une variable qui est une lambda-expression :

```
auto plusX = [x]( int y ){ return x + y; };
```

Programme C++ 13.2 Exemples de définition et d'utilisation de λ-expressions.

```
int x, r;

// Lambda-expressions sans argument :
//    capture par valeur.
x = 9;
auto l1 = [x]{ return x + 1; };
r = l1();
assert( x == 9 && r == 10 );

r = [x]{ return x + 1; }();
assert( x == 9 && r == 10 );

r = [=]{ return x + 1; }();
assert( x == 9 && r == 10 );


// Lambda-expressions sans argument :
//    capture par reference.
x = 9;
auto l2 = [&x]{ return x++; };
r = l2();
assert( x == 10 && r == 9 );

r = [&]{ return x++; }();
assert( x == 11 && r == 10 );
```

```
// Lambda-expressions avec argument.
x = 9;
r = [=]( int y ){ return x + y; }( 25 );
assert( x == 9 && r == 34 );

x = 9;
r = [&]( int y ){ x += 2; return x + y; }( 25 );
assert( x == 11 && r == 36 );

x = 9;
r = 0;
[x, &r]( int y ){ r = x + y; }( 25 );
assert( x == 9 && r == 34 );
```

```
r = [=]( int y ){ x += 2; return x + y; }( 25 );

/*
lambdas.cpp: In lambda function:
lambdas.cpp:11:23: erreur:
        assignment of read-only variable 'x'
     r = [=]( int y ){ x += 2; return x + y; }( 1
*/
```

```
// Si on a indique #include <functional>
x = 11;
std::function<int (int)> plusX
  = [x]( int y ){ return x + y; };

assert( plusX(100) == 111 );


// On peut aussi utiliser auto comme type.
x = 9;
auto plusXbis = [x]( int y ){ return x + y; };

assert( plusXbis(12) == 21 );
```

13.2.3 Structures de contrôle définies par le programmeur avec λ-expressions

Les *templates* génériques et les lambda-expressions (et, plus généralement, les foncteurs) peuvent être utilisés pour définir de **nouvelles structures de contrôle**.

Pour chacun des segments de code C++ ci-bas, indiquez ce qui sera affiché si on compile (avec g++) puis on exécute.

1. Capture par référence d'un tableau statique :

```
int a[n];
auto init_zero = [&](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

2. Capture par valeur d'un tableau statique :

```
int a[n];
auto init_zero = [=](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

3. Capture par valeur d'un tableau passé en argument :

```
void init_zero(int a[], int i) {
  [=](int i){ a[i] = 0; }( i );
}

int a[n];
init_zero(a, 0);
printf( "%d\n", a[0] );
```

4. Capture par valeur d'un tableau dynamique :

```
int* a = (int*) malloc(n * sizeof(int));
auto init_zero = [=](int i) { a[i] = 0; };
init_zero(0);
printf( "%d\n", a[0] );
```

Exercice 13.3: Capture par valeur vs. par référence.

Programme C++ 13.3 Une structure de contrôle `PourChaqueElement` (itération séquentielle), utilisée avec des λ-expressions.

```
template<typename Functor>
void PourChaqueElement( int inf,
                        int sup,
                        Functor f ) {
  for( int i = inf; i < sup; i++ ) {
    f(i);
  }
}

// Exemples d'utilisation.
int a[4] = {10, 20, 93, 12};

PourChaqueElement( 0, 4,
    [&]( int i ){ a[i] += 1; }
);
// a == {11, 21, 94, 13}


le_max = 0;
PourChaqueElement( 0, 4,
    [a, &le_max]( int i ) {
       le_max = a[i] > le_max ? a[i] : le_max;
    }
);
// le_max == 94
```

13.3 Parallélisme de boucles : `parallel_for`

13.3.1 La spécification du `parallel_for`

La méthode générique pour le parallélisme de boucle est `parallel_for`, dont voici 2 parmi les 3 signatures possibles :

```
template<typename Index, typename Func>
  Func parallel_for( Index first,
                     Index_type last,
                     const Func& f );
```

```
template<typename Range, typename Body>
   void parallel_for( const Range &range,
                      const Body &body )
// Parallel iteration over Range with default
// partitioner.
```

```
PROCEDURE parallel_for( first: Index,
                        last: Index,
                        func: Func )
DEBUT
  EN PARALLELE POUR i ← first A last-1 FAIRE
    func(i)
  FIN
FIN
```

Algorithme 13.1: Description *informelle* de l'effet d'un `parallel_for` avec des index.

Remarque :

> *There is no guarantee that the iterations run in parallel. Deadlock may occur if a lesser iteration waits for a greater iteration.*
>
> Source : `https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/parallel_for_func.htm`

```
PROCEDURE parallel_for( r: Range,
                        body: Body )
DEBUT
  sr ← decomposerEnSousIntervallesDisjoints(r)
  k ← sr.size()
  ASSERT: ∀_{i,j∈{1..k}} • i ≠ j ⇒ sr[i] ∩ sr[j] = {}
  ASSERT: ⋃_{1≤i≤k} sr[i] = r

  EN PARALLELE POUR i ← 1 A k FAIRE
    body(sr[i])
  FIN
FIN
```

Algorithme 13.2: Description *informelle* — et *conceptuelle* — de l'effet d'un `parallel_for` avec un `Range`.

Exemple «à la Ruby» pour le `Range` et sa décompositon en sous-`Range` :

- Soit l'appel suivant :
  $sr = decomposerEnSousIntervallesDisjoints(0...100)$

- Divers résultats possibles pour sr :

  $sr = [0...50, 50...100]$

  $sr = [0...20, 20...40, 40...60, 60...80, 80...100]$

  $sr = [0...10, 10...20, 20...30, \ldots, 80...90, 90...100]$

  $sr = [0...1, 1...2, 2...3, \ldots, 97...98, 98...99, 99...100]$

  $sr = [0...20, 20...60, 60...70, 70...100]$

```ruby
# Definition abstraite de parallel_for
def TBB.parallel_for( range, body )
  sr = decomposer_en_sous_intervalles_disjoints(range)
  k = sr.size

  (0...k).peach do |i|
    body.call( sr[i] )
  end
end

# Exemple d'utilisation du parallel_for
a = ...
b = ...
c = Array.new( N )

TBB.parallel_for( 0...N,
                  lambda do |range|
                    range.each do |i|
                      c[i] = a[i] + b[i]
                    end
                  end
                  )
```

Remarque :

> *When worker threads are available,* `parallel_for` *executes iterations in non-deterministic order. [...] However, for efficiency, do expect* **parallel_for** *to tend towards operating on consecutive runs of values.*

Source : `https://www.threadingbuildingblocks.org/docs/help/reference/algorithms/parallel_for_func.htm`

Programme TBB **13.1** Spécification d'un `Range` — utilisé dans le *template* du `parallel_for`.

```
// Class R implementing the concept of range
//          must define:

 R::R( const R& );
// Copy constructor

 R::~R();
// Destructor

 bool R::is_divisible() const;
// True if range can be partitioned into two subra

 bool R::empty() const;
// True if range is empty

 R::R( R& r, split );
// Split range r into two subranges.
```

Programme TBB **13.2** Spécification d'un `Body` — utilisé dans le *template* du `parallel_for`.

```
// Class Body implementing the concept
//       of parallel_for body must define:

Body::Body( const Body& );
// Copy constructor

Body::~Body();
// Destructor

void Body::operator()( Range& r ) const;
// Function call operator applying
//           the body to range r.
```

```
template<typename Range, typename Body>
   void parallel_for( const Range &range,
                      const Body &body )
// Parallel iteration over range
// with default partitioner.
```

13.3.2 Exemples : Les deux formes de `parallel_for`

Le programme `TBB` 13.3 présente une utilisation de
la première et de la deuxième forme de `parallel_for`,
pour **incrémenter les éléments d'un tableau**.

