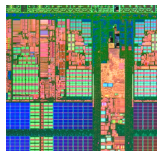
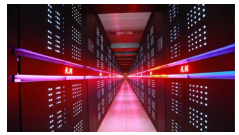
Cours SIM203

Initiation au calcul haute performance

OpenMP *(compléments)*

Axel Modave — 22 avril 2024

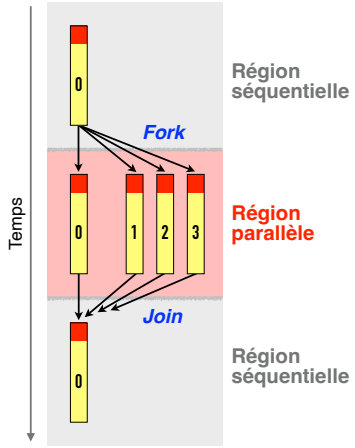
Rappel : Vision globale de calcul haute performance ...

Cœur	**Processeur** *(plusieurs cœurs)*	**Cluster** *(plusieurs processeurs)*
		
Calcul séquentiel Calcul vectoriel	Calcul parallèle à ***mémoire partagée***	Calcul parallèle à ***mémoire distribuée***
Séances 1 et 2	Les cœurs peuvent travailler en parallèle sur des tâches différentes. Ils partagent des mémoires rapides (RAM + L2 ou L3) **Séances 3 et 4**	Les processeurs peuvent travailler en parallèle sur des tâches différentes Les données sont distribuées entre les RAM des processeurs, qui doivent communiquer (avec par ex. **MPI**). **Séance 5** *(intro)* **Cours AMS301 et AMS-I03** ***(3A ModSim et M2 AMS)***

Questions importantes:
Comment **gérer les opérations** ?
(Quel cœur fait quoi ? Quand ?)
Comment **gérer les données** nécessaires pour les opérations ?

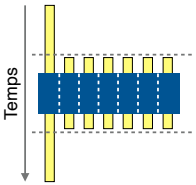
Rappel : Modèle de programmation OpenMP

- Un programme est une alternance de **régions séquentielles** et **parallèles**.
- Au début d'une région parallèle, le **thread maître** crée des **threads esclaves**, qui disparaissent en fin de région parallèle. ***(modèle fork-join)***
- Les threads sont exécutés de façon **concurrente**. Ils sont affectées aux cœurs par le gestionnaire des tâches.
- La région parallèle se termine lorsque tous les threads ont été traités (***synchronisation implicite***).



0 – Thread **maître**
1,2,3 – Threads **esclaves**

Rappel : Parallélisation d'une boucle FOR




```
double a[N], b[N], c[N];
#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    int my_start = N*myRank/numThreads;
    int my_end   = N*(myRank+1)/numThreads;
    for(int i=my_start; i<my_end; i++)
      c[i] = a[i] + b[i];
  }
```

Exemple : Addition de deux tableaux (les trois versions sont équivalentes!)

```
double a[N], b[N], c[N];
#pragma omp parallel
  {
#pragma omp for
    for(int i=0; i<N; i++)
      c[i] = a[i] + b[i];
  }
```

```
double a[N], b[N], c[N];
#pragma omp parallel for
  for(int i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

Gestion des données :
- Les tableaux `a`, `b` et `c` sont **partagés**.
- Les variables `myRank`, `my_start`, ... sont **privées**.

Partage du travail :
- La directive `#pragma omp parallel for` répartit automatiquement les itérations de la boucle entre différents threads.

OpenMP – Parallélisation des boucles

OpenMP – Étude d'un cas (suite)

OpenMP – Commandes synchronisantes

OpenMP – Équilibrage de charge

Conditions pour paralléliser une boucle

```
#pragma omp parallel for
  for(int i=0; i<N; i++)
    c[i] = a[i] + b[i];
```

- **En principe**, OpenMP ne parallélise que des **boucles for** dont le **nombre d'itérations** est **déterminé avant l'exécution de la boucle** et **défini dans les parenthèse** après le **`for`**.

  Les boucles doivent être sous forme canonique :

```
for [ index = start ; index < end | index <= end | index >= end | index > end ; index++ | ++index | index-- | --index | index += incr | index -= incr | index = index + incr | index = index - incr | index = incr + index | index = incr - index ]
```

  → **index** ne doit être modifié que par l'expression d'incrément de la boucle.
  → **start**, **end** et **incr** ne doivent pas être modifiés pendant l'exécution de la boucle.

Exemples de boucles qui ne sont pas parallélisées :

