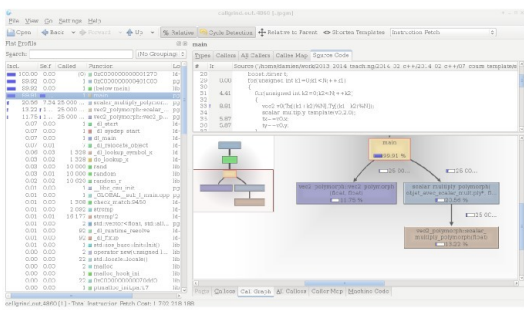


# C++

## Optimisation

#pragma omp

parallel for



# Optimisation

• 1ère règle de la bonne optimisation:

Ne pas optimiser son code!

Optimisation est la **dernière chose à faire**, ne surtout pas chercher à optimiser dès le début.

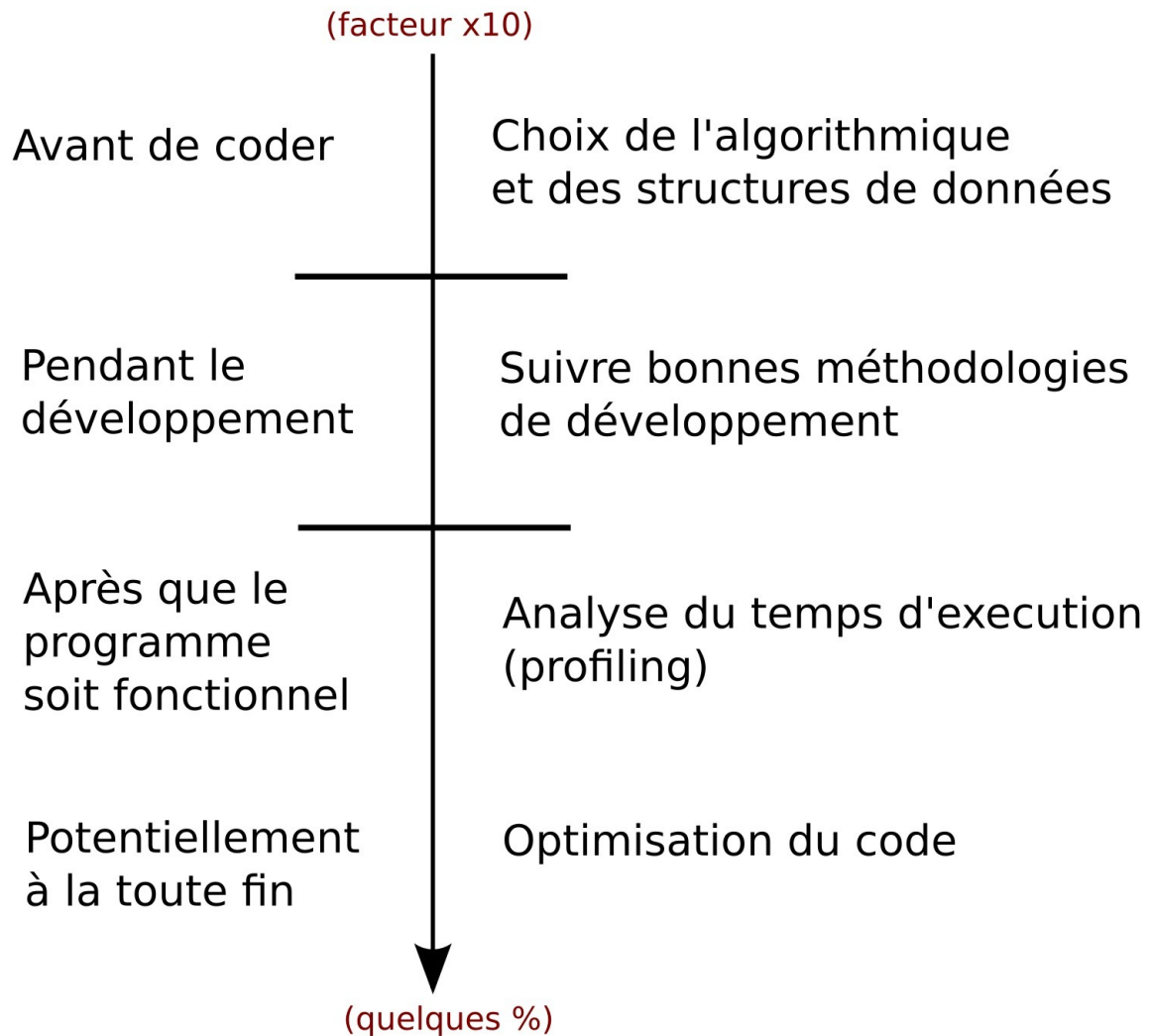
Rend le code plus compliqué, plus de lignes, moins générique, moins lisible

