

Threading Building Blocks (TBB)

Thierry Dumont

Institut Camille Jordan
UMR CNRS 5208.

24 Février 2014.

Une autre manière de concevoir le parallélisme en mémoire partagée.

- ▶ TBB est une *bibliothèque* (pas d'ajout au langage (pragma etc.)).
- ▶ Du C++ standard, rien que du C++ .
- ▶ Mais tout le C++ (templates, λ -fonctions.).
- ▶ Développement débuté en 2004.

Une autre manière de concevoir le parallélisme en mémoire partagée.

- ▶ TBB est une *bibliothèque* (pas d'ajout au langage (pragma etc.)).
- ▶ Du C++ standard, rien que du C++ .
- ▶ Mais tout le C++ (templates, λ -fonctions.).
- ▶ Développement débuté en 2004.

Développée par Intel.

Deux licences :

- ▶ GPLv2.
- ▶ Licence commerciale avec support.
(livrée avec le compilateur Intel).

Une autre manière de concevoir le parallélisme en mémoire partagée.

- ▶ TBB est une *bibliothèque* (pas d'ajout au langage (pragma etc.)).
- ▶ Du C++ standard, rien que du C++ .
- ▶ Mais tout le C++ (templates, λ -fonctions.).
- ▶ Développement débuté en 2004.

Développée par Intel.

Deux licences :

- ▶ GPLv2.
- ▶ Licence commerciale avec support.
(livrée avec le compilateur Intel).

Documentation :

- ▶ En ligne.
- ▶ Livre chez O'Reilly.

Ce que c'est, ce que ce n'est pas

- ▶ repose sur des processus légers (threads),
- ▶ parallélisme de données.
- ▶ pas de connaissances nécessaires sur les processus légers.
- ▶ on spécifie des tâches, pas des processus légers.
- ▶ parallélisme emboîté, récursivité.
- ▶ portabilité.

Ce que c'est, ce que ce n'est pas

- ▶ repose sur des processus légers (threads),
- ▶ parallélisme de données.
- ▶ pas de connaissances nécessaires sur les processus légers.
- ▶ on spécifie des tâches, pas des processus légers.
- ▶ parallélisme emboîté, récursivité.
- ▶ portabilité.

TBB / Programmation direct des fils.

- ▶ TBB Beaucoup plus léger : mécanisme des fils caché.
- ▶ Pratiquement pas de gestion de verrous avec TBB .

Ce que c'est, ce que ce n'est pas

- ▶ repose sur des processus légers (threads),
- ▶ parallélisme de données.
- ▶ pas de connaissances nécessaires sur les processus légers.
- ▶ on spécifie des tâches, pas des processus légers.
- ▶ parallélisme emboîté, récursivité.
- ▶ portabilité.

TBB / Programmation direct des fils.

- ▶ TBB Beaucoup plus léger : mécanisme des fils caché.
- ▶ Pratiquement pas de gestion de verrous avec TBB .

TBB / OMP.

- ▶ TBB : uniquement C++.
- ▶ OMP « *An excellent Fortran-style code written in C* »... même s'il y a de la gestion de tâches dans les dernières versions de OMP .
- ▶ La *récurtivité* est centrale dans TBB , alors que OMP est plutôt statique.
Récurtivité => passage à l'échelle.

TBB / OMP.

- ▶ TBB : uniquement C++.
- ▶ OMP « *An excellent Fortran-style code written in C* »... même s'il y a de la gestion de tâches dans les dernières versions de OMP .
- ▶ La *récurtivité* est centrale dans TBB , alors que OMP est plutôt statique.
Récurtivité => passage à l'échelle.

L'apprentissage de TBB est simple : on se concentre sur des concepts de haut niveau.

Penser pour TBB.

Décomposition : en tâches qui peuvent tourner en même temps.

Passage à l'échelle : le nombre de tâches doit croître quand la taille du problème augmente.

Ne pas penser aux verrous, et rarement à la synchronisation.

Bien sûr, TBB ne simplifie pas les problèmes de partage de données (« *thread safety* »).

Mais pourquoi t'intéresses tu à ça ?

Mais pourquoi t'intéresses tu à ça ?

Parce que je n'ai pas d'autre solution.

Mais pourquoi t'intéresses tu à ça ?

Parce que je n'ai pas d'autre solution. Parce que TBB résout élégamment le problème de l'équilibrage des charges.

Mais pourquoi t'intéresses tu à ça ?