```
for(int i=0; i<N; i++){
  if(condition) break;
}
```

Boucle avec un break

```
while(condition){
  …
}
```

Boucle conditionnelle

Dans ces deux cas, une erreur sera générée à la compilation !

*Pour les autres cas, le compilateur ne vous signalera pas toujours que vous faites une erreur : **prudence** !*

Conditions pour paralléliser une boucle

- **En principe**, **une itération** ne doit **pas dépendre** du résultat d'**une autre itération**.
  Les itérations doivent pouvoir être effectuées de façon concurrente!

Exemple de calcul qui ne peut pas être parallélisé : Calcul des nombres de Fibonacci par récurrence

```
vector<int> A(N);
A[0] = 1;
A[1] = 1;
for (int i=2, i<N, i++)
  A[i] = A[i-1] + A[i-2];
```

```
A[2] = A[1] + A[0];
A[3] = A[2] + A[1];
A[4] = A[3] + A[2];
A[5] = A[4] + A[3];
…
```

```
>> icpc fibonacci.cpp
```

```
>> ./a.out
N ? 10
A[0] = 1
A[1] = 1
A[2] = 2
A[3] = 3
A[4] = 5
A[5] = 8
A[6] = 13
A[7] = 21
A[8] = 34
A[9] = 55
```

Chaque itération a besoin des résultats des deux itérations précédentes.

Les itérations **doivent** être réalisées successivement.
Elle ne peuvent pas être réalisées en concurrence.

Conditions pour paralléliser une boucle

- **En principe**, **une itération** ne doit **pas dépendre** du résultat d'**une autre itération**.
  Les itérations doivent pouvoir être effectuées de façon concurrente!

Exemple de calcul qui ne peut pas être parallélisé : Calcul des nombres de Fibonacci par récurrence

```
vector<int> A(N);
A[0] = 1;
A[1] = 1;
#pragma omp parallel for
for (int i=2, i<N, i++)
  A[i] = A[i-1] + A[i-2];
```

```
A[2] = A[1] + A[0];
A[3] = A[2] + A[1];
A[4] = A[3] + A[2];
A[5] = A[4] + A[3];
…
```

```
>> icpc -qopenmp fibonacci.cpp
```

```
>> export OMP_NUM_THREADS=1
>> ./a.out
N ? 10
A[0] = 1
A[1] = 1
A[2] = 2
A[3] = 3
A[4] = 5
A[5] = 8
A[6] = 13
A[7] = 21
A[8] = 34
A[9] = 55
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
N ? 10
A[0] = 1
A[1] = 1
A[2] = 2
A[3] = 3
A[4] = 5
A[5] = 8
A[6] = 0
A[7] = 0
A[8] = 0
A[9] = 0
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
N ? 10
A[0] = 1
A[1] = 1
A[2] = 2
A[3] = 3
A[4] = 0
A[5] = 0
A[6] = 0
A[7] = 0
A[8] = 0
A[9] = 0
```

Data Race!

Le compilateur autorise la parallélisation,
mais le résultat est faux !

Quelques stratégies pour contourner les dépendances …

- **Changer l'algorithme**

  Il n'est pas possible de paralléliser la formule récursive pour calculer les nombres de Fibonacci, mais il y a une autre formule …

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \quad n = 3, \ldots \end{aligned}$$

Formule récursive
Non-parallélisable

$$F_n = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}, \quad n = 1, \ldots \quad \text{avec} \quad \varphi = \frac{1+\sqrt{5}}{2}$$

Formule de Binet (non-récursive)
Parallélisable

- **Utiliser une réduction**

```
double sum = 0.;
for(int i=0; i<N; i++)
  sum = sum + f(i);
```

```
double sum = 0.;
#pragma omp parallel for reduction(+:sum)
for(int i=0; i<N; i++)
  sum = sum + f(i);
```


Quelques stratégies pour contourner les dépendances …

- **Réarranger la boucle**

  Parfois, une inspection de la boucle permet d'identifier les opérations problématiques et, en réordonnant cette boucle, de supprimer ces dépendances.

