

# C++ : Multithreading & Parallelism



# Les multiples vies du *multithreading*

HARDWARE

## Pentium 4

1er Processeur pour  
PC permettant le  
multithreading



2002

2011

2020

**AMD Ryzen 3990x**  
1er Processeur pour  
PC atteignant les **128**  
**threads (64 coeurs)**

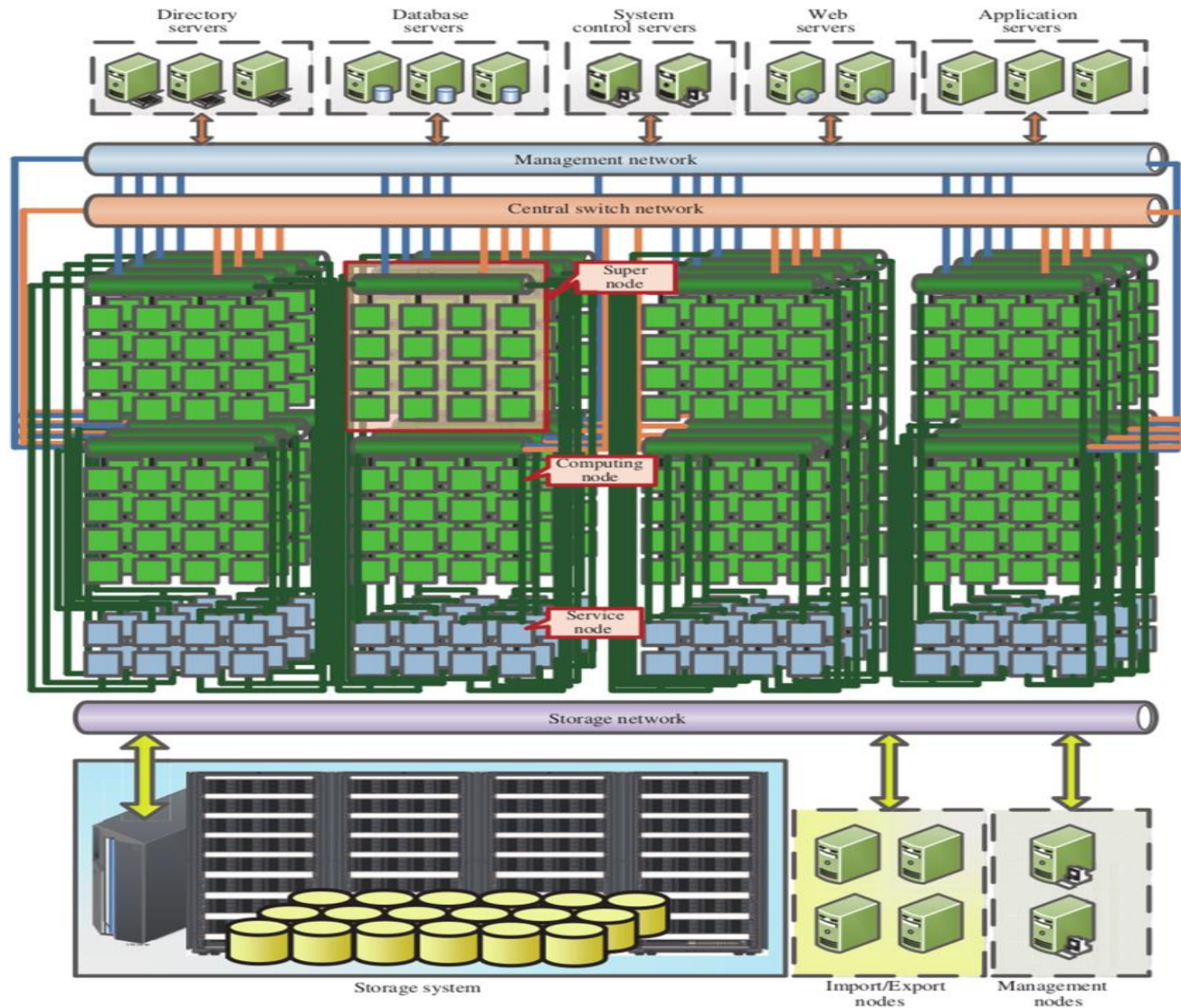


Compilateurs & Langages



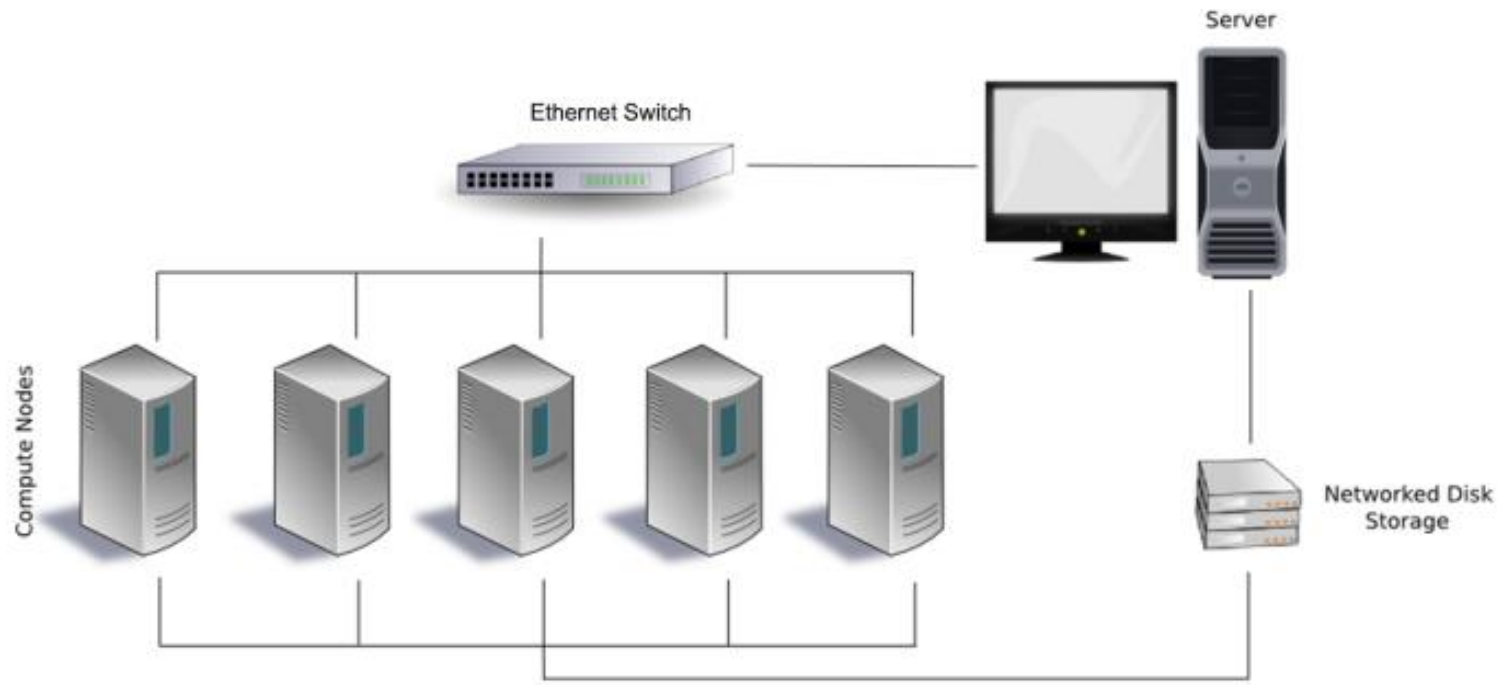
**std::thread >> C++11**  
Gestion native des  
threads (norme  
ISO/CEI 14882:2011).

Des « *nœuds* » de calculs, en veux-tu en voilà !