Parce que je n'ai pas d'autre solution. Parce que TBB résout élégamment le problème de l'équilibrage des charges.

Un premier exemple très concret.

$$\begin{cases} \frac{\partial u_i}{\partial t}(x, t) - \varepsilon \Delta u_i(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \\ \quad \quad \quad 1 \leq i \leq m, x \in \Omega, \\ u_i(x, 0) = u_i^0(x), \quad \quad 1 \leq i \leq m, x \in \Omega. \end{cases}$$

Mais pourquoi t'intéresses tu à ça ?

Parce que je n'ai pas d'autre solution. Parce que TBB résout élégamment le problème de l'équilibrage des charges.

Un premier exemple très concret.

$$\begin{cases} \frac{\partial u_i}{\partial t}(x, t) - \varepsilon \Delta u_i(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \\ \qquad \qquad \qquad 1 \leq i \leq m, x \in \Omega, \\ u_i(x, 0) = u_i^0(x), \qquad \qquad 1 \leq i \leq m, x \in \Omega. \end{cases}$$

Système de Réaction–Diffusion (chimie, médecine...).

Mais pourquoi t'intéresses tu à ça ?

Parce que je n'ai pas d'autre solution. Parce que TBB résout élégamment le problème de l'équilibrage des charges.

Un premier exemple très concret.

$$\begin{cases} \frac{\partial u_i}{\partial t}(x, t) - \varepsilon \Delta u_i(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \\ \quad \quad \quad 1 \leq i \leq m, x \in \Omega, \\ u_i(x, 0) = u_i^0(x), \quad \quad \quad 1 \leq i \leq m, x \in \Omega. \end{cases}$$

Système de Réaction–Diffusion (chimie, médecine...).

Découper en deux blocs élémentaires :

1.

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, \dots, n.$$

2.

$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$

1. Le premier sous-problème :

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, \dots, n.$$

fait apparaître un *parallélisme du pauvre* (n tâches indépendantes) ; mais l'exécution de chaque tâche peut créer du parallélisme de tâches (exemple : produits matrice x vecteur).

1. Le premier sous-problème :

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, \dots, n.$$

fait apparaître un *parallélisme du pauvre* (n tâches indépendantes) ; mais l'exécution de chaque tâche peut créer du parallélisme de tâches (exemple : produits matrice \times vecteur).

2. Le second sous problème :

$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$

fait apparaître un parallélisme colossal : autant de systèmes d'EDO à résoudre que de points dans la grille de discrétisation.

1. Le premier sous-problème :

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, \dots, n.$$

fait apparaître un *parallélisme du pauvre* (n tâches indépendantes) ; mais l'exécution de chaque tâche peut créer du parallélisme de tâches (exemple : produits matrice x vecteur).

2. Le second sous problème :

$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$

fait apparaître un parallélisme colossal : autant de systèmes d'EDO à résoudre que de points dans la grille de discrétisation.

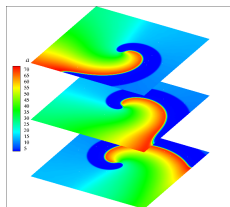
=> tâches= paquets de points ? taille des paquets ? **TBB fait ça pour vous.**

L'autre problème

$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$

L'autre problème

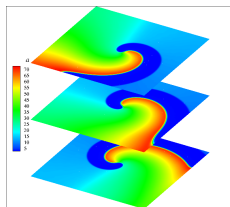
$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$



Réaction de
Belousov-Zhabotinsky
(Ondes spirales).

L'autre problème

$$\frac{\partial u_i}{\partial t}(x, t) = f_i(u_1(x, t), \dots, u_m(x, t)), \quad i = 1, \dots, n.$$



Réaction de
Belousov-Zhabotinsky
(Ondes spirales).

Zones à l'équilibre (coût de la résolution du système d'EDOs faible)
et zones loin de l'équilibre (coût élevé). => difficile de faire un
équilibrage des charges à-priori ; **TBB fait ça pour vous.**

Différents types de parallélisme

1. Parallélisme par les données.

Appliquer la même transformation à tous les éléments d'un ensemble, quand les transformations sont indépendantes.

Différents types de parallélisme

1. Parallélisme par les données.

Appliquer la même transformation à tous les éléments d'un ensemble, quand les transformations sont indépendantes.

2. Parallélisme par les tâches.

Graphe de dépendance de tâches ; regroupement pour le parallélisme de données.

Différents types de parallélisme

1. Parallélisme par les données.

Appliquer la même transformation à tous les éléments d'un ensemble, quand les transformations sont indépendantes.