Programme TBB 13.3 Deux exemples simples d'utilisation d'un `parallel_for`.

```
// Premiere forme: avec bornes inferieure et super
int a[4] = {10, 20, 93, 12};
parallel_for( 0, 4,
    [&]( int i ){ a[i] += 1; }
);

// a == {11, 21, 94, 13}
```

```
// Deuxieme forme: avec blocked_range.
int a[4] = {10, 20, 93, 12};
parallel_for(
  blocked_range<size_t>(0, 4),
  [&]( blocked_range<size_t> r ){
       for( int i = r.begin(); i < r.end(); i++ ){
         a[i] += 1;
       }
  }
);

// a == {11, 21, 94, 13}
```

13.3.3 La notion de `blocked_range`

Le type `std::size_t` :

- *unsigned integer type*
- *can store the maximum size of a theoretically possible object of any type*
- *commonly used for array indexing and loop counting*

Source : `http://en.cppreference.com/w/cpp/types/size_t`

Un `blocked_range` représente un intervalle d'indices adjacents, qui exclut la borne supérieure :

- `blocked_range<size_t>(a, b)`
  $= [\mathtt{a}, \mathtt{b})$
  $= \mathtt{a}, \mathtt{a}+1, \mathtt{a}+2, \ldots, \mathtt{b}-1$

- Exemple :

```
r = blocked_range<size_t>(0, N)
r.begin() == 0
r.end() == N
```

- Donc :
  - C++ : `blocked_range<size_t>(a, b)`
  - Ruby : `a...b`

Le programme TBB 13.4 présente deux versions séquentielles d'une procédure pour la somme de deux tableaux :

- `additionner_seq` : Style standard où les éléments à traiter sont spécifiés par l'intermédiaire de bornes explicites — bornes inférieure *inclusive* (`debut`) et borne supérieure *exclusive* (`fin`).

- `additionner_seq_range` : Style où les éléments à traiter sont spécifiés par un `blocked_range` :
  - `blocked_range<size_t>(0, n)` représente l'intervalle $[0, n)$ — donc $0, 1, 2, \ldots, n-1$ (borne supérieure **exclusive**).

Programme TBB **13.4** Deux versions séquentielles d'une procédure effectuant la somme de deux tableaux.

```
// Premiere version sequentielle: style "standard", i.e., av
void additionner_seq( const float a[], const float b[],
                      float c[],
                      const size_t begin, const size_t end )
{
  for ( size_t i = begin; i < end; i++ ) {
    c[i] = a[i] +  b[i];
  }
}



// Deuxieme version sequentielle: avec blocked_range.
void additionner_seq_range( const float a[], const float b[]
                            float c[],
                            const blocked_range<size_t> r )
{
  for ( size_t i = r.begin(); i < r.end(); i++ ) {
    c[i] = a[i] + b[i];
  }
}



// Appels, avec les declarations "float a[n], b[n] et c[n]".

// Appel de la premiere version.
additionner_seq(a, b, c, 0, n);

// Appel de la deuxieme version.
additionner_seq_range(a, b, c, blocked_range<size_t>(0, n));
```

Programme Ruby 13.1 Méthodes Ruby semblables à celle en TBB, où on suppose que `a`, `b` et `c` ont `n` éléments.

```
def additionner_seq( a, b, c, begin_, end_ )
  (begin_...end_).each do |i|
    c[i] = a[i] + b[i]
  end
end

def additionner_seq_range( a, b, c, r )
  r.each do |i|
    c[i] = a[i] + b[i]
  end
end

additionner_seq( a, b, c, 0, n )

additionner_seq_range( a, b, c, 0...n )
```

Programme TBB 13.5 Spécification du type `blocked_range`.

```
// tbb::blocked_range< Value > Class Template Reference

// Public Member Functions

blocked_range(Value begin_,
              Value end_,
              size_type grainsize_=1)
// Construct range over half-open interval [begin,end),
// with the given grainsize.

Value begin() const
// Beginning of range.
Value end() const
// One past last value in range.

size_t size() const
// Size of the range.
size_t grainsize() const
// The grain size for this range.

bool empty() const
// True if range is empty.
bool is_divisible() const
// True if range is divisible.

blocked_range(blocked_range &r, split)
// Split range.
```

13.3.4 Exemple : Somme de deux tableaux avec parallel_for

Programme TBB 13.6 Une version parallèle d'une procédure effectuant la somme de deux tableaux.

```
// Version parallele avec lambda-expression et cod
void additionner_par(
        const float a[],
        const float b[],
        float c[],
        size_t n )
{
  parallel_for( blocked_range<size_t>(0, n),
    [=]( blocked_range<size_t> r ) {
        for ( size_t i = r.begin(); i < r.end(); i++ ) {
          c[i] = a[i] + b[i];
        }
    }
 );
}

// Appel, avec declarations float a[n], b[n] et c[
additionner_par( a, b, c, n );
```

13.4 Réduction : `parallel_reduce`

13.4.1 La spécification du `parallel_reduce`

La forme la plus simple est la forme **fonctionnelle** :

```
template<typename Range,
         typename Value,
         typename Func,
         typename Reduction>
Value parallel_reduce( const Range& range,
                       const Value& identity,
                       const Func& func,
                       const Reduction& reduction );
```

- `Range& range` :

  Intervalle à traiter

- `Value identity` :

  Élément neutre.

- `Func::operator()( const Range& range, const Value& in )` :

  Calcule le résultat pour l'intervalle `range` en utilisant `in` comme valeur initiale.

- `Reduction::operator()( const Value& x, const Value& y )` :

  Combine les résultats `x` et `y`.

  Utilisée pour **combiner** les résultats produits par le traitement des différents `Range`s.

Note : Si on compare avec `preduce` de PRuby :

- `Func` = bloc passé à `preduce`
- `Reduction` = lambda-expression passée via l'argument `final_reduce`:

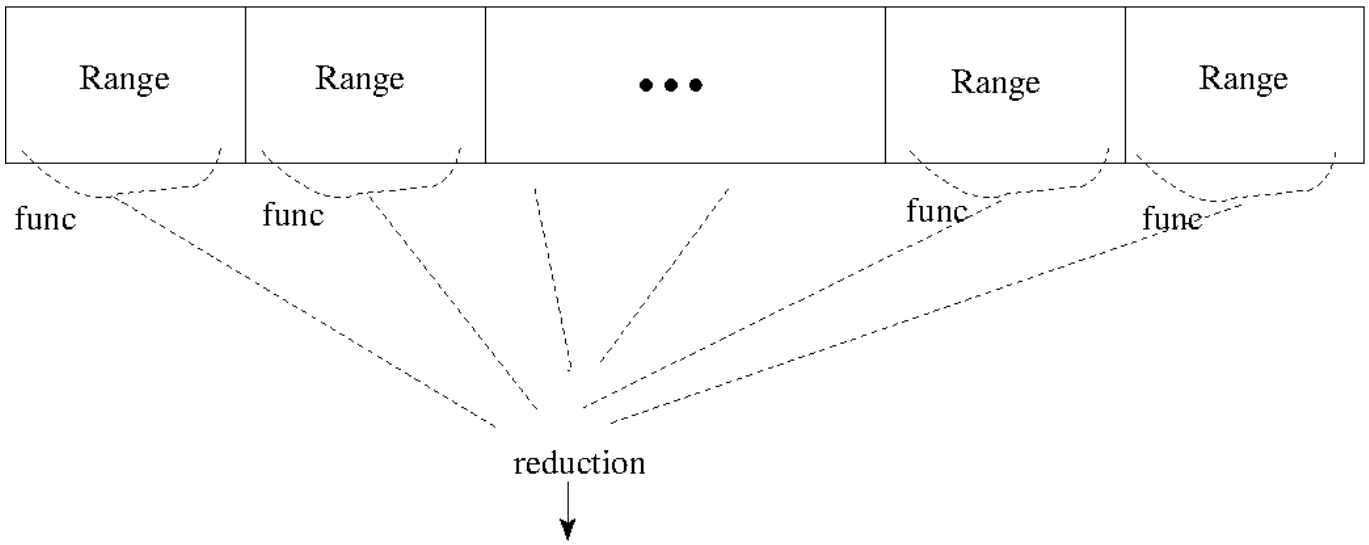

Figure 13.3: Fonctionnement d'une réduction avec `parallel_reduce` et les deux fonctions passées en argument.

13.4.2 Exemple : Sommation des éléments d'un tableau avec `parallel_reduce`

Le programme TBB 13.7 présente la fonction `sommePar` qui calcule la somme des éléments d'un tableau `a`.

Programme TBB 13.7 Fonction `sommePar` pour calculer la somme des éléments d'un tableau avec un `parallel_for` — version incorrecte!.

```
// Version INCORRECTE avec parallel_for.
float sommeParINCORRECTE( float a[], size_t n )
{
  float somme = 0.0;

  parallel_for(
      blocked_range<size_t>(0,n),
      [&]( blocked_range<size_t> r ) {
          float sommeLocal = 0.0;
          for ( size_t i = r.begin(); i < r.end(); i++ ) {
             sommeLocal += a[i];
          }
          somme += sommeLocal;
      }
  );

  return somme;
}
```

Programme TBB **13.7** Fonction `sommePar` pour calculer la somme des éléments d'un tableau avec un `parallel_for` — version avec un verrou.

```
// Version correcte avec parallel_for et verrou.
float sommeParVerrou( float a[], size_t n )
{
  float somme = 0.0;
  tbb::mutex mutex;
  parallel_for(
      blocked_range<size_t>(0,n),
      [&]( blocked_range<size_t> r ) {
           float sommeLocal = 0.0;
           for ( size_t i = r.begin(); i < r.end(); i++ ) {
              sommeLocal += a[i];
           }
           mutex.lock();
           somme += sommeLocal;
           mutex.unlock();
      }
  );

  return somme;
}
```

Programme TBB 13.7 Fonction `sommePar` pour calculer la somme des éléments d'un tableau avec un `parallel_for` — avec verrou automatiquement libéré.

```
// Version correcte avec parallel_for et verrou automatiquem
float sommeParVerrouBis( float a[], size_t n )
{
  float somme = 0.0;
  tbb::mutex mutex;
  parallel_for(
      blocked_range<size_t>(0,n),
      [&]( blocked_range<size_t> r ) {
          float sommeLocal = 0.0;
          for ( size_t i = r.begin(); i < r.end(); i++ ) {
             sommeLocal += a[i];
          }
          {
             tbb::mutex::scoped_lock lock(mutex);
             somme += sommeLocal;
          }
      }
  );

  return somme;
}
```

Programme TBB **13.7** Fonction `sommePar` pour calculer la somme des éléments d'un tableau, avec `parallel_reduce`.

```
// Version correcte (et + efficace!) avec parallel_reduce.
float sommePar( float a[], size_t n )
{
  return parallel_reduce(
      blocked_range<size_t>(0,n),
      0.f,
      [=]( blocked_range<size_t> r, float acc ) {
           for ( size_t i = r.begin(); i < r.end(); i++ ) {
             acc += a[i];
           }
           return acc;
         },
      std::plus<float>()
   );
}
```

13.4.3 Exemple : Calcul de π avec `parallel_reduce`

Le programme TBB 13.8 présente la fonction `calculDePi` qui produit une approximation de la valeur de π par intégration numérique.

Programme TBB 13.8 Fonction `calculDePi` pour calculer la valeur de π.

```
#include "tbb/parallel_reduce.h"
using namespace tbb;

double calculDePi( const blocked_range<size_t> r,
                   const double largeur );

int main( int argc, char* argv[] )
{
  int nb_rectangles = atoi(argv[1]);
  double largeur = 1.0 / (double) nb_rectangles;

  pi = calculDePi( blocked_range<size_t>(0,nb_rectangles),
                   largeur );
  printf( "La valeur de pi est %15.12f\n", pi );

  return 0;
}
```

```
double calculDePi( const blocked_range<size_t> r,
                   const double largeur )
{
  double somme = parallel_reduce(
     r,
     0.d,
     [=]( blocked_range<size_t> r, double somme ) {
          for ( size_t i = r.begin(); i < r.end(); ++i ) {
            double x = (i + 0.5) * largeur;
            somme += 4.0 / (1.0 + x*x);
          }
          return somme;
        },
     std::plus<double>()
     );

  return somme * largeur;
}
```

13.4.4 Exercices : Génération d'un histogramme

Le programme C 13.1 présente une fonction séquentielle pour produire un histogramme.

```
elems == { 10, 1, 3, 3, 3, 2, 9, 1, 1, 1, 3, 10 }

histo = histogramme( elems, 12, 10 )
```

Voici la valeur de `histo` après l'appel :

```
histo == { 0, 4, 1, 4, 0, 0, 0, 0, 0, 1, 2 }
```

Question : Peut-on «facilement» paralléliser cette solution séquentielle (Programme C 13.1)? Si oui de quelle façon? Sinon comment faire?

Programme C 13.1 Fonction séquentielle, en C, pour le calcul d'un histogramme.

```
/*
    Fonction pour la generation d'un histogramme.

    Donnees d'entree:
    - elems: Tableau d'entiers non-negatifs
             (0 ≤ elems[i] ≤ valMax)

    - n: Taille du tableau elems

    - valMax = Valeur maximum parmi les elements
               d'elems

    Resultat:
    - Pointeur vers le tableau (dynamique) de
      l'histogramme resultant
      (alloue dynamiquement par la fonction)
*/
```

```
int* histogramme( int elems[], int n, int valMax )
{
  // On alloue le tableau pour l'histogramme resul
  int* histo
    = (int*) malloc( (valMax+1) * sizeof(int) );

  // On initialise l'histogramme.
  for( int i = 0; i <= valMax; i++ ) {
    histo[i] = 0;
  }

  // On construit l'histogramme en analysant les d
  for( int i = 0; i < n; i++ ) {
    histo[elems[i]] += 1;
  }

  return histo;
}
```

Programme C 13.2 Autre version d'une fonction séquentielle, en C, pour le calcul d'un histogramme.

```c
int nb_occurrences(int val, int elems[], int nb)
{
  int nb_occ = 0;
  for( int k = 0; k < nb; k++ ) {
    if ( elems[k] == val ) { nb_occ += 1; }
  }

  return nb_occ;
}



int* histogramme(int elems[], int nb, int valMax)
{
  // On alloue et on initialise l'histogramme.
  int* histo
    = (int*) malloc( (valMax+1) * sizeof(int) );

  for( int val = 0; val <= valMax; val++ ) {
    histo[val] = nb_occurrences(val, elems, nb);
  }

  return histo;
}
```

On veut paralléliser la fonction `histogramme` présentée plus haut (Programme C 13.2), fonction qui produit un histogramme pour les entiers du tableau `elems`.