# Des *noeuds* de calculs (nodes), en veux-tu en voilà !

- Les **clusters de calculs** ne sont plus réservés aux projets à très grande échelle (la baisse des coûts des machines intermédiaires, la connexion rapide et le routage facile et peu onéreux en Ethernet)



# Concurrence **Versus** Parallélisme

## Concurrence

2 files et 1 machine à café.

Threads exécutés sur le même processeur:

- Partage les ressources d'un unique cœur;
- Chaque thread dispose de ses propres registres et pointeurs d'instruction.

Augmente l'utilisation du cœur en introduisant un parallélisme eu niveau des instructions



## Parallélisme

2 files et 2 machines à café.

Les tâches sont réparties sur plusieurs processeurs.



## Concurrence

- **Accès simultané à une même ressource** (e.g., mémoire, machine à café)
  - **Risque de corruption** (i.e., collusion), e.g., :
    - un « *thread* » modifie une variable alors qu'un autre récupère sa valeur ;
    - Pb appelé “**Data Race**”
  - **Besoin de synchroniser** grâce à des verrous (**locking**) les accès mémoire des *threads* pour les instructions liées à la mémoire partagée (“**section critique**”) et ouvert à l'écriture (et non que en lecture).

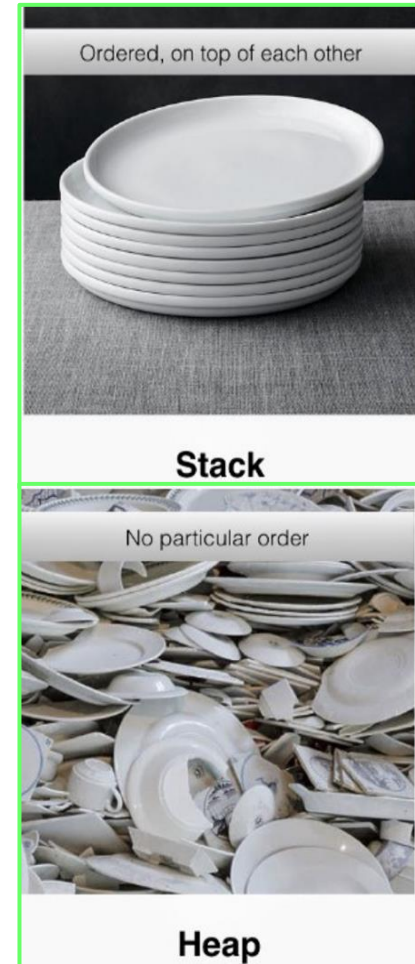
## Parallélisme

- **Exécution simultanée de plusieurs tâches.**
  - pouvant être complètement indépendantes
  - tente (donc pas systématiquement) d'éviter la concurrence, e.g., :
    - La mémoire est séparée
    - Distribuable facilement sur des énormes ferme de calculs (i.e., sur beaucoup de cœurs/nœuds de calculs).

# Hello *multithreading* worlds!

Quelques points importants :

- Les *threads* du C++ s'accordent (*mapping*) un à un avec les *threads* systèmes.
- Chaque *thread* peut se caractériser comme une séquence d'instructions que le processeur doit traiter.
- Chaque processeur ne peut généralement exécuter qu'un seul thread à la fois. Des technologies comme l'« hyper threading » peuvent permettre d'exécuter plusieurs threads sur un même processeur (deux cœurs logiques pour un cœur physique)
- Chaque thread possède sa propre pile (Stack) mais tous ensemble ils partagent la mémoire du tas (Heap).
- En conséquence, les variables partagées sont donc susceptibles de concurrences malheureuses entre threads.