2. Parallélisme par les tâches.

Graphe de dépendance de tâches ; regroupement pour le parallélisme de données.

3. Pipeline.

Appliquer plusieurs traitements successifs à une collection d'objets.

On retrouve tout ça dans TBB .

Exemples : des tâches complètement indépendantes.

Produit matrice x vecteur $Y = A.X$.

```
for i in [1..N] do  
   $Y_i = \langle A_{i,*}, X \rangle$   
end
```

TBB : PARALLEL_FOR

```
static const int n=1000;  
double a[n][n],x[n],y[n];  
.....
```

```
void prod(int n,double **a,double x[],double y[])  
{  
    for(int line=0;line<n;line++)  
    {  
        double pscal=0.0;  
        for(int col=0;col<n;col++)  
            pscal+=a[line][col]*x[col];  
        y[line]=pscal;  
    }  
}
```

TBB : PARALLEL_FOR

```
class Mprod{
    const int n;
    double **a, *x,*y;
public:

//constructeur:
    Mprod(int N,double **A,double *X,double *Y):
        n(N),a(A),x(X),y(Y){}

//constructeur par copie.
    Mprod(const Mprod& M):
        n(M.n),a(M.a),x(M.x),y(M.y){}
```

TBB : PARALLEL_FOR

Il faut ajouter une méthode, dont la signature est imposée :

```
void operator()(const blocked_range<size_t>& r) const  
{  
    for(int line=r.begin(); line<r.end(); line++)  
    {  
        double pscal=0.0;  
        for(int col=0; col<n; col++)  
            pscal+=a[line][col]*x[col];  
        y[line]=pscal;  
    }  
}
```

TBB : PARALLEL_FOR

Il faut ajouter une méthode, dont la signature est imposée :

```
void operator()(const blocked_range<size_t>& r) const  
{  
    for(int line=r.begin(); line<r.end(); line++)  
    {  
        double pscal=0.0;  
        for(int col=0; col<n; col++)  
            pscal+=a[line][col]*x[col];  
        y[line]=pscal;  
    }  
}
```

maintenant, je peux faire :

```
parallel_for(blocked_range<size_t>(0,n),  
             Mprod(n,a,x,y));
```

... et voilà !

TBB : PARALLEL_FOR

Tâchons de comprendre.

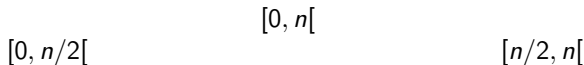
Décomposition récursive de $[0, n[$ jusqu'à un certain niveau (là, c'est TBB qui décide).

$[0, n[$

TBB : PARALLEL_FOR

Tâchons de comprendre.

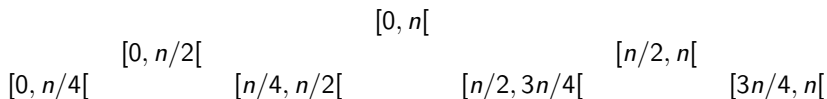
Décomposition récursive de $[0, n[$ jusqu'à un certain niveau (là, c'est TBB qui décide).



TBB : PARALLEL_FOR

Tâchons de comprendre.

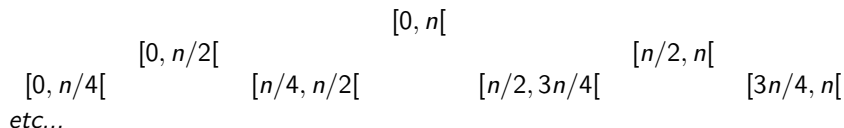
Décomposition récursive de $[0, n[$ jusqu'à un certain niveau (là, c'est TBB qui décide).



TBB : PARALLEL_FOR

Tâchons de comprendre.

Décomposition récursive de $[0, n[$ jusqu'à un certain niveau (là, c'est TBB qui décide).



Analysons :

1. `parallel_for` crée un objet de type `Mprod` avec l'intervalle $[0, n[$.
2. Deux nouveaux objets de type `Mprod` avec les intervalles $[0, n/2[$ et $[n/2, n[$ etc... Le constructeur par copie est utilisé.
3. les feuilles terminales sont les **tâches** indépendantes, qui n'ont plus qu'à être exécutées.

TBB : PARALLEL_FOR : commentaires et ajouts.

- ▶ le nombre de tâches créées est à-priori largement supérieur au nombre de fils de calcul disponibles,
- ▶ l'efficacité ne peut être obtenue que si chaque tâche créée est suffisamment *coûteuse*..