**Donald Knuth:**

*We should forget about small efficiencies,  
say about 97% of the time:*

***premature optimization is the root of all evil.***

# Méthodologie d'optimisation



# Choix structures données

## . Recherche de valeurs, insertions, suppressions

- . Recherche très rapide, peu d'insertions `std::unordered_map`
- . Beaucoup d'insertions, ordonnancement `std::map`

## . Pas de recherche, insertions/suppression locale `std::list`

## . Accès direct, pas d'insertions/suppression

- . Taille fixe, faible dimension `std::array`
- . Taille variable, dimension importante `std::vector`

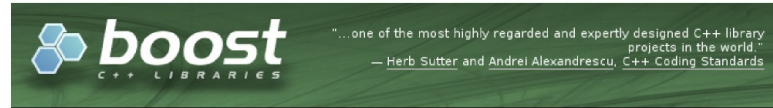
## . Récupération meilleur valeur, insertion constante `std::priority_queue`

## . Organisation en graphe `boost:graph`

...

# Choix structures données

- Utilisez la **STL**
- Utilisez la **boost**
  - D'avantages de structures de données
  - Future de la STL



## BOOST LIBRARY DOCUMENTATION

[All](#) [Categorized](#) | [Standard Proposals](#) | [TR1 libraries](#)

Sort by: [Name](#) | [First Release](#)

### Accumulators

Framework for incremental calculation, and collection of statistical accumulators.

*Author(s).....* Eric Niebler  
*First Release..* 1.36.0  
*Standard.....*  
*Categories.....* [Math and numerics](#)

### Algorithm

A collection of useful generic algorithms.

*Author(s).....* Marshall Clow  
*First Release..* 1.50.0  
*Standard.....*  
*Categories.....* [Algorithms](#)

### Any

Safe, generic container for single values of different value types.

*Author(s).....* Kevin Henney  
*First Release..* 1.23.0  
*Standard.....*  
*Categories.....* [Data structures](#)

### Array

STL compliant container wrapper for arrays of constant size.



Google Custom Search

[WELCOME](#)

[INTRODUCTION](#)

[COMMUNITY](#) >

[DEVELOPMENT](#)

[DOCUMENTATION](#)

Getting Started

Libraries

[1.55.0 - Current Release](#)

[1.54.0](#)

[1.53.0](#)

[1.52.0](#)

[1.51.0](#)

[1.50.0](#)

[1.49.0](#)

[1.48.0](#)

[1.47.0](#)

[1.46.1](#)

[1.45.0](#)

[1.44.0](#)

[1.43.0](#)

[1.42.0](#)

[1.41.0](#)

# Bonnes pratiques de code

- Ecrivez du code structuré et lisible.  
Séparez vos traitements en fonctions.

90% des optimisations sont réalisées par le compilateur.

Mais il faut le laisser libre d'optimiser, il faut pouvoir analyser votre code.

- Evitez les allocations dynamiques inutiles

`T[N]`, `std::array<T,N>` plutôt que `new T[N]`

- Passez vos arguments de fonctions en références constantes  
Idem pour les valeurs de retours de fonction.

sauf built-in (int,float, ...)

ex. ~~`fonction(std::string s)`~~      `fonction(std::string const& s)`  
`fonction(TYPE const& x);`

- Quand le développement est terminé:  
Compilez avec l'option **-O3** de gcc.