```
for(int i=1; i<N; i++){
  y[i] = f( x[i-1] );
  x[i] = x[i] + c[i];
}
```

```
y[1] = f( x[0] );
#pragma omp parallel for
for(int i=1; i<N-1; i++){
  x[i] = x[i] + c[i];
  y[i+1] = f( x[i] );
}
```

- **Parallélisation partielle**

```
for(int i=1; i<N; i++)
  for(int j=1; j<N; j++)
    A[i][j] = A[i-1][j] + A[i-1][j-1];
```

```
for(int i=1; i<N; i++)
#pragma omp parallel for
  for(int j=1; j<N; j++)
    A[i][j] = A[i-1][j] + A[i-1][j-1];
```

Pour chaque ligne `i`, on a besoin des nouvelles valeurs de la ligne `i-1`. (dépendance)
En revanche, les éléments d'une ligne donnée peuvent être calculés en parallèle. (parallélisation OK)

Quelques stratégies pour contourner les dépendances …

- **Scission**

  Lorsqu'une boucle contient une partie parallélisable et une partie non-parallélisable, on peut la scinder pour séparer ces parties.

```
double sum=0.;
for(int i=1; i<N; i++){
  A[i] = A[i-1] + B[i];
  sum += f( A[i] );
}
```

```
a[1] = A[0] + B[1];
a[2] = A[1] + B[2];
a[3] = A[2] + B[3];
```

Écriture puis Lecture !

```
double sum=0.;
for(int i=1; i<N-1; i++){
  A[i] = A[i-1] + B[i];
#pragma omp parallel for reduction(+:sum)
for(int i=1; i<N; i++)
  sum += f( A[i] );
```

- **Gestion des anti-dépendances**

  Une variable est utilisée par une itération, puis modifiée par une itération ultérieure.
  Avec le calcul concurrent : risque que la *nouvelle valeur* soit utilisée à la place de l'*ancienne*.

```
for(int i=0; i<N-1; i++)
  A[i] = A[i+1] + B[i];
```

```
a[0] = A[1] + B[0];
a[1] = A[2] + B[1];
a[2] = A[3] + B[2];
```

Lecture puis Écriture !

```
for(int i=0; i<N-1; i++)
  tmp[i] = A[i+1];
#pragma omp parallel for
for(int j=1; j<N; j++)
  A[i] = tmp[i] + B[i];
```

Stratégie : sauvegarder temporairement les anciennes valeurs.

Combiner vectorisation et parallélisation OpenMP

Codes "`addition.cpp`"

1
```
for(int i=0; i<N; i++)
  A[i] = A[i] + B[i];
```

2
```
#pragma omp parallel for
for(int i=0; i<N; i++)
  A[i] = A[i] + B[i];
```

Rapports de compilation *(avec compilateur Intel)*

```
for(int i=0; i<N; i++)
```
```
LOOP BEGIN at /auto/m/modave/…
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```
```
  A[i] = A[i] + B[i];
```

```
#pragma omp parallel for
```
```
/auto/m/modave/myDirectory/:OMP:main:
OpenMP DEFINED LOOP WAS PARALLELIZED
LOOP BEGIN at /auto/m/modave/…
<Peeled loop for vectorization>
LOOP END
LOOP BEGIN at /myDirectory/…
   remark #15300: LOOP WAS VECTORIZED
LOOP END
LOOP BEGIN at /auto/m/modave…
<Remainder loop for vectorization>
LOOP END
```
```
for(int i=0; i<N; i++)
  A[i] = A[i] + B[i];
```

Avec la **parallélisation**, le travail est réparti sur plusieurs cœurs.

Avec **`#pragma omp parallel for`**, les itérations sont attribués par *(gros)* blocs aux différents threads.

Les threads sont attribués aux différents cœurs pendant le run.

Sur chaque cœur, les itérations sont réalisées par *(petits)* blocs *"vectoriels"*. (**vectorisation**)

Combiner vectorisation et parallélisation

Comparaison

	Vectorisation	Parallélisation avec OpenMP
Objectif	Utiliser les unités de calcul vectorielles	Utiliser plusieurs cœurs en même temps
Type de calcul	**SIMD** *(Single Instruction Multiple Data stream)*	**MIMD** *(Multiple Instructions Multiple Data stream)*
Critères sur les itérations	Les itérations font strictement les **mêmes tâches** Les itérations doivent *(en principe)* être **indépendantes**	Les itérations peuvent faire des **tâches différentes** Les itérations doivent *(en principe)* être **indépendantes**
Critères sur les données	Alignement strict des données	*Pas de besoin* d'alignement des données
Comment utiliser ?	**Vectorisation automatique** *+ directives de compilations, librairies, …*	**Directives de compilation** *+ librairies, options de compilation, …*