TBB : PARALLEL_FOR : commentaires et ajouts.

- ▶ le nombre de tâches créées est à-priori largement supérieur au nombre de fils de calcul disponibles,
- ▶ l'efficacité ne peut être obtenue que si chaque tâche créée est suffisamment *coûteuse*..

d'où quelques raffinements :

- ▶ L'objet de type `blocked_range<size_t>` contient un paramètre supplémentaire facultatif :
`blocked_range<size_t>(0,n) =>`
`blocked_range<size_t>(0,n,k)`
`k` limite la descente récursive à des intervalles de taille `k`.

TBB : PARALLEL_FOR : commentaires et ajouts.

- ▶ le nombre de tâches créées est à-priori largement supérieur au nombre de fils de calcul disponibles,
- ▶ l'efficacité ne peut être obtenue que si chaque tâche créée est suffisamment *coûteuse*..

d'où quelques raffinements :

- ▶ L'objet de type `blocked_range<size_t>` contient un paramètre supplémentaire facultatif :
`blocked_range<size_t>(0,n) =>`
`blocked_range<size_t>(0,n,k)`
k limite la descente récursive à des intervalles de taille k.
- ▶ `parallel_for` a un 3^e argument, facultatif : le `partitioner`.
Correspond à différents algorithmes pour le partitionnement de l'intervalle $[1, n[$. Par défaut, on utilise `auto_partitioner`.

TBB : PARALLEL_FOR : commentaires et ajouts.

- ▶ le nombre de tâches créées est à-priori largement supérieur au nombre de fils de calcul disponibles,
- ▶ l'efficacité ne peut être obtenue que si chaque tâche créée est suffisamment *coûteuse*..

d'où quelques raffinements :

- ▶ L'objet de type `blocked_range<size_t>` contient un paramètre supplémentaire facultatif :
`blocked_range<size_t>(0,n) =>`
`blocked_range<size_t>(0,n,k)`
`k` limite la descente récursive à des intervalles de taille `k`.
- ▶ `parallel_for` a un 3^e argument, facultatif : le `partitioner`.
Correspond à différents algorithmes pour le partitionnement de l'intervalle $[1, n[$. Par défaut, on utilise `auto_partitioner`.

Note : je n'ai jamais gagné grand chose à utiliser ces raffinements !

Réductions

Exemple emblématique : le produit scalaire : $s = \sum_{i=1}^{n-1} x_i \cdot y_i$.

Réductions

Exemple emblématique : le produit scalaire : $s = \sum_{i=1}^{n-1} x_i \cdot y_i$.

```
double dotprod(int n, double x[], double y[])
{
    double ret=0.0;
    for(int i=0;i<n;i++)
        ret+=x[i]*y[i];
    return ret
}
```


Réductions

Exemple emblématique : le produit scalaire : $s = \sum_{i=1}^{n-1} x_i \cdot y_i$.

```
double dotprod(int n, double x[], double y[])
{
    double ret=0.0;
    for(int i=0;i<n;i++)
        ret+=x[i]*y[i];
    return ret
}
```

TBB va :

- ▶ découper l'intervalle $[1, n[$ en sous intervalles (arbre binaire),
- ▶ calculer les produits scalaires sur les sous intervalles,
- ▶ faire remonter les résultats élémentaires dans l'arbre binaire.

Réductions

Exemple emblématique : le produit scalaire : $s = \sum_{i=1}^{n-1} x_i \cdot y_i$.

```
double dotprod(int n, double x[], double y[])
{
    double ret=0.0;
    for(int i=0; i<n; i++)
        ret+=x[i]*y[i];
    return ret
}
```

TBB va :

- ▶ découper l'intervalle $[1, n[$ en sous intervalles (arbre binaire),
- ▶ calculer les produits scalaires sur les sous intervalles,
- ▶ faire remonter les résultats élémentaires dans l'arbre binaire.

Ingrédients :

- ▶ parallel_reduce,
- ▶ une classe pour emballer le calcul.