Optimisation la plus élevée

# Profiling

- Profileur libre: Valgrind + KCacheGrind

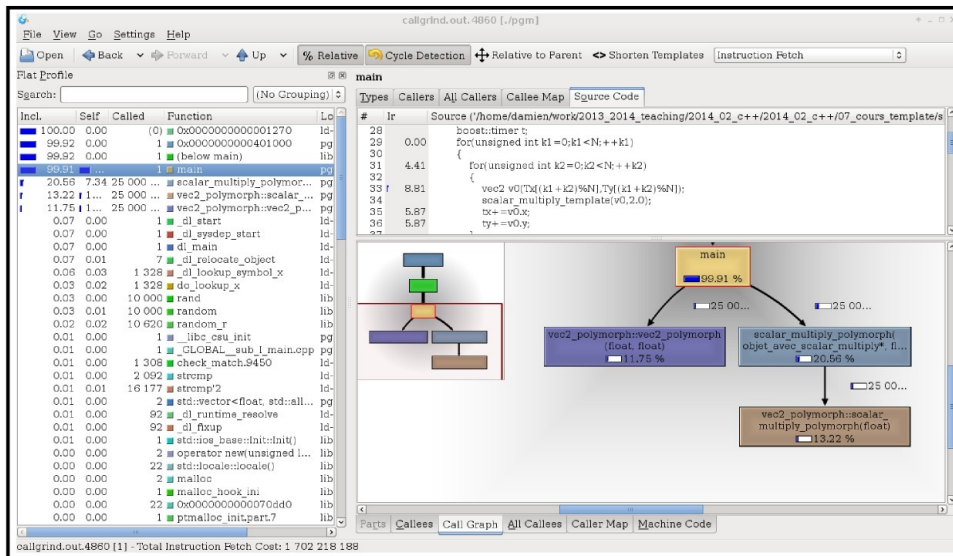
Ne pas chercher à optimiser sans avoir utilisé de profiler

1. Compilez en mode debug
2. Lancez valgrind en mode profiling

```
valgrind --tool=callgrind ./mon_executable
```

3. Visualisez le temps passé dans chaque fonction

```
kcachegrind callgrind.out.xxxx
```



# Optimisation du code

- Evitez la création d'objet temporaires
  - Préférez `a+=b` à `c=a+b`
  - Préférez allocation sur la pile plutôt que sur le tas
- Calculez en float plutôt que double
- Calculez en int plutôt que float
- Précalculez les valeurs de fonction mathématiques
  - Table LUT
  - Méta-programmation (`constexpr`, ...)
- Parallélisez

