Programmation parallèle et ordonnancement de tâches par vol de travail

Thierry Gautier
thierry.gautier@**inrialpes.fr**
MOAIS, INRIA, Grenoble

LyonCalcul 24/02/2014

Plan

1. Contexte technologique des architectures multi-cœurs et many-cœurs
2. Programmation parallèle
3. Quelques environnement de programmation parallèle
4. Ordonnancement d'application parallèle
5. Vol de travail : implémentation et optimisation
6. Conclusion

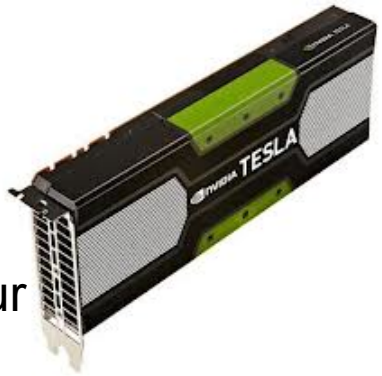
Un ordinateur (aujourd'hui)

processeur

+

mémoire

+

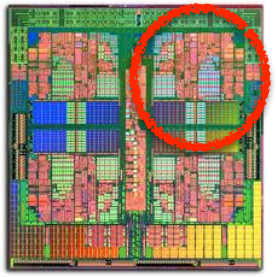


accélérateur