En utilisant les *Threading Building Blocks* d'Intel, écrivez une version parallèle de la fonction `histogramme`. Utilisez une approche de **parallélisme de résultat** — même si cela implique de parcourir plusieurs fois la matrice `elems`.

- Parallélisez tout d'abord `histogramme`. Ensuite, parallélisez `nb_occurrences`!

Exercice 13.4: Problème de l'histogramme.

Que fait la fonction suivante?

```
int* mystere( int elems[], int nb, int vm )
{
  auto init = [vm]() {
    int* h0 = (int*) malloc( (vm+1) * sizeof(int) );
    for( int k = 0; k <= vm; k++ ) {
      h0[k] = 0;
    }
    return h0;
  };

  auto foo = [=]( blocked_range<size_t> r,
                  int* h ) {
    for( auto i = r.begin(); i < r.end(); i++ ) {
      h[elems[i]] += 1;
    }
    return h;
  };

  auto bar = [vm]( int* h1, int* h2 ) {
    for( int k = 0; k <= vm; k++ ) {
      h1[k] += h2[k];
    }
    return h1;
  };

  return
    parallel_reduce( blocked_range<size_t>(0, nb),
                     init(),
                     foo,
                     bar
                     );
}
```

Exercice 13.5: Programme mystère.

13.5 Fonctionnement du `parallel_for` avec `blocked_range` et rôle du `partitioner`

Note : Les explications qui suivent s'appliquent aussi au `parallel_reduce`.

Rappel : Spécification du type `blocked_range`.

```
// tbb::blocked_range< Value > Class Template Reference

// Public Member Functions

blocked_range(Value begin_,
              Value end_,
              size_type grainsize_=1)
// Construct range over half-open interval [begin,end),
// with the given grainsize.

Value begin() const
// Beginning of range.
Value end() const
// One past last value in range.

size_t size() const
// Size of the range.
size_t grainsize() const
// The grain size for this range.

bool empty() const
// True if range is empty.
bool is_divisible() const
// True if range is divisible.

blocked_range(blocked_range &r, split)
// Split range.
```

Programme TBB 13.9 Mise en oeuvre du type `blocked_range`.

```
template <typename T>
class blocked_range {
public:
  blocked_range( T begin, T end, size_t grainsize ) :
    my_begin(begin), my_end(end), my_grainsize(grainsize) {}

  blocked_range( T begin, T end ) :
    my_begin(begin), my_end(end), my_grainsize(1) {}

  T begin() const { return my_begin; }

  T end() const { return my_end; }

  size_t size() const { return size_t( my_end - my_begin ); }

  size_t grainsize() const { return my_grainsize; }

  // Methodes pour Range.
  bool empty() const { return !( my_begin < my_end ); }

  bool is_divisible() const { return my_grainsize < size(); }

  blocked_range( blocked_range& r, split )
  {
    T middle = r.my_begin + (r.my_end - r.my_begin) / 2u;

    // Nouveau blocked_range = intervalle superieur.
    my_begin = middle;
    my_end = r.my_end;

    // blocked_range initial r = intervalle inferieur.
    // r.my_begin est inchange.
    r.my_end = middle;

    // Les deux intervalles ont la meme granularite.
    my_grainsize = r.my_grainsize;
  }

private:
  T my_begin;
  T my_end;
  size_t my_grainsize;
};
```

```
// Split constructor
//    => definit un nouvel objet blocked_range
//
// r = range existant qu'on veut "splitter".
//
blocked_range( blocked_range& r, split )
{
  T middle
    = r.my_begin + (r.my_end - r.my_begin) / 2u;

  // Nouveau blocked_range = intervalle droit.
  my_begin = middle;
  my_end = r.my_end;

  // blocked_range initial r = intervalle gauche.
  r.my_begin = r.my_begin; // i.e., inchange!
  r.my_end = middle;

  // Les deux intervalles ont la meme granularite.
  my_grainsize = r.my_grainsize;
}
```

```
void parallel_for( Range range, Body body )
DEBUT
  SI  !range.is_divisible() ALORS
    // Cas de base => traitement séquentiel via le body.
    body( range )
  SINON
     // On décompose le range en deux sous-ranges
     //    avec le constructeur split
     //    (diviser-pour-régner dichotomique).
    autre_range ← Range( range, split )

     // On crée deux nouvelles tâches.
     SPAWN parallel_for( autre_range, body )
     SPAWN parallel_for( range, body )
  FIN
FIN
```

Algorithme 13.4: Pseudocode (version simplifiée!) pour `parallel_for` utilisant les méthodes d'un objet `Range`... en ne tenant pas compte du `grainsize` et du `partitioner()`.

Pour comprendre le rôle du `grainsize`, il faut savoir que la signature de `parallel_for` est la suivante :

```
template<typename Range, typename Body>
void parallel_for(
      const Range& range,
      const Body& body
      [, partitioner
      [, task_group_context& group]] );
```

Rôle d'un `partitioner` = déterminer **jusqu'à quel point les intervalles doivent être décomposés**.

Les deux `partitioners` de base :

- `simple_partitioner()`
- `auto_partitioner()` (défaut!)

Les deux appels suivants sont équivalents :

```
parallel_for( r, lambdaExpr )
parallel_for( r, lambdaExpr, auto_partitioner() )
```

Les deux appels suivants sont équivalents :

```
blocked_range<size_t>( i, j )
blocked_range<size_t>( i, j, 1 )
```

Programme TBB **13.10** Des appels à une procédure `additionner` par l'intermédiaire d'un `parallel_for` utilisant un `simple_partitioner()`. La procédure appelée affiche une trace d'exécution indiquant les bornes de l'intervalle à traiter.

```
void additionner( const float a[], const float b[],
                  float c[],
                  blocked_range<size_t> r )
{
  printf( "additionner( a, b, c, [%d, %d) )\n",
          r.begin(), r.end() );

  for ( size_t i = r.begin(); i < r.end(); i++ ) {
    c[i] = a[i] + b[i];
  }
}
...

const int N = 1000;
int a[N], b[N], c[N];
...
int grainsize = ...; // 200, 100, 1.
parallel_for(
   blocked_range<size_t>(0, N, grainsize),
   [=]( blocked_range<size_t> r ) {
        additionner( a, b, c, r );
   },
   simple_partitioner()
);
```

```
// int grainsize = 200;
additionner( a, b, c, [0, 125) )
additionner( a, b, c, [500, 625) )
additionner( a, b, c, [250, 375) )
additionner( a, b, c, [375, 500) )
additionner( a, b, c, [125, 250) )
additionner( a, b, c, [750, 875) )
additionner( a, b, c, [625, 750) )
additionner( a, b, c, [875, 1000) )

// int grainsize = 100;
additionner( a, b, c, [500, 562) )
additionner( a, b, c, [750, 812) )
additionner( a, b, c, [875, 937) )
additionner( a, b, c, [937, 1000) )
additionner( a, b, c, [812, 875) )
additionner( a, b, c, [562, 625) )
additionner( a, b, c, [625, 687) )
additionner( a, b, c, [687, 750) )
additionner( a, b, c, [0, 62) )
additionner( a, b, c, [250, 312) )
additionner( a, b, c, [125, 187) )
additionner( a, b, c, [375, 437) )
additionner( a, b, c, [62, 125) )
additionner( a, b, c, [312, 375) )
additionner( a, b, c, [187, 250) )
additionner( a, b, c, [437, 500) )

// int grainsize = 1;
additionner( a, b, c, [500, 501) )
additionner( a, b, c, [750, 751) )
additionner( a, b, c, [875, 876) )
additionner( a, b, c, [812, 813) )
.
.
.
additionner( a, b, c, [59, 60) )
additionner( a, b, c, [60, 61) )
additionner( a, b, c, [626, 627) )
...
additionner( a, b, c, [484, 485) )
additionner( a, b, c, [371, 372) )
...
additionner( a, b, c, [498, 499) )
additionner( a, b, c, [499, 500) )
additionner( a, b, c, [467, 468) )
```