Étude d'un cas : Différences finies (suite)

Implémentation 3

```
int main(){

  vector<double> C(N * N);
  vector<double> Cnew(N * N);
  …
  double coef1 = dt/(dx*dx);
  double coef2 = 1 - 4*coef1;

// Version 3

  for(int n=0; n<T; n++){
    for(int i=1; i<(N-1); i++)
      for(int j=1; j<(N-1); j++)
        Cnew[N*i+j]
          = coef2 * C[N*i+j]
          + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j]
                    + C[N*i+(j+1)] + C[N*i+(j-1)] );
    C.swap(Cnew);
  }
}
```

Cette boucle est vectorisée.

```
>> icpc -O3 -qopenmp diffusionOmp.cpp
>> ./a.out 3 10000 512
```


Étude d'un cas : Différences finies (suite)

Implémentation 4

```
int main(){

  vector<double> C(N * N);
  vector<double> Cnew(N * N);
  …
  double coef1 = dt/(dx*dx);
  double coef2 = 1 - 4*coef1;

// Version 4

  for(int n=0; n<T; n++){
#pragma omp parallel for
    for(int i=1; i<(N-1); i++)
      for(int j=1; j<(N-1); j++)
        Cnew[N*i+j]
          = coef2 * C[N*i+j]
          + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j]
                    + C[N*i+(j+1)] + C[N*i+(j-1)] );
    C.swap(Cnew);
  }
}
```

Cette boucle est parallélisée.
Cette boucle est vectorisée.

```
>> icpc -O3 -qopenmp diffusionOmp.cpp
>> ./a.out 4 10000 512
```

Étude d'un cas : Différences finies (suite)

Implémentation 5

```
int main(){

  vector<double> C(N * N);
  vector<double> Cnew(N * N);
  …
  double coef1 = dt/(dx*dx);
  double coef2 = 1 - 4*coef1;

// Version 5

  for(int n=0; n<T; n++){
    for(int i=1; i<(N-1); i++)
#pragma omp parallel for
      for(int j=1; j<(N-1); j++)
        Cnew[N*i+j]
          = coef2 * C[N*i+j]
          + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j]
                    + C[N*i+(j+1)] + C[N*i+(j-1)] );
    C.swap(Cnew);
  }
}
```

Cette boucle parallélisée et *(peut-être)* est vectorisée

```
>> icpc -O3 -qopenmp diffusionOmp.cpp
>> ./a.out 5 10000 512
```


Étude d'un cas : Différences finies (suite)

Temps de calcul *(en secondes)*

Machines de l'ENSTA

Sans vectorisation

```
>> icpc -O0 -qopenmp …
```

#Threads	Implément. 4	Implément. 5
1	21.2	23.4
2	10.6	14.1
3	7.4	11.4
4	**5.7**	9.4
5	9.9	25.0
6	11.0	29.8
8	6.3	32.7
16	7.3	66.6

Avec vectorisation

```
>> icpc -O3 -qopenmp …
```

#Threads	Implément. 4	Implément. 5
1	2.14	4.0
2	1.09	4.2
3	0.77	4.1
4	**0.61**	4.2
5	1.19	15.4
6	1.04	21.4
8	0.97	25.1
16	0.81	55.7

En général, meilleure solution : { **Paralléliser** la boucle la plus **extérieure** / **Vectoriser** la boucle la plus **intérieure**

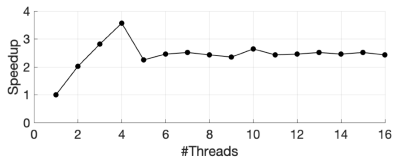
Étude d'un cas : Différences finies (suite)

- L'amélioration de performance est mesurée en utilisant le **speedup** :

$$\textbf{Speedup} = \frac{\text{Temps } \textit{avant} \text{ amélioration}}{\text{Temps } \textit{après} \text{ amélioration}}$$

L'*amélioration* peut *par exemple* être une stratégie algorithmique ou de programmation.