**Pic (et bonnes infos sur ces deux mémoires) :**

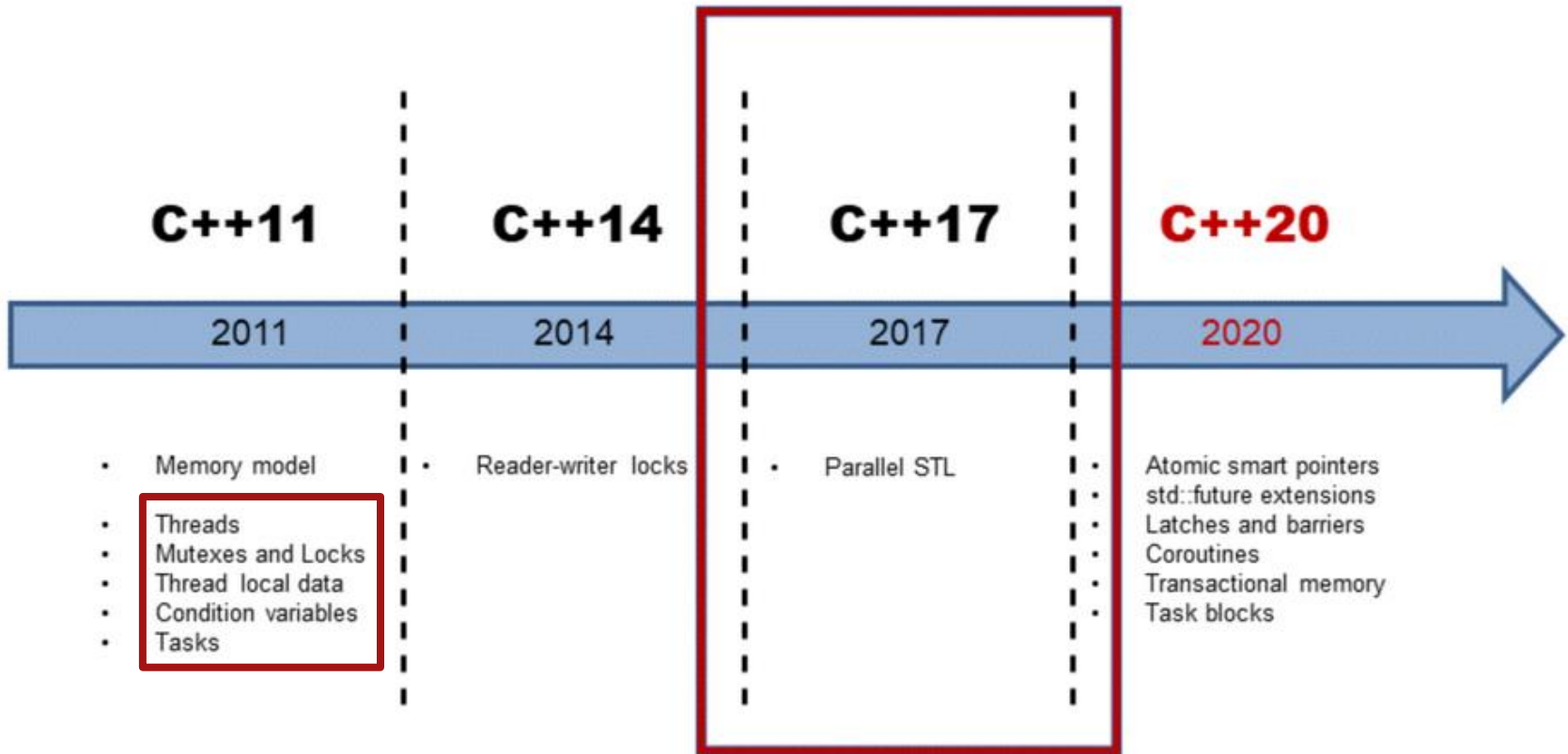
<https://medium.com/thinkel/confused-about-stack-and-heap-2cf3e6adb771>.

# Concurrency & *Threads*

- Points à prendre en compte :
  - la gestion des *threads* est complexe.
  - La pertinence de l'utilisation de beaucoup (trop) de *thread* doit être discutée :
    - la création de *threads* est relativement coûteuse, alors :
    - pour être efficace, il faut créer un nombre optimal de *thread nécessaire*.