Nombre de tâches pour $n = 10000$ et `grainsize = 1` :

	Partitionneur
No. exécution	`simple_partitioner()`
1	10 000
2	10 000
3	10 000
...	10 000

Nombre de tâches pour $n = 10000$ et `grainsize = 1` :

	Partitionneur
No. exécution	`simple_partitioner()`
1	10 000
2	10 000
3	10 000
...	10 000

Donc : avec le `simple_partitioner()`, la décomposition se fait toujours jusqu'à ce que la taille du sous-problème soit $\leq$ **`grainsize`**!

Nombre de tâches pour $n = 10000$ et `grainsize = 1` :

	Partitionneur	
No. exécution	`simple_partitioner()`	`auto_partitioner()`
1	10 000	988
2	10 000	1 033
3	10 000	1 020
...	10 000	...

Nombre de tâches pour $n = 10000$ et `grainsize = 1` :

	Partitionneur	
No. exécution	`simple_partitioner()`	`auto_partitioner()`
1	10 000	988
2	10 000	1 033
3	10 000	1 020
...	10 000	...

L'`auto_partitioner()` cesse la décomposition quand les *threads* sont suffisamment «occupés».

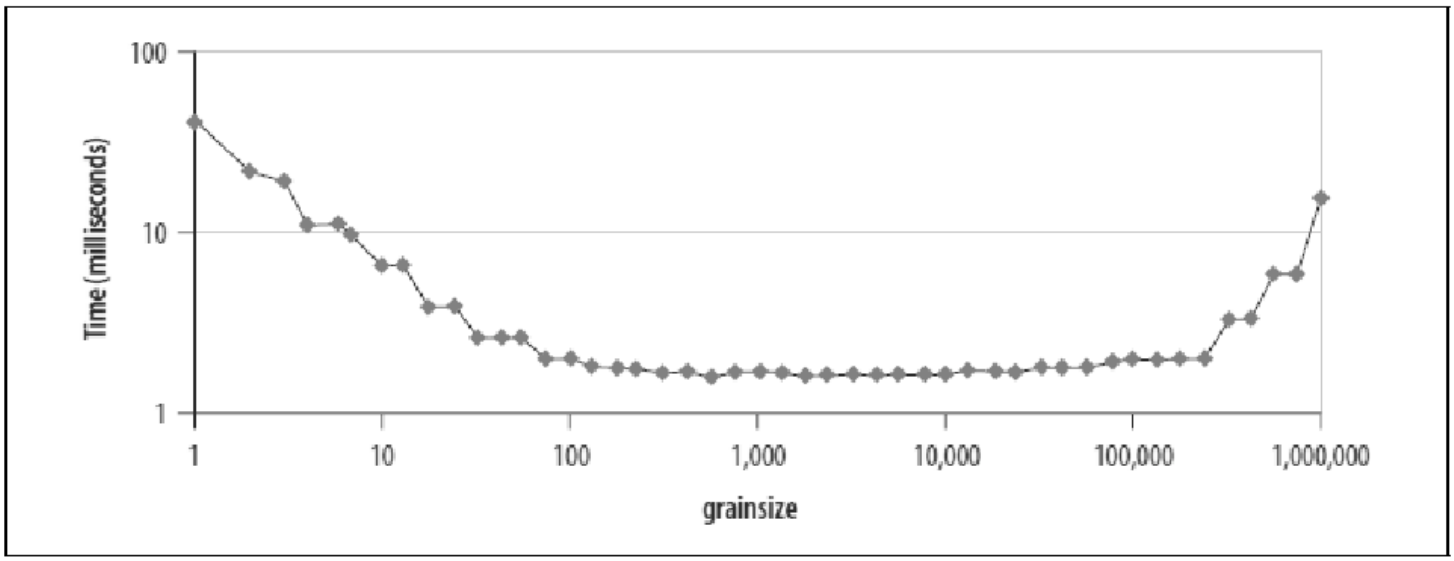

Figure 3-2. Wall clock time versus grainsize

Figure 13.4: Courbe (avec échelle logarithmique) illustrant l'effet typique de la taille des grains sur le temps d'exécution d'un programme `TBB` lorsqu'un `simple_partitioner()` est utilisé.

13.6 Parallélisme de contrôle : `parallel_invoke` et `task_group`

```
// Deux fonctions en parallele.
template<typename Func0,
         typename Func1>
  void parallel_invoke(const Func0& f0,
                       const Func1& f1);

// Trois fonctions en parallele.
template<typename Func0,
         typename Func1,
         typename Func2>
  void parallel_invoke(const Func0& f0,
                       const Func1& f1,
                       const Func2& f2);

.
.
.

// Dix fonctions en parallele.
template<typename Func0,
         typename Func1,
         ...,
         typename Func9>
  void parallel_invoke(const Func0& f0,
                       const Func1& f1,
                       ...,
                       const Func9& f9);
```

Programme TBB **13.11** Fonction pour calculer le $n^{ième}$ nombre de Fibonacci à l'aide de parallélisme récursif réalisé avec `parallel_invoke`.

```
#include "tbb/parallel_invoke.h"
using namespace tbb;

int fibo_par( int n )
{
  if( n == 1 || n == 2 ) {
    return 1;
  } else {
    int r1, r2;
    parallel_invoke( [&] { r1 = fibo_par(n-1); },
                     [&] { r2 = fibo_par(n-2); } );
    return r1 + r2;
  }
}
```

On peut aussi utiliser des groupes de tâches, tel que présenté dans le programme TBB 13.12.

Les opérations associées sont les suivantes :

```
// Spawns a task to compute f() and returns immediately.
template<typename Func>
  void run( const Func& f );

template<typename Func>
  void run( task_handle<Func>& handle );

template<typename Func>
  void run_and_wait( const Func& f );

template<typename Func>
  void run_and_wait( task_handle<Func>& handle );

// Waits for all tasks in the group to complete or be cancelled.
// Returns: True if this task group is cancelling its tasks.
task_group_status wait();

// Returns: True if this task group is cancelling its tasks.
bool is_canceling();

// Cancels all tasks in this task group.
void cancel();
```

Programme TBB **13.12** Fonction pour calculer le $n^{ième}$ nombre de Fibonacci à l'aide de parallélisme récursif réalisé avec un `task_group`.

```
#include "tbb/task_group.h"
using namespace tbb;

int fibo_par( int n )
{
  if( n == 1 || n == 2 ) {
    return 1;
  } else {
    int r1, r2;
    task_group g;
    g.run( [&]{ r1 = fibo_par(n-1); } );
    g.run( [&]{ r2 = fibo_par(n-2); } );
    g.wait();
    return r1 + r2;
  }
}
```

13.7 Tri parallèle : parallel_sort

```
template<typename RandomAccessIterator>
  void parallel_sort( RandomAccessIterator begin,
                      RandomAccessIterator end );

template<typename RandomAccessIterator, typename Compare>
  void parallel_sort( RandomAccessIterator begin,
                      RandomAccessIterator end,
                      const Compare& comp );
```