Réductions

```
class dotprod{  
    double prod;  
    double *x,*y;  
public:  
    dotprod(double *X,double *Y): x(X),y(Y),  
                                   prod(0.0){}  
  
    dotprod(dotprod& D, split ):x(D.x),y(D.y),  
                                   prod(0.0){}  
  
    void join(const dotprod& D){prod+=D.prod;}
```

Réductions

```
class dotprod{  
    double prod;  
    double *x,*y;  
public:  
    dotprod(double *X,double *Y): x(X),y(Y),  
                                   prod(0.0){}  
  
    dotprod(dotprod& D, split):x(D.x),y(D.y),  
                                prod(0.0){}  
  
    void join(const dotprod& D){prod+=D.prod;}
```

Bien remarquer :

- ▶ la deuxième méthode *n'est pas* un constructeur de copie : l'argument `split` est là pour le marquer.
- ▶ c'est bien sûr `join` qui fait la réduction.

Réductions

Il reste à écrire l'opérateur qui calcule les produits scalaires sur les sous intervalles.

```
void operator()(const blocked_range<size_t>& r)
{
    prod=0.0;
    for(size_t i=r.begin(); i<r.end(); i++)
        prod+=x[i]*y[i];
}
double result() const {return prod;}
```

Réductions

Il reste à écrire l'opérateur qui calcule les produits scalaires sur les sous intervalles.

```
void operator()(const blocked_range<size_t>& r)
{
    prod=0.0;
    for(size_t i=r.begin(); i<r.end(); i++)
        prod+=x[i]*y[i];
}
double result() const {return prod;}
```

L'utilisation :

```
dotprod prod(x,y);
parallel_reduce(blocked_range<size_t>(0,n), prod);
double prodscal=prod.result();
```

Réductions : commentaires

```
dotprod prod(x,y);  
parallel_reduce(blocked_range<size_t>(0,n), prod);  
double prodsca=prod.result();
```

1. `parallel_reduce` reçoit une référence à `prod`.
2. lors de la remontée de l'arbre binaire, `join` met à jour `prod` des fils vers le père.
3. Même remarques que pour `parallel_for` : on peut limiter la taille des grains et/ou préciser un partitionneur.

Les Ranges

On a :

- ▶ `blocked_range`
- ▶ `blocked_range2d` : opère sur des produits cartésiens de deux intervalles semi ouverts.

Les Ranges

On a :

- ▶ `blocked_range`
- ▶ `blocked_range2d` : opère sur des produits cartésiens de deux intervalles semi ouverts.

En fait ces deux types (templates) sont des *modèles* de Range. On peut fabriquer ses propres Range à condition de se conformer au *concept* de Range.

Les Ranges

Le concept de Range

```
R(const R&)  
~R()  
empty() const  
is_divisible() const  
R(R& r, split)
```

Les Ranges

Le concept de Range

```
R(const R&)
~R()
empty() const
is_divisible() const
R(R& r, split)
```

La dernière méthode découpe `r` en deux `Range`. `split` est une classe (qui ne fait rien, vide), pour ne pas confondre la méthode avec un constructeur de copie.

Note : c'est vraiment du C++.

Retour a une problème précédemment évoqué

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, ..n.$$

Retour a une problème précédemment évoqué

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, ..n.$$

En fait cela revient, après discrétisation, à résoudre :

$$\frac{dU_i}{dt}(t) = A_i U_i, \quad i = 1, n.$$

où les A_i sont des matrices et les U_i de grands vecteurs.

Retour a une problème précédemment évoqué

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, ..n.$$

En fait cela revient, après discrétisation, à résoudre :

$$\frac{dU_i}{dt}(t) = A_i U_i, \quad i = 1, n.$$

où les A_i sont des matrices et les U_i de grands vecteurs.

En général, il faut, pour chaque équation ($i = 1, n$), résoudre des systèmes linéaires avec $A_i \Rightarrow$ (avec des méthodes itératives) :

Retour a une problème précédemment évoqué

$$\frac{\partial u_i}{\partial t}(x, t) - \varepsilon_i \Delta u_i(x, t) = 0. \quad i = 1, \dots, n.$$

En fait cela revient, après discrétisation, à résoudre :

$$\frac{dU_i}{dt}(t) = A_i U_i, \quad i = 1, n.$$

où les A_i sont des matrices et les U_i de grands vecteurs.

En général, il faut, pour chaque équation ($i = 1, n$), résoudre des systèmes linéaires avec $A_i \Rightarrow$ (avec des méthodes itératives) :

1. calculer des produits matrice x vecteur avec A_i ,
2. effectuer des produits scalaires
3. en nombre qui peut dépendre de i .

Retour a une problème précédemment évoqué

Deux niveaux de parallélisme

1. sur les équations :

```
parallel_for(blocked_range<size_t>(0,n-1),MesEquations);
```


Retour a une problème précédemment évoqué

Deux niveaux de parallélisme

1. sur les équations :

```
parallel_for(blocked_range<size_t>(0,n-1),MesEquations);
```

2. puis pour chaque équation, lancer les méthodes itératives, qui vont faire des produits matrice x vecteur (parallel_for !) et des produits scalaires (parallel_for !).