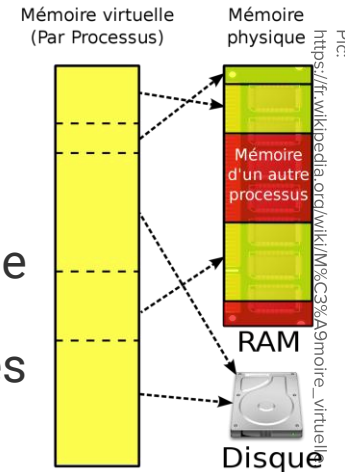


# C++ concurrency & parallelism



# Join() the thread(s)

- **thread**
  - **Unité d'exécution** d'instructions du langage d'un processeur,
  - Dispose de **sa propre pile d'appels**,
  - Peut être exécuté en parallèle (mais attention au partage de ressource),
  - Peut avoir à partager sa **mémoire virtuelle** avec les autres *threads* du même processus,
    - Ceci permet un lancement plus rapide.



```
std::thread monObjetThread(nomDeLaFonctionAExecuter, paramètre(s)DelaFonction);
```

- **join**
  - Bloque le thread jusqu'à la fin de son exécution. **Le programme ne devrait pas se terminer si tous les *threads* "jointes" ne le sont pas.**
  - Une fois la fonction d'un thread terminée ce dernier est terminé.

```
monObjetThread.join();
```

# Création de *Threads* en C++

- **Exemples:**


- `std::thread threadNumberOne (ma_fonction);`
  - `threadNumberOne` exécutera `ma_fonction` qui n'a pas de paramètre.
- `std::thread threadNumberTwo (ma_fonction2, premierParametre, secondParametre);`
  - `threadNumberTwo` exécutera `ma_fonction2` qui a deux paramètres renseignés dans le constructeur du thread .

# Hello ~~parallel~~ threading worlds!

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

void versLaPremiereMarcheDesEcolesDInge(int place)
{
    for (int i = place ; i > 0 ; i--) std::cout << i << ' ' ;
}

int main(){
    // Classement des ecoles base sur usinenouvelle.com 2019
    std::thread threadISEN(versLaPremiereMarcheDesEcolesDInge, 69);
    std::thread threadHEI(versLaPremiereMarcheDesEcolesDInge, 96);
    std::thread threadISA(versLaPremiereMarcheDesEcolesDInge, 119);
    threadISEN.join();
    threadHEI.join();
    threadISA.join();
    return 0 ;
}
```



Lorsque la fonction avec ses paramètres est terminée, le thread est terminé.

## Questions.

- Par intuition, que va-t-il s'afficher sur le terminal ?
- Quelles sont les mots clés qui vous paraissent liés au *multithreading* ?

# Hello ~~parallel~~ threading worlds!

```
(base) |> g++ testThread.cpp -o helloParallelWorlds -pthread
(base) |> ./helloParallelWorlds
6996 9568 67 66 65 64 63 9462 61 60 59 58 5793 56 55 54 53 52 51 50 49 48 47 46 9245 4491 90
43 42 41 40 8939 119 88 118 117 116 115 114 1133887 86 85 84 83 37 36 35 34 33 32 31 30 29 28 27
26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 82 81 80 79 78 77 76 75 74
73 72 71 70 69 68 67112 111 110 109 66 65 64 63 62 61 10860 59107 106 105 104 103 102 101 100
99 98 5897 57 56 55 54 53 9652 95 94 93 92 91 90 89 88 87 86 5185 50 49 48 47 46 45 44 8443 42
41 40 39 38 37 8336 82 81 80 79 78 3577 76 75 74 73 72 3471 33 32 31 30 29 28 7027 69 68 67 6
6 65 6426 63 62 61 60 59 58 57 56 55 54 2553 52 51 50 49 48 2447 23 22 21 20 19 1846 45 44 43 4
2 41 4017 16 15 14 13 12 11 10 9 8 397 38 37 36 35 34 336 5 4 3 2 1 32 31 30 29 28 27 26 25 24
23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 (base) |> █
```

## - Mutex & Threads

- Le désordre provient du fait que **l'accès à la console (cout est concurrentiel (threadISEN, threadHEI et threadISA veulent y avoir accès).**
- Lorsque la ressource est critique...le problème est critique...
- On peut **synchroniser** l'accès aux ressources grâce aux **Mutex** ou aux variables de condition.

# Hello ~~parallel~~ threading worlds! (MAJ Exemple)