Les propriétés requises d'un objet RandomAccessIterator sont les suivantes :

```
void swap( T& x, T& y )
// Interchange les deux elements.

bool Compare::operator()( const T& x, const T& y )
// Vrai si x vient avant y dans la sequence triee.
```

Programme TBB **13.13** Un exemple de programme de tri.

```
#include "tbb/parallel_sort.h"

using namespace tbb;

void initialiser( int a[], int N ) { ... }

const int N = 100;

int main( int argc, char *argv[] ) {

  int a[N] = ...;

  ...

  // Tri en ordre croissant.
  parallel_sort( a, a + N );
  ...

  // Tri en ordre decroissant.
  parallel_sort( a, a + N, std::greater<int>() );
  ...
}
```

13.8 Parallélisme de flux : `pipeline`

Soit un polynôme p:

$$p(x) = a + bx + cx^2 + dx^3$$

Ce polynome peut être évalué par la méthode de Horner :

$$p(x) = (((((d * x) + c) * x) + b) * x) + a$$

Soit alors les fonctions suivantes :

$$\begin{aligned} f_0\ (x,\ y) &= (x, y * x + d) \\ f_1\ (x,\ y) &= (x, y * x + c) \\ f_2\ (x,\ y) &= (x, y * x + b) \\ f_3\ (x,\ y) &= (x, y * x + a) \end{aligned}$$

L'évaluation du polynôme $p(x)$ peut aussi être exprimée comme suit :

$$\begin{aligned} p(x) &= r_2 \\ &\quad \texttt{where}\ (r_1, r_2) = f_3\ (f_2\ (f_1\ (f_0\ (x, 0)))) \end{aligned}$$

$$
\begin{aligned}
p(x) &= r_2 \\
&\quad \texttt{where } (r_1, r_2) = f_3\ (f_2\ (f_1\ (f_0\ (x, 0))))
\end{aligned}
$$

$$
\begin{aligned}
(r_1, r_2) &= f_3\ (f_2\ (f_1\ (f_0\ (x, 0)))) \\
&= f_3\ (f_2\ (f_1\ (x, d))) \\
&= f_3\ (f_2\ (x, d * x + c)) \\
&= f_3\ (x, (d * x + c) * x + b) \\
&= (x, ((d * x + c) * x + b) * x + a) \\
&= (x, (d * x + c) * x^2 + b * x + a) \\
&= (x, d * x^3 + c * x^2 + b * x + a)
\end{aligned}
$$

Donc :

$$p(x) = r_2 = dx^3 + cx^2 + bx + a$$

Programme TBB **13.14** Classe `Paire` pour représenter des tuples formés de deux composants.

```
class Paire {
public:
  double x;
  double y;
public:
  Paire( double x, double y ) :
    x(x), y(y) {}
};
```

Programme TBB **13.15** Trois sortes de `filter` pour manipuler des valeurs et des coefficients pour évaluer un polynome.

```
class GenererVals : public filter {
  int prochain = 0;
  int nbVals;
  double *vals;
public:
  GenererVals( double *vals, int nbVals ) :
      filter(serial_in_order), nbVals(nbVals), vals(vals) {}

  void* operator()( void* ) {
    if (prochain < nbVals) {
      return new Paire( vals[prochain++], 0.d );
    } else {
      return NULL;
    }
  }
};

class EvaluerCoeff : public filter {
  double coeff;
public:
  EvaluerCoeff( double coeff ) :
      filter(serial_in_order), coeff(coeff) {}

  void* operator()( void* v ) {
    Paire* p = (Paire *) v;
    p->y = p->x * p->y + coeff;
    return p;
  }
};

class AccumulerResultats : public filter {
  double *resultats;
  int suivant = 0;
public:
  AccumulerResultats( double *resultats ) :
      filter(serial_in_order), resultats(resultats) {}

  void* operator()( void* v ) {
    Paire* p = (Paire *) v;
    resultats[suivant++] = p->y;
    free( p );
  }
};
```

Programme TBB 13.16 Programme TBB pour évaluer un polynome en une série de points, à l'aide d'une approche fondée sur le parallélisme de flux, et utilisant les filtres du programme TBB 13.15.

```
//
// Exemple d'utilisation.
//

//
// On suppose qu'on a les trois tableaux suivants:
//   coeffs:    les nbCoeffs coefficients definissant le pol
//   vals:      les nbVals valeurs a utiliser pour evaluer l
//   resultats: les nbVals resultats
//
pipeline p;

// Le producteur (la source) qui genere les differentes vale
GenererVals generateurVals(vals, nbVals);
p.add_filter( generateurVals );

// Les differents filtres pour chacun des coefficients.
for( int i = 0; i < nbCoeffs; i++ ) {
  p.add_filter( *(new EvaluerCoeff(coeffs[i])) );
}

// Le consommateur (sink, puits) qui accumule les divers res
AccumulerResultats accumulerResultats(resultats);
p.add_filter( accumulerResultats );

p.run( nbCoeffs );

// Les resultats sont maintenant disponible dans le tableau
```

13.9 Mesures de performance

13.9.1 Mesures du temps d'exécution

Programme TBB **13.17** Un petit exemple illustrant la mesure du temps d'exécution d'un segment de code.

```
#include "tbb/tick_count.h"
using namespace tbb;

void foo()
{
    tick_count t0 = tick_count::now();
    ... action being timed ...
    tick_count t1 = tick_count::now();

    printf( "time for action = %g seconds\n",
            (t1-t0).seconds() );
}
```

Source : http://www.threadingbuildingblocks.org/docs/help/reference/timing/tick_count_cls.htm

13.9.2 Dimensionabilité

Règle générale, dans un programme TBB, on laisse le soin au système d'exécution de choisir le nombre de *threads* ; **on se concentre plutôt sur l'identification des *tâches***.

Par contre, il est quand même possible de contrôler le nombre de *threads* utilisés :

```
task_scheduler_init(
    int max_threads=automatic,
    stack_size_type thread_stack_size=0
)
```

Programme TBB **13.18** Squelette de programme pour déterminer l'effet du nombre de *threads* sur les performances d'un programme TBB.

```
#include "tbb/task_scheduler_init.h"

using namespace tbb;

int main( int argc, char* argv[] )
{
  int nb_threads
    = task_scheduler_init::default_num_threads();

  for( int k = 1; k <= nb_threads; k *= 2 ) {
    printf( "Execution avec %d thread(s)\n", k );

    task_scheduler_init init( k ); // Allocation statique.
    ... code dont on veut mesurer les performances ...
    .
    .
    .

    // Fin du bloc: l'objet task_scheduler_init est libere.
  }
}
```

13.10 Approche fondée sur les tâches et ordonnancement par «vol de tâches»

Another advantage of tasks versus logical threads is that **tasks are much lighter weight**.

On Linux systems, starting and terminating a task is about 18 times faster than starting and terminating a thread. On Windows systems, the ratio is more than 100-fold.

Source : «*Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*», Reinders, 2007.

[F]or scheduling loop iterations, Threading Building Blocks does not require the programmer to worry about scheduling policies. Threading Building Blocks does away with this in favor of a single, **automatic, divide-and-conquer approach to scheduling**.

Implemented with **work stealing** *(a technique for moving tasks from loaded processors to idle ones), it compares favorably to [OpenMP's] dynamic or guided scheduling, but without the problems of a centralized dealer.*

Source : «*Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*», Reinders, 2007.

Le modèle TBB est fondé sur la **décomposition en *tâches***, indépendamment des *threads*.