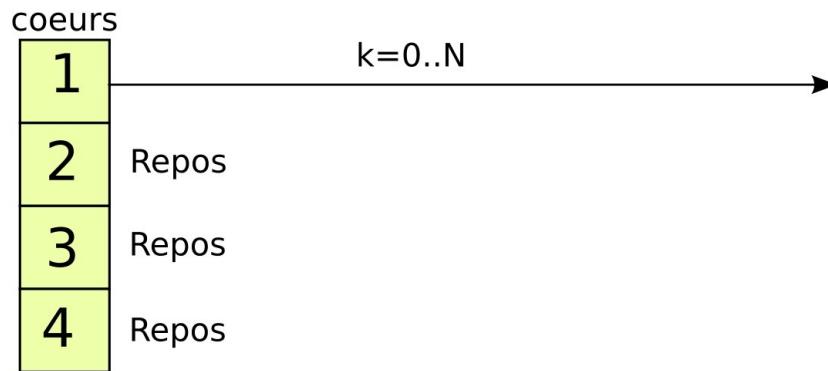


# Parallélisation

## Principe

```
for (k=0;k<N;++k)  
    fonction_de_calcul_lente();
```

## Non parallélisé

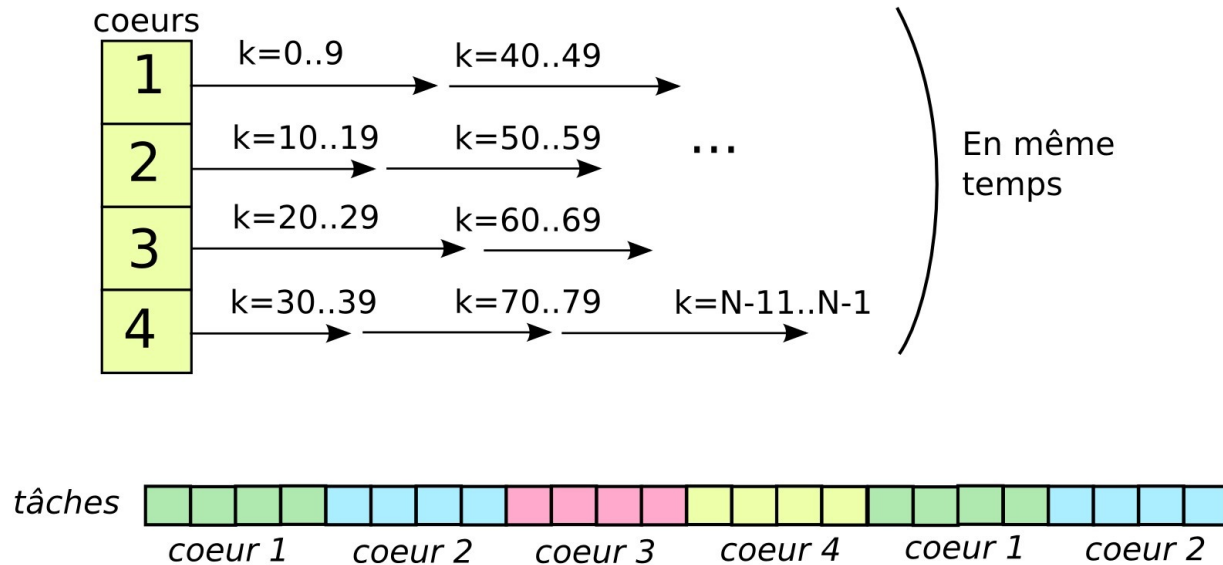


# Parallélisation

## Principe

```
for (k=0;k<N;++k)
    fonction_de_calcul_lente();
```

## Version parallélisé



# Parallélisation

## Implémentation

- Threads 

```
pthread_t threads[N];  
pthread_create(...);
```

Créez une liste de tâche, ordonner les threads, etc  
=> Fastidieux

- OpenMP

```
#pragma omp parallel for  
for (k=0;k<N;++k)  
    fonction_de_calcul_lente();
```

Boucle parallélisée

# Exemple OpenMP

```
float calcul_longueur_courbe(float r,  
                             const std::vector<float>& tab_rand)  
{  
  
    const unsigned int N_sample=tab_rand.size()/2;  
    float L=0.0f;  
    for(unsigned int k_sample=0;k_sample<N_sample-1;++k_sample)  
    {  
        float x0=tab_rand[2*k_sample+0];  
        float y0=tab_rand[2*k_sample+1];  
        float x1=tab_rand[2*(k_sample+1)+0];  
        float y1=tab_rand[2*(k_sample+1)+1];  
  
        L+=std::pow(std::pow(std::fabs(x1-x0),r)+  
                   std::pow(std::fabs(y1-y0),r),1.0f/r)  
    }  
    return L;  
}
```

```
#pragma omp parallel for  
for(unsigned int k=0;k<N;++k)  
{  
    float norme_alpha=8.0f*static_cast<float>(k)/N;  
    aire[k]=calcul_longueur_courbe(norme_alpha,tab_rand);  
}
```

Sans omp

```
1 [ | 0.7%]  
2 [ | 0.7%]  
3 [ | 0.0%]  
4 [ ||| 100.0%]  
Mem [ ||| 1820/3820MB]  
Swp [ | 0/5718MB]
```

9.8s

Avec omp

```
1 [ ||| 100.0%]  
2 [ ||| 100.0%]  
3 [ ||| 100.0%]  
4 [ ||| 100.0%]  
Mem [ ||| 1801/3820MB]  
Swp [ | 0/5718MB]
```

3.2s

Compiler avec  
gcc `-fopenmp` ...

# Utilisation d'OpenMP

- Boucle avec calculs longs

ex. Multiplication matricielle  
Traitement d'images, vidéos, etc...

- Les traitements doivent être indépendants !

$v[k] = f(v[k-1])$  Ne fonctionne pas

Algo récursifs non parallélisables

- Les traitements doivent être longs

Inutile : somme d'un vecteur de 1000 éléments

# OpenMP

```
int main()
{
#pragma omp parallel
{
    std::cout<<"Hello World!"<<std::endl;
}

return 0;
}
```

`gcc` `pgm.c` `-fopenmp`

Processeurs 4 coeurs: *Hello World!Hello World!*

*Hello World!*

*Hello World!*

# OpenMP

parallel for: ordre non prévisible

```
int main()
{
#pragma omp parallel for
    for(int k=0;k<10;++k)
    {
        std::cout<<k<<std::endl;
    }

    return 0;
}
```

08  
1  
2  
3  
64  
5  
7  
9

# OpenMP

## Affichage des ID des threads

```
#pragma omp parallel for
  for(int k=0;k<10;++k)
  {
#pragma omp critical
  {
    std::cout<<omp_get_thread_num()<<"/"
              <<omp_get_num_threads()<<std::endl;
  }
}
```