- En parallélisant avec OpenMP, on peut ***au mieux*** atteindre un speedup égal au nombre de cœurs. *(Il y a 4 cœurs ici!)*



Implémentation 4

```
>> icpc -O3 -qopenmp …
```

#Threads	Temps [s]	Speedup
1	2.14	–
2	1.06	2.0
3	0.76	2.8
4	**0.60**	**3.6**
5	0.95	2.6
6	0.87	2.5
7	0.85	2.5
8	**0.88**	**2.4**
9	0.91	2.3
10	0.81	2.6
11	0.88	2.4
12	**0.87**	**2.4**
13	0.85	2.5
14	0.87	2.5
15	0.85	2.5
16	**0.88**	**2.4**

The Best!

Clause NOWAIT et directive BARRIER

- La clause **nowait** indique que les threads ne doivent pas être synchronisés à la fin de la boucle (si combiné avec **#pragma omp for**) ou des sections (si avec **...sections**)
- La directive **#pragma omp barrier** impose une synchronisation explicite.
  Le système attend que tous les threads aient terminé leurs tâches. ***C'est coûteux !!!***

```
#pragma omp parallel reduction(+:tot)
{
#pragma omp for nowait
  for (int i=0; i<N; i++)
    z[i] = x[i] + y[i];
#pragma omp for nowait
  for (int i=0; i<M; i++)
    a[i] = b[i] + c[i];
#pragma omp barrier
  tot += sum(a,M) + sum(z,N);
}
```

Pas de synchronisation

Pas de synchronisation

Synchronisation explicite

Les calculs de la deuxième boucle pourront commencer avant que la première boucle ne soit terminée!

DATA RACE

```
int main()
{
  int i=0;
#pragma omp parallel
  {
    i++;
  }
  printf("%i\n", i);
  return 0;
}
```

```
>> export OMP_NUM_THREADS=100
>> ./a.out
100
>> ./a.out
98
>> ./a.out
99
```

Phénomène du "Data Race" :

- Plusieurs threads utilisent une variable partagée, et au moins un thread écrit dessus.
- Si l'ordre d'accès des threads est non-déterministe, le résultat final ***peut varier suivant le run*** et ***ne peut pas être prédit***.

Directive ATOMIC

- La directive **#pragma omp atomic** assure qu'une variable partagée est lue et modifiée avec une instruction simple par un seul thread à la fois *(mais dans n'importe quel ordre)*.
  Son effet est local à l'instruction qui suit immédiatement la directive.
- L'instruction peut être : **x = x(op)** ou **x = x(op)exp** ou **x = f(x,exp)**

op	++ -- + - * / && || == != …
f	max min …
exp	une expression arithmétique indépendante de **x**

```
int main()
{
  int i=0;
#pragma omp parallel
  {
#pragma omp atomic
    i++;
  }
  printf("%i\n", i);
  return 0;
}
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
4
>> export OMP_NUM_THREADS=100
>> ./a.out
100
```


Directive CRITICAL

- La directive **#pragma omp critical** assure que la section de code qui suit est exécutée par tous les threads de façon séquentielle *(mais dans n'importe quel ordre)*.
- La directive **critical** peut être vue comme une généralisation de la directive **atomic**, mais elle peut s'appliquer sur une plus grande section de code *(pas seulement une instruction)*. Cependant, elle est beaucoup plus lente!

```
int main()
{
  int s=0, p=1;
#pragma omp parallel
  {
#pragma omp critical
    {
      s++;
      p*=2;
    }
  }
  printf("%i %i\n", s, p);
  return 0;
}
```

```
>> export OMP_NUM_THREADS=10
>> ./a.out
10 1024
>> export OMP_NUM_THREADS=20
>> ./a.out
20 1048576
```

Sans la directive `critical` ...

```
>> export OMP_NUM_THREADS=20
>> ./a.out
19 524288
>> ./a.out
20 1048576
>> ./a.out
18 262144
```

Data Race!