Retour a une problème précédemment évoqué

Deux niveaux de parallélisme

1. sur les équations :

```
parallel_for(blocked_range<size_t>(0,n-1),MesEquations);
```

2. puis pour chaque équation, lancer les méthodes itératives, qui vont faire des produits matrice x vecteur (parallel_for !) et des produits scalaires (parallel_for !).

- ▶ pas de synchronisation à gérer,
- ▶ bonne occupation des fils.

Conteneurs

Une tentative pour faire (une partie d')une STL parallèle.
La STL ne permet pas d'accès concurrent aux conteneurs =>
utilisation de verrous.

Conteneurs

Une tentative pour faire (une partie d')une STL parallèle.

La STL ne permet pas d'accès concurrent aux conteneurs => utilisation de verrous.

Les conteneurs TBB :

- ▶ verrouillage à grain fin.
- ▶ algorithmes ne necessitant pas de verrous (??).

Plus coûteux que les conteneurs de la STL.

Conteneurs

Une tentative pour faire (une partie d')une STL parallèle.
La STL ne permet pas d'accès concurrent aux conteneurs => utilisation de verrous.

Les conteneurs TBB :

- ▶ verrouillage à grain fin.
- ▶ algorithmes ne necessitant pas de verrous (??).

Plus coûteux que les conteneurs de la STL.

- ▶ `concurrent_queue`.
- ▶ `concurrent_vector`.
- ▶ `concurrent_hash_map`.

Par exemple, `concurrent_vector` permet un agrandissement sans risque.

Allocation mémoire

Comment faire pour que l'allocation mémoire :

1. passe à l'échelle,
2. concurrente.
3. pas de faux partage.

Allocation mémoire

Comment faire pour que l'allocation mémoire :

1. passe à l'échelle,
2. concurrente.
3. pas de faux partage.

Le faux partage ?

```
double x[1000], y[1000];
```

le fil 1 écrit en `x[999]` et le fil 2 écrit en `y[0]` **ET** `x[999]` et `y[0]` **sont dans la même ligne de cache.** \Rightarrow ça marche, mais au prix d'un ralentissement considérable !

Allocation mémoire

Deux allocateurs :

1. `scalable_allocator`
2. `cache_aligned_allocator`.

`cache_aligned_allocator =`
`scalable_allocator + alignement sur les lignes de caches.`

Allocation mémoire

Deux allocateurs :

1. `scalable_allocator`
2. `cache_aligned_allocator`.

`cache_aligned_allocator =`
`scalable_allocator` + alignement sur les lignes de caches.



ça craint un peu...

- ▶ allocation dans des blocs de mémoire différents.
- ▶ les versions anciennes (2011) de TBB fabriquaient du *gruyère de mémoire*.

Allouer, désallouer oui, mais pas trop en parallèle.

Exclusion mutuelle

Le style TBB : éviter les `MUTEX` et autres systèmes de verrous.
Si nécessaire : TBB fournit des classes de `MUTEX`.

Exclusion mutuelle

Le style TBB : éviter les `MUTEX` et autres systèmes de verrous.
Si nécessaire : TBB fournit des classes de `MUTEX`.

Les opérations atomiques

- ▶ beaucoup plus rapides que les mutex.
- ▶ uniquement des objets de petite taille (taille d'un `double`).
- ▶ liste des opérations possibles très limitées.

Opérations atomiques

Déclaration :

```
atomic<T> x
```

Exemple : `atomic<int> x`

On a les opérations suivantes :

Opération	Résultat
<code>=x</code>	lire la valeur de x
<code>x=</code>	affecter une valeur à x
<code>x.fetch_and_store(y)</code>	<code>x=y</code> et retourne la valeur de x
<code>x.fetch_and_add(y)</code>	<code>x+=y</code> et retourne l'ancienne valeur de x
<code>x.compare_and_swap(y,z)</code>	si <code>x==z</code> , <code>x=y</code> retourne l'ancienne valeur de x.