Le modèle d'exécution de TBB doit être vu comme une approche de style «sac de tâches», donc avec **association dynamique entre tâches et *threads***

L'approche utilisée pour l'affectation des tâches est le **vol de tâches** (*work stealing*).

Traitement effectué par les *threads* :

- Chaque *thread* utilise un *deque* :

`empty?`	Vide?
`push`	Ajoute en tête
`pop`	Retire de la tête
`remove`	Retire de la queue

- Soit D le *deque* associé à un *thread* T.
  - T **crée une nouvelle tâche** :
    l'ajout se fait avec D`.push`.
  - T **termine une tâche**, alors pour obtenir une nouvelle tâche :
    * `!`D`.empty?` $\Rightarrow$
      tâche locale, obtenue avec D`.pop`.
    * D`.empty?` $\Rightarrow$
      T va *voler une tâche* à un *thread* T' **choisi aléatoirement**, obtenue avec D'`.remove`.

Fait : Lorsqu'il y a assez de tâches locales, le *deque* est **manipulé *comme une pile***.

Voir les diapositives suivantes :

`http://www.labunix.uqam.ca/~tremblay/INF7235/Materiel/vol-de-taches.pdf`

- Les avantages de l'approche avec *work stealing* :
  - Le coût pour traiter une tâche locale est faible

    Le coût n'est élevé que lorsque cette tâche «migre» suite à un vol

  - L'arbre des tâches est exploré localement **en profondeur**, ce qui minimise l'espace requis et utilise efficacement les caches

    L'exploration **en largeur** ne se fait que si des *threads* sont libres et n'ont aucun travail à effectuer. Ceci favorise **l'exploration en parallèle**

To summarize, the task scheduler's fundamental strategy is **breadth-first theft** *and* **depth-first work**.

The breadth-first theft rule **raises parallelism** *sufficiently to keep threads busy.*

The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do.

Source : «*Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*», Reinders, 2007.

13.11 L'approche «Diviser-pour-régner» et la notion de `Range`

En TBB, la décomposition en sous-problème avec les `Range` peut être beaucoup plus complexe qu'une simple modification des bornes!

Dans ce qui suit, deux versions du tri rapide :

1. Version «ordinaire»
2. Version où tout travail de décomposition se fait lors de la création des `Range`s.

Programme TBB **13.19** Une mise en oeuvre parallèle du tri rapide (*quicksort*) avec une approche style *parallélisme récursif*, et avec un seuil de récursion.

```
void partitionner(int* a, size_t taille, size_t& posPivot)
{
  // Pour simplifier: On selectionne le premier element comm
  auto pivot = a[0];
  posPivot = 0;

  // On trouve la position finale ou devra aller le pivot,
  // en deplacant les elements lorsque requis.
  for( size_t i = 1; i < taille; i++ ) {
    if ( a[i] < pivot ) {
      posPivot += 1;
      swap( a[i], a[posPivot] );
    }
  }

  // On met le pivot a sa bonne position.
  swap( a[posPivot], a[0] );
  // ASSERT: a[0:posPivot] <= pivot < a[posPivot+1:taille-1]
};
```

```
void parallel_qs_rec(int* a, size_t n, size_t seuil)
{
  if ( n <= seuil ) {
    // Cas de base.
    std::sort( a, a+n, std::less<int>() );
  } else {
    // Cas recursif.
    size_t posPivot;

    partitionner( a, n, posPivot );

    tbb::parallel_invoke(
     [=]{ parallel_qs_rec(a, posPivot, seuil); },
     [=]{ parallel_qs_rec(a+posPivot+1, n-posPivot-1, seuil)
    );
  }
};
```

Programme TBB **13.20** Une mise en oeuvre du tri rapide (*quicksort*) avec un objet de type `Range`, qui effectue le partitionnement et qui fait tout le travail, utilisé dans un `parallel_for`.

```
struct qs_range {
  int* a;
  size_t taille, seuil;

  qs_range( int* a, size_t taille, size_t seuil )
    : a(a), taille(taille), seuil(seuil) {}

  bool empty() const { return taille == 0; }

  bool is_divisible() const { return taille > seuil; }

  qs_range( qs_range& range, split ) {
    size_t posPivot;
    partitionner( range.a, range.taille, posPivot );

    // Le nouveau range traitera la partie droite.
    a = range.a + posPivot + 1;
    taille = range.taille - posPivot - 1;

    // Le range qui vient d'etre "splitte" traitera la gauch
    range.a = range.a;
    range.taille = posPivot;
    range.seuil = seuil;
  }
};
```

```
void parallel_qs(int* a, size_t n, size_t seuil)
{
 parallel_for(
   qs_range(a, n, seuil),
   []( const qs_range range ) {
     // Appel a la procedure (seq.) de la bibliotheque.
     std::sort(range.a,
               range.a+range.taille,
               std::less<int>());
   }
 );
};
```

Les figures 13.5 et 13.6 présentent des mesures sur deux variantes de ces deux procédures :

- Dans la première variante (indiqué par `p0`), on choisit toujours l'élément à la position 0 du tableau comme pivot.
- Dans la deuxième variante, le choix du pivot est effectué comme suit :
  - On examine trois éléments du tableau `a` : le premier élément, le dernier et l'élement au milieu du tableau.
  - On calcule la médiane de ces trois éléments.
  - C'est cet élément médiane qui est sélectionné comme pivot.