```
std::mutex lock;

void versLaPremiereMarcheDesEcolesDInge(int place) {
    lock.lock();
    for (int i = place ; i > 0 ; i--) std::cout << i << ' ';
    lock.unlock();
}

int main(){ // Classement des ecoles base sur usinenouvelle.com 2019
    std::thread threadISEN(versLaPremiereMarcheDesEcolesDInge, 69);
    std::thread threadHEI(versLaPremiereMarcheDesEcolesDInge, 96);
    std::thread threadISA(versLaPremiereMarcheDesEcolesDInge, 119);
    threadISEN.join();
    threadHEI.join();
    threadISA.join();
    return 0 ;
}
```

## Questions.

a. Par intuition, que va-t-il s'afficher sur le terminal ?

# Hello ~~parallel~~ threading worlds!

## Welcome `mutex`

```
spydel@spydel-NUC10i5FNH:~/Documents/vracCode$ g++ MISC.cpp -lpthread
spydel@spydel-NUC10i5FNH:~/Documents/vracCode$ ./a.out
69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 3
8 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6
5 4 3 2 1 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69
68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 3
7 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4
3 2 1 119 118 117 116 115 114 113 112 111 110 109 108 107 106 105 104 103 102 101 100 99 98 9
7 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77 76 75 74 73 72 71 70 69 68 67 66
65 64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35
34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 s
pydel@spydel-NUC10i5FNH:~/Documents/vracCode$
```

# Retour sur le TP 8

Produit scalaire de deux vecteurs (dot product) :  $\sum_{i=1}^n a(i) * b(i)$

```
template <typename Iter>
void dot_product(const Iter &v1_begin, const Iter &v1_end, const Iter &v2_begin,
                 typename Iter::value_type &result) {
    Iter iter1 = v1_begin, iter2 = v2_begin;
    while (iter1 != v1_end) {
        result += *iter1 * *iter2;
        ++iter1;
        ++iter2;
    }
}
```

Utilisation de `<valarray>` ?



# Produit scalaire de deux vecteurs avec « dot\_product » (séquentiel)

```
int main(){  
    constexpr size_t nb_elems    = 500000000;  
    constexpr size_t nb_iter     = 10;
```

```
    typedef double T;
```

```
    vector<T> va1(nb_elems), va2(nb_elems);  
    T incr = 1. / static_cast<T>(nb_elems);  
    for (size_t i = 0; i < nb_elems; ++i)  
        va1[i] = static_cast<T>(i) * incr;  
    for (size_t i = 0; i < nb_elems; ++i)  
        va2[i] = static_cast<T>(nb_elems - i) * incr;
```

Initialisation des vecteurs

```
    auto start = chrono::high_resolution_clock::now();
```

Mesure du temps d'exécution

```
    T result_global = static_cast<T>(0);  
    for (size_t i = 0; i < nb_iter; ++i) {  
        T result = static_cast<T>(0);  
        dot_product(va1.begin(), va1.end(), va2.begin(), result);  
        result_global += result;
```

Lancement de nb\_iter iterations  
du produit scalaire

```
    }  
    cout << "Time taken by function (dot_product): "  
        << chrono::duration_cast<chrono::microseconds>  
        (chrono::high_resolution_clock::now() - start).count() << " microseconds" << endl << flush;  
    return EXIT_SUCCESS;
```

```
}
```

# Produit scalaire de deux vecteurs avec C++ (séquentiel)

```
start = chrono::high_resolution_clock::now();
```

Mesure du temps d'exécution

```
T result1_global = static_cast<T>(0);  
#include <numeric>  
for (size_t i = 0; i < nb_iter; ++i) {  
    T result1 = inner_product(va1.begin(), va1.end(), va2.begin(), static_cast<T>(0));  
    result1_global += result1;  
}
```