Opérations atomiques

Déclaration :

```
atomic<T> x
```

Exemple : `atomic<int> x`

On a les opérations suivantes :

Opération	Résultat
<code>=x</code>	lire la valeur de x
<code>x=</code>	affecter une valeur à x
<code>x.fetch_and_store(y)</code>	<code>x=y</code> et retourne la valeur de x
<code>x.fetch_and_add(y)</code>	<code>x+=y</code> et retourne l'ancienne valeur de x
<code>x.compare_and_swap(y,z)</code>	si <code>x==z</code> , <code>x=y</code> retourne l'ancienne valeur de x.
<code>++</code> , <code>--</code> , <code>+=</code> , <code>-=</code>	réalisées avec les opérations ci-dessus.

Opérations atomiques

Déclaration :

```
atomic<T> x
```

Exemple : `atomic<int> x`

On a les opérations suivantes :

Opération	Résultat
<code>=x</code>	lire la valeur de x
<code>x=</code>	affecter une valeur à x
<code>x.fetch_and_store(y)</code>	<code>x=y</code> et retourne la valeur de x
<code>x.fetch_and_add(y)</code>	<code>x+=y</code> et retourne l'ancienne valeur de x
<code>x.compare_and_swap(y,z)</code>	si <code>x==z</code> , <code>x=y</code> retourne l'ancienne valeur de x.
<code>++, --, +=, -=</code>	réalisées avec les opérations ci-dessus.

Exemple : `static atomic<int> counter`
dans une classe passée à `parallel_for`.



Forcément, ça coûte un peu.

Installation et mise en œuvre

Installation

- ▶ Compilateur Intel : livrée avec.
- ▶ g++ ou autre : téléchargement du source et installation (Makefile).

Le source contient des tests et des exemples.

Il y a des mises à jour fréquentes : la bibliothèque évolue.

Pour les masochistes : ça marche même sous Windows !.

Installation et mise en œuvre

Mise en œuvre

Dans le préambule :

- ▶ `#include "tbb/tbb.h"`
- ▶ `using namespace tbb;`

Dans main :

`task_scheduler_init init;` ← nombre de fils est choisi par TBB .

ou

`task_scheduler_init init(5);` // ← limité à 5 fils.

- ▶ On peut donc tester le programme en séquentiel, mais avec le mécanisme de TBB .
- ▶ on peut mesurer le passage à l'échelle.

Installation et mise en œuvre

Mise en œuvre

Dans le préambule :

- ▶ `#include "tbb/tbb.h"`
- ▶ `using namespace tbb;`

Dans main :

`task_scheduler_init init;` ← nombre de fils est choisi par TBB .