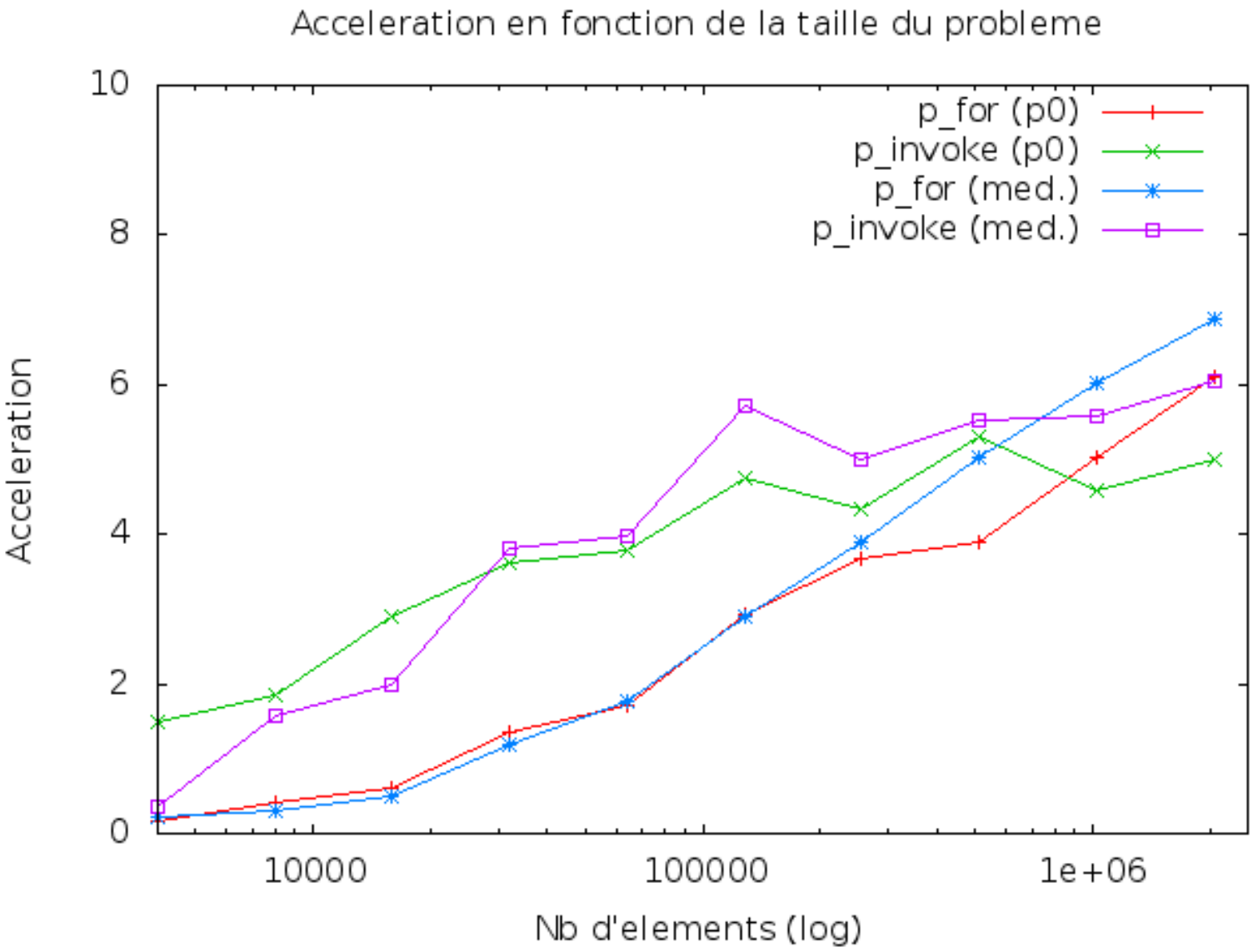

Figure 13.5: Accélérations obtenues pour différentes tailles de tableau (échelle logarithmique) et pour les deux variantes des deux procédures de tri rapide avec deux façons de choisir le pivot.

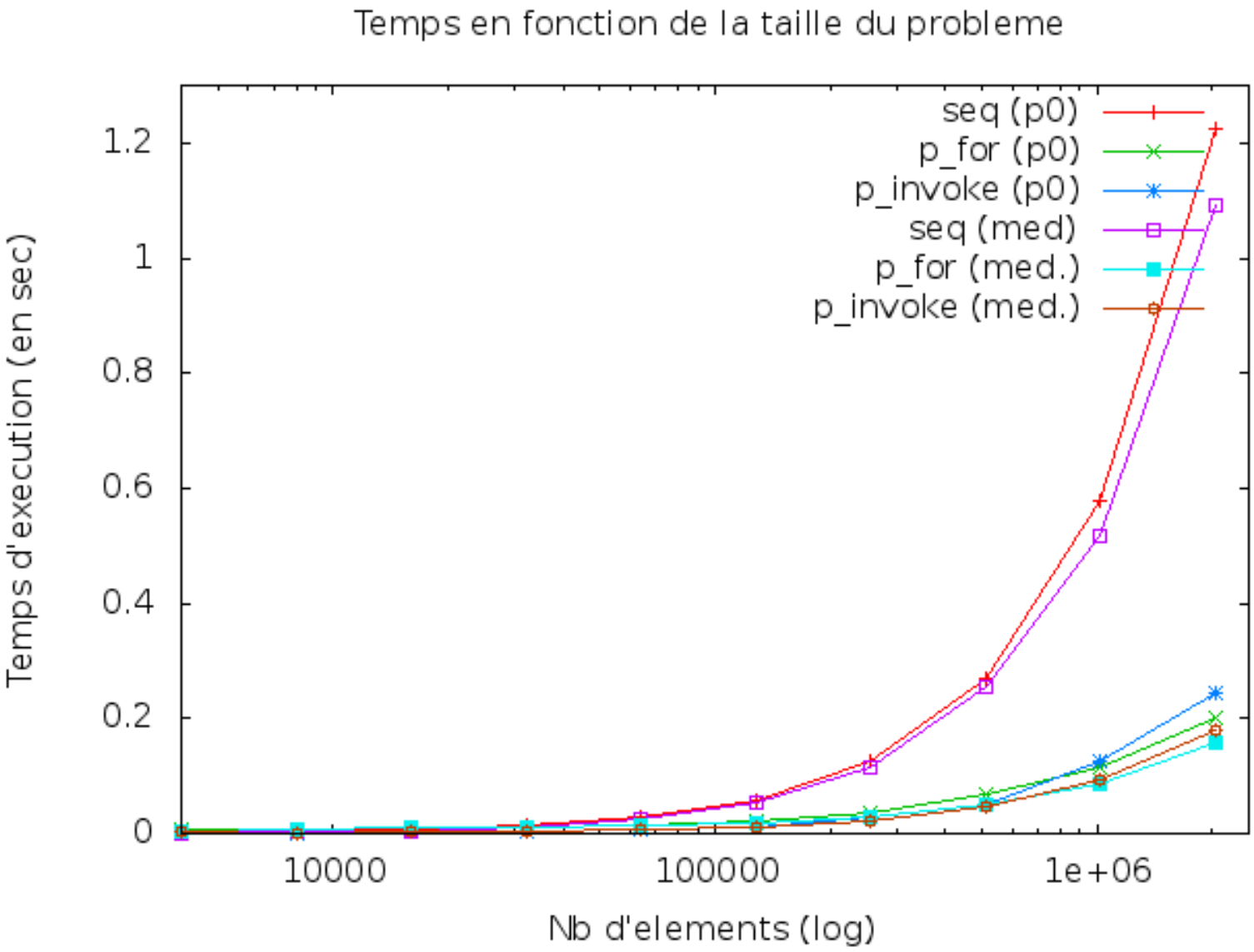

Figure 13.6: Temps d'exécution obtenus pour différentes tailles de tableau (échelle logarithmique) et pour les deux variantes des deux procédures de tri rapide avec deux façons de choisir le pivot.

Temps d'exécution sur `japet` — pour une taille donnée, le temps en **rouge** indique le meilleur temps :

```
# pivot = mediane
#      n       seq     p_for  p_invoke
    4000    0.0011    0.0051    0.0030
    8000    0.0025    0.0082    0.0016
   16000    0.0052    0.0103    0.0026
   32000    0.0118    0.0100    0.0031
   64000    0.0250    0.0141    0.0063
  128000    0.0544    0.0187    0.0095
  256000    0.1151    0.0296    0.0230
  512000    0.2558    0.0509    0.0462
 1024000    0.5174    0.0859    0.0928
 2048000    1.0928    0.1589    0.1804

# pivot = pos0
#      n       seq     p_for  p_invoke
    4000    0.0012    0.0069    0.0008
    8000    0.0026    0.0062    0.0014
   16000    0.0058    0.0094    0.0020
   32000    0.0130    0.0096    0.0036
   64000    0.0273    0.0159    0.0072
  128000    0.0584    0.0200    0.0123
  256000    0.1259    0.0342    0.0290
  512000    0.2692    0.0693    0.0508
 1024000    0.5771    0.1145    0.1256
 2048000    1.2233    0.2006    0.2444
```

Quelques constatations :

- Si on examine les accélérations, `parallel_for` et choix simple du pivot (`pivot=a[0]`) génère de bonnes accélérations.
- Si on examine les temps c'est `parallel_for` et choix de pivot qui «approxime» la médiane qui génère les meilleurs temps.

  Et `parallel_invoke` avec médiane a de meilleurs temps d'éxécution, même si son accélération n'est pas aussi bonne.

Conclusion : Les accélérations sont des caractéristiques importantes des programmes parallèles. . . mais le temps d'exécution ne doit jamais être ignoré!