```
cout << "Time taken by function (inner_product): "  
      << chrono::duration_cast<chrono::microseconds>(chrono::high_resolution_clock::now() -  
start).count() <<  
      " microseconds" << endl << flush;  
//
```

```
start = chrono::high_resolution_clock::now();
```

```
T result2_global = static_cast<T>(0);  
#include <numeric>  
for (size_t i = 0; i < nb_iter; ++i) {  
    T result2 = std::transform_reduce(execution::seq, va1.cbegin(), va1.cend(), va2.cbegin(),  
    .0, std::plus<T>(), std::multiplies<T>());  
    result2_global += result2;  
}
```

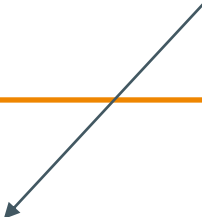
```
cout << "Time taken by function (transform_reduce): "  
      << chrono::duration_cast<chrono::microseconds>(chrono::high_resolution_clock::now() -  
start).count() <<  
      " microseconds" << endl << flush;
```

## Produit scalaire de deux vecteurs avec C++ (Multithread)

```
start = chrono::high_resolution_clock::now();
```

seq (C++17)  
par (C++17)  
par\_unseq (C++17)  
unseq (C++20)

```
T result3_global = static_cast<T>(0);  
for (size_t i = 0; i < nb_iter; ++i) {  
    T result3 = std::transform_reduce(execution::par, va1.cbegin(), va1.cend(), va2.cbegin(),  
        .0, std::plus<T>(), std::multiplies<T>());  
    result3_global += result3;  
}
```



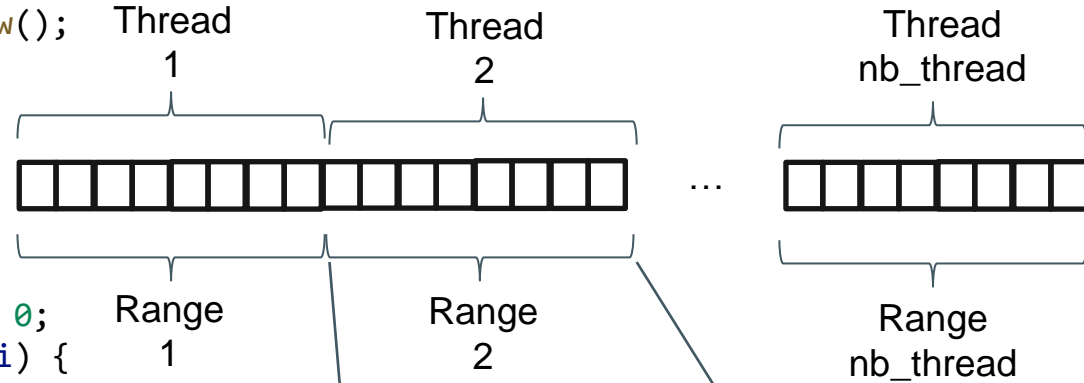
```
cout << "Time taken by function (transform_reduce parallel): "  
    << chrono::duration_cast<chrono::microseconds>(chrono::high_resolution_clock::now() -  
        start).count() << " microseconds" << endl << flush;
```

# Produit scalaire de deux vecteurs avec C++ thread (Multithread)

```
constexpr size_t nb_threads = 8;
```

```
#include <thread>
```

```
start = chrono::high_resolution_clock::now();
T result4_global = static_cast<T>(0);
array<thread, nb_threads> threads;
array<T, nb_threads> results;
for (size_t i = 0; i < nb_iter; ++i) {
    T result4 = static_cast<T>(0);
    size_t range = nb_elems / nb_threads;
    size_t lower_bound = 0, upper_bound = 0;
    for (size_t i = 0; i < nb_threads; ++i) {
        results[i] = 0;
        upper_bound = lower_bound + range;
        if (i == nb_threads - 1) {
            upper_bound = nb_elems;
        }
    }
```



Création des « threads »