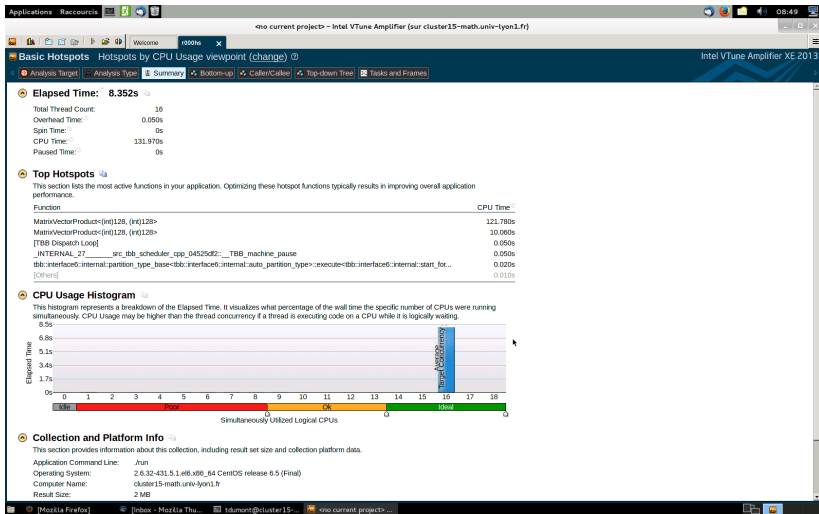
ou

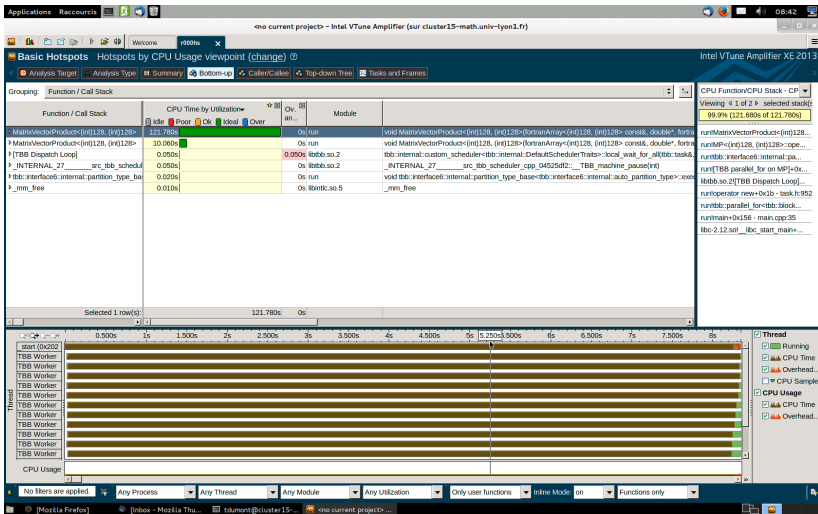
`task_scheduler_init init(5);` // ← limité à 5 fils.

- ▶ On peut donc tester le programme en séquentiel, mais avec le mécanisme de TBB .
- ▶ on peut mesurer le passage à l'échelle.

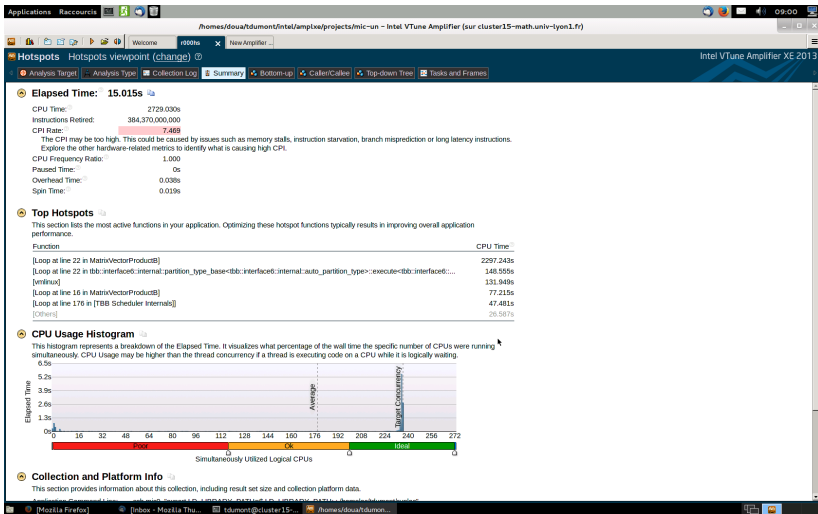
Édition des liens :

- ▶ `-ltbb -ltbbmalloc_proxy -ltbbmalloc`
- ▶ il existe une version debug de `tbbmalloc`.

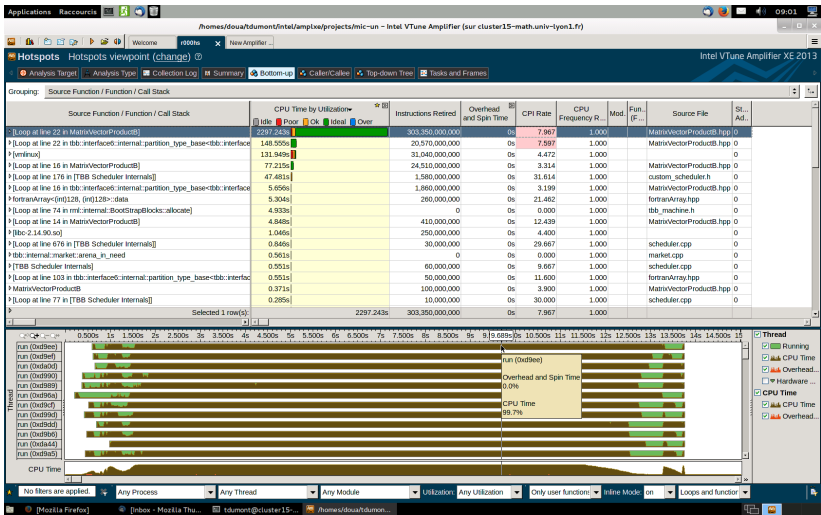




Vtune sur le Xeon-Phi



Vtune sur le Xeon-Phi



Conclusion

- ▶ facile et agréable à utiliser,
- ▶ pas forcément facile à déboguer,
- ▶ un outil de mesures de performances est bien venu (VTune).

Conclusion

- ▶ facile et agréable à utiliser,
- ▶ pas forcément facile à déboguer,
- ▶ un outil de mesures de performances est bien venu (VTune).

Merci !