OpenMP – Parallélisation des boucles
OpenMP – Étude d'un cas (suite)
OpenMP – Commandes synchronisantes
OpenMP – Équilibrage de charge

Répartition des itérations (par défaut)

Comment sont répartis les itérations entre les threads ?

```
int main(){
  vector<int> iter(N);
#pragma omp parallel for
  for(int i=0; i<N; i++)
    iter[i] = omp_get_thread_num();
  for(int i=0; i<N; i++)
    cout << i << " " << iter[i] << "\n";
}
```

```
>> icpc -qopenmp schedule.cpp
>> export OMP_NUM_THREADS=4
>> ./a.out
Number of iterations? 16
0 0
1 0
2 0
3 0
4 1
5 1
6 1
7 1
8 2
9 2
10 2
11 2
12 3
13 3
14 3
15 3
```

Répartition des itérations par défaut :
Les itérations sont divisés en blocs d'itérations consécutives, de taille équivalente.

Les blocs sont attribués aux threads dans l'ordre.

N° thread	Blocs
0	0–3
1	4–7
2	8–11
3	12–15

Exemple pour 16 itérations et 4 threads

Étude d'un cas "mal équilibré"

```
int main(){
  double sum=0.;
#pragma omp parallel for reduction(+:sum)
  for(int i=0; i<N; i++)
    sum += f(i);
  for(int i=0; i<N; i++)
    cout << i << " " << iter[i] << "\n";
}
```

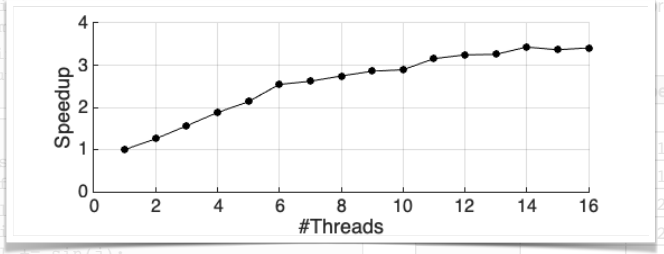
```
double f(int i){
  int start = i*(i+1)/2;
  int finish = start+i;
  double val = 0.;
  for(int j=start; j<=finish; j++)
    val += sin(j);
  return val;
}
```

```
>> icpc -qopenmp balancing.cpp
>> export OMP_NUM_THREADS=X
>> ./a.out 20000
```

Ici, les itérations ont des coûts différents.
(L'itération `i` aura un coût proportionnel à `i`.)
Une répartition équitable des itérations entre les threads mène à un déséquilibre de charge entre les threads.

#Threads	Temps [s]	Speedup
1	2.84	–
2	2.30	1.2
3	1.68	1.7
4	1.35	2.1
5	1.18	2.4
6	1.09	2.6
7	1.03	2.7
8	0.97	2.9
9	0.95	3.0
10	0.91	3.1
11	0.88	3.2
12	0.87	3.3

Étude d'un cas "mal équilibré"



Équilibrage de charge — Clause SCHEDULE

- Pour avoir de bonnes performances, on cherche à maximiser l'utilisation de tous les cœurs.
  ***Idéalement*, la charge de calcul doit être répartie équitablement sur tous les cœurs.**

- Pour des **opérations régulières**, l'équilibre de charge n'est **pas un problème**.
  Chaque itération possède la même charge, et la répartition sur les threads sera équitable.

  Exemples :
  - Addition de deux tableaux
  - Intégration numérique
  - Différences finies

- Pour des **opérations irrégulières**, il est important de **vérifier** si la charge est bien répartie.
  Au besoin, il faut modifier le programme pour **améliorer la répartition**.

  Exemples :
  - Transposition de matrice
  - Multiplication de matrices triangulaires
  - Recherche parallèle dans une liste liée

Pour les boucles avec des opérations irrégulières, la `clause schedule` permet d'adapter la répartition des itérations entre les différents threads, et éventuellement d'équilibrer la charge de calcul.

Équilibrage de charge — Clause SCHEDULE

```
#pragma omp parallel for schedule(static,X)
```

`schedule(static,X)`
Les itérations sont groupées en blocs de taille **`X`** *(sauf peut-être pour le dernier)*.
Les blocs sont attribués aux threads d'une façon cyclique ***dans l'ordre***.

`schedule(static)`
Les itérations sont groupées en **`#Threads`** blocs d'itérations consécutives (*i.e.* avec **`X=N/T`**.).
Les blocs sont attribués aux threads d'une façon cyclique ***dans l'ordre***.