```
threads[i] = thread(dot_product<vector<T>::iterator>,
                    va1.begin() + lower_bound, va1.begin() + upper_bound,
                    va2.begin() + lower_bound, ref(results[i]));
```

```
lower_bound += range;
```

```
}
for (size_t i = 0; i < nb_threads; ++i) {
    threads[i].join();
}
```

```
for (size_t i = 0; i < nb_threads; ++i) {
    result4 += results[i];
}
```

```
result4_global += result4;
```

En attente que tous les « threads » finissent

## Produit scalaire de deux vecteurs avec OpenMP (Multithread)

```
start = chrono::high_resolution_clock::now();
auto start_omp = omp_get_wtime();
```

```
cout<<"Number of threads max (OpenMP) = "<<omp_get_max_threads()<<endl;
T result5_global = static_cast<T>(0);
for (size_t i = 0; i < nb_iter;++i) {
    T result5 = 0;
```

```
#pragma omp parallel for default(none) \
    shared(va1,va2) reduction(+:result5)
    for (size_t i = 0; i < va1.size(); i++) {
        result5 += va1[i] * va2[i];
    }
    result5_global += result5;
}
```

```
auto end_omp = omp_get_wtime();
```

```
cout << "Time taken by function (OpenMP): "<< end_omp - start_omp << "sec.\n";
```

```
cout << "Time taken by function (OpenMP parallel for): "
    << chrono::duration_cast<chrono::microseconds>(chrono::high_resolution_clock::now() -
start).count() << " microseconds" << endl << flush;
```

**OpenMP** (Open Multi-Processing) est une [interface de programmation](#) pour le [calcul parallèle](#) sur architecture à mémoire partagée. Il se présente sous la forme d'un ensemble de [directives](#), d'une [bibliothèque logicielle](#) et de [variables d'environnement](#).

## Produit scalaire de deux vecteurs avec BLAS (Multithread)

```
double dot_product(const vector<double> &v1, const vector<double> &v2){  
    int n = v1.size();  
    assert(n == v2.size());  
    int one = 1;  
    return cblas_ddot(n, &v1.front(), one, &v2.front(), one);  
}
```

Bibliothèque BLAS (INTEL MKL)

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.

```
.....  
start = chrono::high_resolution_clock::now();  
  
T result6_global = static_cast<T>(0);  
for (size_t i = 0; i < nb_iter;++i) {  
    T result6 = dot_product(va1, va2);  
    result6_global += result6;  
}  
cout << "Time taken by function blas: "  
    << chrono::duration_cast<chrono::microseconds>  
(chrono::high_resolution_clock::now() - start).count() << " microseconds" << endl << flush;  
.....
```

## Résultats

Time taken by function (dot\_product): 6170086 microseconds  
Time taken by function (inner\_product): 6117333 microseconds  
Time taken by function (transform\_reduce): 6077379 microseconds

Sequentiel

%CPU %MEM

100.0 60.5

Time taken by function (transform\_reduce parallel): 2747281 microseconds  
Time taken by function (dot\_product thread): 2835172 microseconds  
Number of threads max (OpenMP) = 8  
Time taken by function (OpenMP): 2.81895sec.  
Time taken by function (OpenMP parallel for): 2825619 microseconds  
Time taken by function BLAS: 3364237 microseconds

Parallèle

%CPU %MEM

749.0 60.5

↖  
Sous Unix (problème mesure du temps  
sous Windows) ~800% pour 8 threads

Les temps pour les calculs « séquentiels »  
sont approximativement équivalents



Les temps pour les calculs « parallèles » sont  
approximativement équivalents sauf avec BLAS  
qui devrait pourtant être le plus performant (à  
confirmer...)

Pour cet exemple (calcul numérique), les méthodes  
« multithread » les plus faciles à déployer sont celles  
utilisant : la STL (transform\_reduce), OpenMP (par  
directives) et BLAS (bibliothèque spécialisée)



En revanche, les threads peuvent être plus  
généralement utilisés.