*zone critique:  
1 seul thread à la fois*

*ID thread courant*

*nbr total thread*

0/4  
0/4  
1/4  
1/4  
0/4  
1/4  
3/4  
3/4  
2/4  
2/4

Par défaut: nbr thread=nbr coeurs

└─> set\_num\_threads(N)



# Réductions en OpenMP

```
int sum=0;
#pragma omp parallel for
for(int k=1;k<10;++k)
{
    sum += k;
}
std::cout<<sum<<std::endl;
```

Résultats aléatoires et **faux** !

sum est partagée pour tous les threads  
sum+=k n'est pas atomique

# Réductions en OpenMP

```
    int sum=0;
#pragma omp parallel for
    for(int k=1;k<10;++k)
    {
#pragma omp critical
        sum += k;
    }
    std::cout<<sum<<std::endl;
```

Résultat juste ...

Mais plus aucune parallélisation:  
plus lent que le code original

# Réductions en OpenMP

```
int sum=0;
#pragma omp parallel for reduction(+:sum)
for(int k=1;k<10;++k)
{
    sum += k;
}
std::cout<<sum<<std::endl;
```

Définit sum en mémoire locale aux thread

(sum\_1, sum\_2, sum\_3, sum\_4)

Réalise les sommes en parallèles sur les variables locales

Réunit à la fin.

(sum=sum\_1+sum\_2+sum\_3+sum\_4)

# Mesure temps avec OpenMP

```
double start=omp_get_wtime();  
  
int sum=0;  
#pragma omp parallel for reduction(+:sum)  
for(int k=1;k<10;++k)  
{  
    sum += k;  
}  
std::cout<<sum<<std::endl;  
  
double end=omp_get_wtime();  
std::cout<<"temps = "<<end-start<<"s"<<std::endl;
```

Se méfier des autres mesures avec les threads  
(ex. boost::timer cumule le temps des threads)

# Accélération avec OpenMP

- Il est inutile de paralléliser quelques opérations simples et rapides

Version non parallèle

```
double start=omp_get_wtime();  
  
int sum=0;  
  
for(int k=1;k<10;++k)  
{  
    sum += k;  
}  
  
double end=omp_get_wtime();  
  
std::cout<<sum<<std::endl;  
std::cout<<"temps = "<<end-start<<"s"<<std::endl;
```

$2.4 \cdot 10^{-7} \text{ s}$

Version parallèle

```
double start=omp_get_wtime();  
  
int sum=0;  
#pragma omp parallel for reduction(+:sum)  
for(int k=1;k<10;++k)  
{  
    sum += k;  
}  
  
double end=omp_get_wtime();  
  
std::cout<<sum<<std::endl;  
std::cout<<"temps = "<<end-start<<"s"<<std::endl;
```

$2.4 \cdot 10^{-3} \text{ s}$

10000x plus lent!

=> Création, synchronisation des threads possède un cout!  
Cout de la fonction doit être supérieur au cout de gestion des threads

# Accélération avec OpenMP

- Commence à devenir intéressant pour 100 millions d'opérations

```
double start=omp_get_wtime();

unsigned long int sum=0;
#pragma omp parallel for reduction(+:sum)
for(unsigned long int k=0;k<100000000; ++k)
{
    sum += k;
}

double end=omp_get_wtime();

std::cout<<sum<<std::endl;
std::cout<<"temps = "<<end-start<<"s"<<std::endl;
```

non parallélisé                      parallélisé

0.005s **VS** 0.003s

- Dépend de nombreux facteurs  
(architecture, compilateur, niveau d'optimisation, etc)

# Accélération avec OpenMP

- Calculs plus complexes sont plus intéressants

```
double start=omp_get_wtime();

double sum=0.0f;
#pragma omp parallel for reduction(+:sum)
for(unsigned long int k=1;k<100000000; ++k)
{
    sum += tan(1.4*k)*tanh(0.25*k+1.35/k)+cos(7.4*(k-5));
}

double end=omp_get_wtime();

std::cout<<"temps = "<<end-start<<"s"<<std::endl;
```

Non parallélisé

16.7s

Parallélisé

7.7s

# Autre type de parallélisation

- Parallélisation sur clusters de PC: **MPI (Message Passing Interface)**

Parallélisation de type fork(), échange par msg

- Parallélisation sur carte graphique: (GP)**GPU**

2000 threads de calculs en parallèles