Répartition des itérations avec `schedule(static,X)` :

N° thread	Blocs				
	`(static)`	`(static,1)`	`(static,2)`	`(static,3)`	`(static,4)`
0	0–3	0,4,8,12	0–1,8–9	0–2,12–14	0–3
1	4–7	1,5,9,13	2–3,10–11	3–5,15	4–7
2	8–11	2,6,10,14	4–5,12–13	6–8	8–11
3	12–15	3,7,11,15	6–7,14–15	9–11	12–15

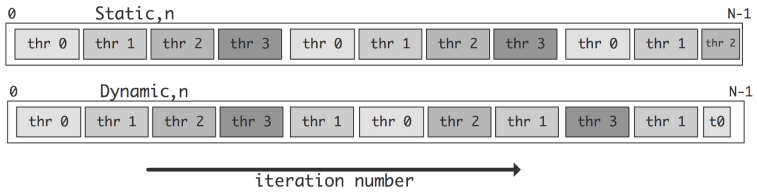
Exemple pour 16 itérations et 4 threads

Équilibrage de charge — Clause SCHEDULE

```
#pragma omp parallel for schedule(dynamic,X)
```

`schedule(dynamic,X)`
Les itérations sont groupées en blocs de taille **`X`** *(sauf peut-être pour le dernier)*.
Chaque thread exécute un bloc d'itération, et en demande un autre lorsqu'il a terminé.

`schedule(dynamic)`
`...` corresponds à **`schedule(dynamic,1)`**.
Chaque thread exécute une itération, et en demande une autre lorsqu'il a terminé.



Équilibrage de charge — Clause SCHEDULE

```
#pragma omp parallel for schedule(guided,X)
```

`schedule(guided,X)`
Les itérations sont groupées en blocs dont la taille décroît jusque **`X`** *(taille min)*.
La taille de correspond approx. au *nombre d'itérations restantes* divisé par le *nombre de threads*.
Chaque thread exécute un bloc d'itération, et en demande un autre lorsqu'il a terminé.

`schedule(guided)` corresponds à **`schedule(guided,1)`**

N° thread	Paquet	Taille du bloc	Iterations restantes
0	1–5000	5000	4999
1	5001–7500	2500	2499
1	7501–8750	1250	1249
1	8751–9375	625	624
0	9376–9687	312	312
1	9688–9843	156	156
0	9844–9921	78	78
1	9922–9960	39	39
1	9961–9980	20	19
1	9981–9990	10	9
1	9991–9995	5	4
0	9996–9997	2	2
1	9998–9998	1	1
0	9999–9999	1	0

*Exemple avec 2 threads et **`schedule(guided)`** pour des itérations équilibrées.*

Équilibrage de charge — Clause SCHEDULE

```
#pragma omp parallel for schedule(runtime)
```

`schedule(runtime)`
La répartition et la taille du bloc sont définies par une fonction de la librairie OpenMP ou par une variable d'environnement.

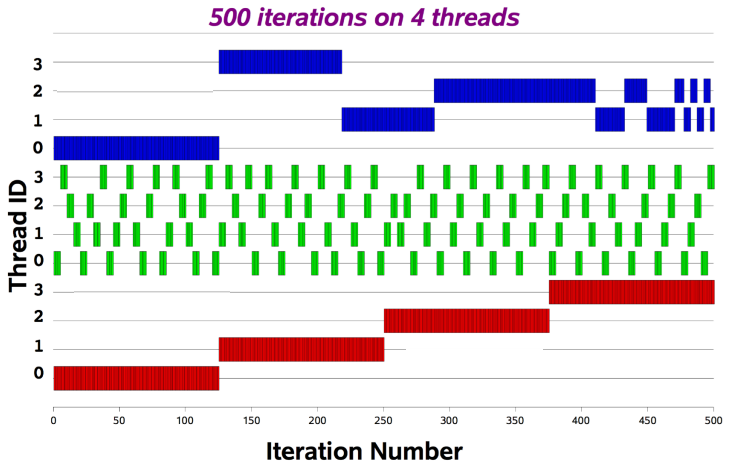
Fonction supplémentaire :

```
omp_set_schedule(omp_sched_static,1024);
omp_set_schedule(omp_sched_dynamic,156);
omp_set_schedule(omp_sched_guided,32);
```

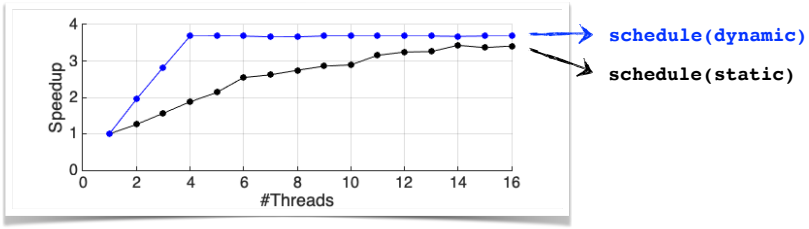
Variable d'environnement :

```
>> export OMP_SCHEDULE="STATIC"
>> export OMP_SCHEDULE="STATIC,1024"
>> export OMP_SCHEDULE="DYNAMIC,156"
>> export OMP_SCHEDULE="GUIDED"
>> export OMP_SCHEDULE="GUIDED,32"
```


Équilibrage de charge — Clause SCHEDULE



Étude d'un cas "mal équilibré" (suite)



Interprétation :

- **`schedule(dynamic)`** permet d'équilibrer la charge.

```
>> icpc -qopenmp balancing.cpp
>> export OMP_NUM_THREADS=16
>> export OMP_SCHEDULE='DYNAMIC'
>> ./a.out 20000
```

Résumé …

Ce qu'on veut …
Le meilleur **speedup.**

Ce qu'on peut faire …
Analyser les tâches qui peuvent être parallélisées.
Utiliser les **directives pour partager le travail** (**`parallel`**, **`sections`**, **`single`** et **`for`**)
Utiliser les **directives synchronisantes** uniquement quand c'est nécessaire.
Vérifier le **statut des variables** *(privées ou partagées)* et faire attention au **data race** !

Mais plus encore … **[Mode NINJA]**
Rendre des boucles parallélisables en modifiant les opérations
Combiner la **parallélisation** *(boucles les plus extérieures)*
et la **vectorisation** *(boucles les plus intérieures)*
Améliorer l'**équilibre de charge** (clause **`schedule`**)
Optimiser le nombre de threads

Tester des idées et des stratégies de programmation pour comprendre le processeur multi-cœur …

Résumé des commandes OpenMP vues au cours

Directives de compilation

Sentinelle	*Nom*	*Clauses*	
`#pragma omp`	`parallel`	`num_threads(…)` `default(none)` `private(…)` `firstprivate(…)` `shared(…)` `reduction(…:…)`	**Région parallèle (RP)** *+ Option pour définir le nombre de thread* *+ Options pour caractériser les variables* *+ Opération de réduction*
`#pragma omp`	`for`	`nowait` `schedule`	**Boucle parallèle dans une RP** *+ Option pour non-synchronisation* *+ Option pour répartission des itérations*
`#pragma omp`	`single`	`nowait`	**Exécution sur un thread dans une RP** *+ Option pour non-synchronisation*
`#pragma omp`	`sections` `(section)`	`nowait`	**Sections parallèles dans une RP** *+ Option pour non-synchronisation*
`#pragma omp`	`barrier`		**Synchronisation des threads**
`#pragma omp`	`atomic`		**Instruction effectuée par un thread à la fois**
`#pragma omp`	`critical`		**Bloc d'instructions effectué par un thread à la fois**

Résumé des commandes OpenMP vues au cours

Librairie et fonctions

```
#include <omp.h>
```

```
void omp_set_num_threads(int n);
int omp_get_num_threads();
int omp_get_thread_num();
void omp_set_schedule(…);
double omp_get_wtime();
```

Compilation et variables d'environ.

```
>> icpc -qopenmp myCode.cpp
>> g++ -fopenmp myCode.cpp
```

```
>> export OMP_NUM_THREADS=2
>> export OMP_SCHEDULE=…
>> ./a.out
```


Ressources

Documentation officiel

- Site Internet d'OpenMP
  http://www.openmp.org/specifications/

Cours en ligne

- PRACE Training Portal
  https://training.prace-ri.eu/
- Formation OpenMP de l'IDRIS
  http://www.idris.fr/formations/openmp/
- An Overview of OpenMP — Ruud van der Pas (Sun Microsystems)
  http://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf
- Cours d'introduction à OpenMP — Cédric Bastoul (U. de Strasbourg)
  http://icps.u-strasbg.fr/people/bastoul/public_html/teaching/openmp/bastoul_cours_openmp.pdf

Livres

- *An Introduction to Parallel Programming*
  P. Pachenco
- *Using OpenMP*
  B. Chapman, G. Jost, R. Van Der Pas