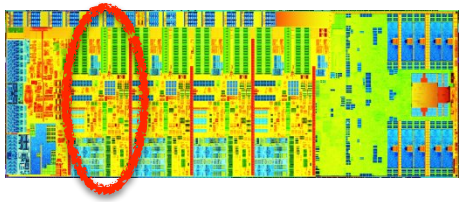
+

Zoom sur un processeur

- **Multi-cœurs**
- **Caches**

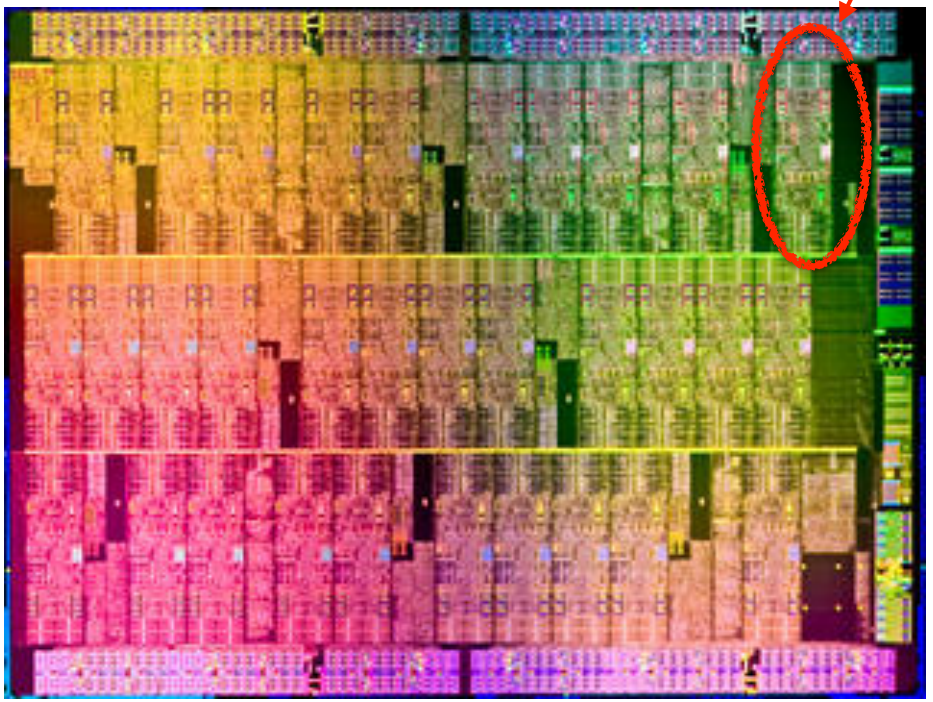


4-cœurs AMD Barcelona

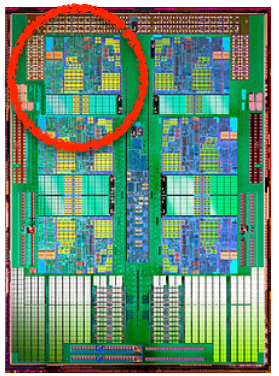


4-cœursHaswell

1 cœur

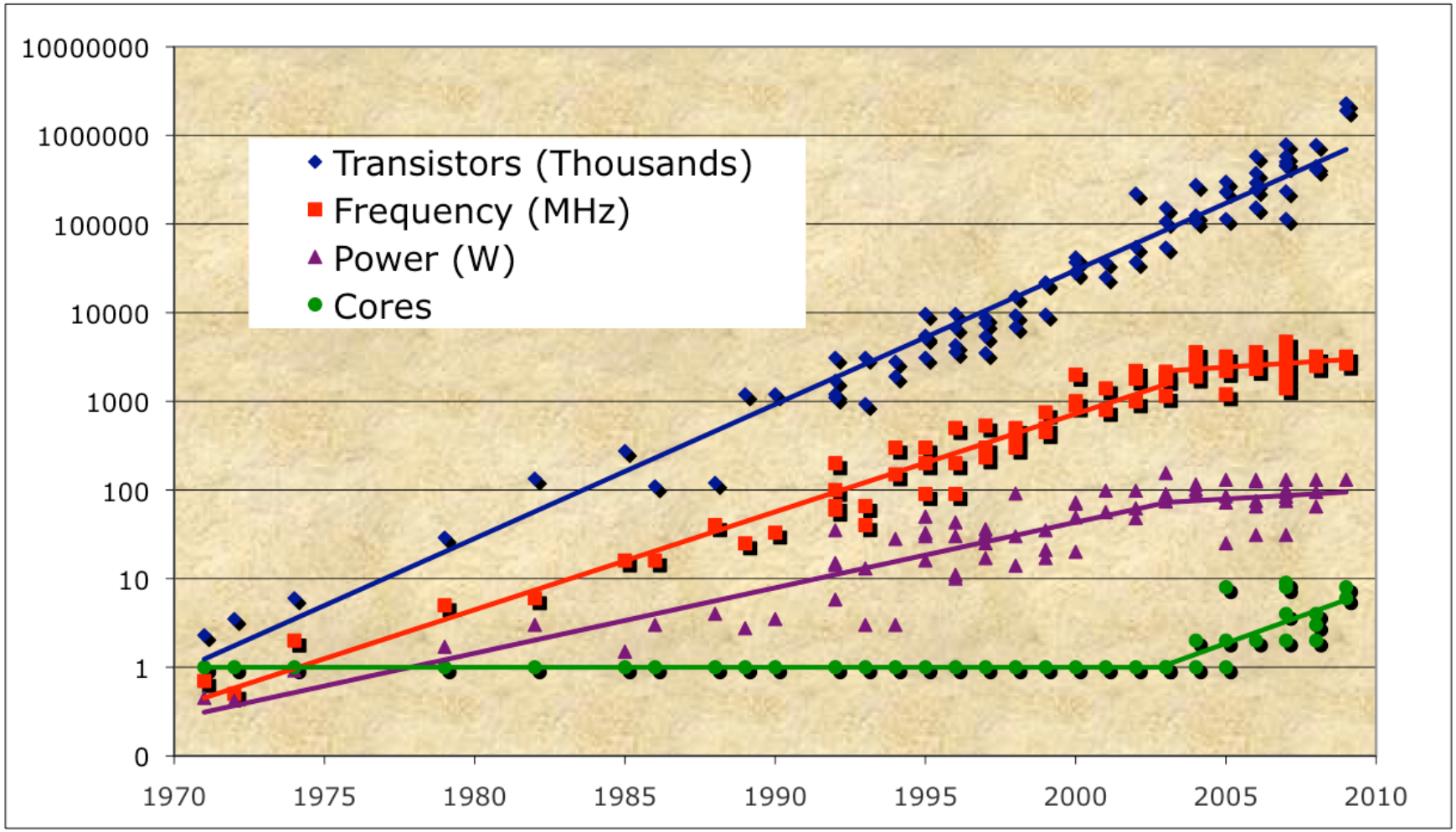


Intel Xeon Phi, 60 cœurs !



6-cœurs AMD Istanbul

Tendance

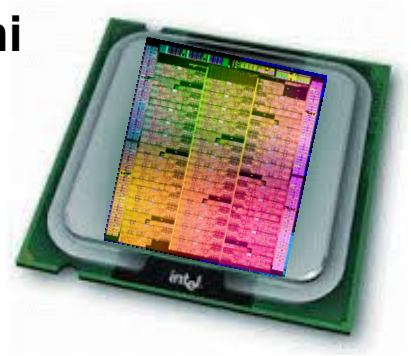


Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

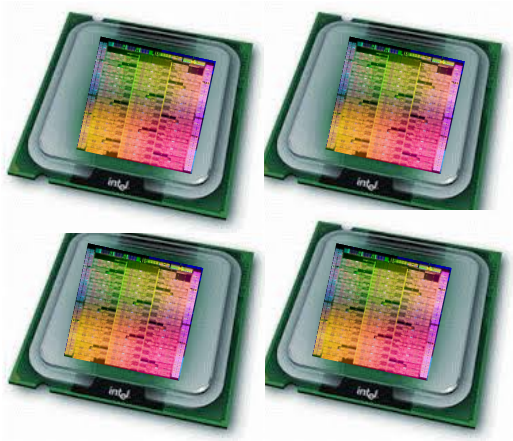
- **Accroissement du nombre de cœurs**
- **Hiérarchie de caches**
- **Architecture hybride, rapprochement accélérateur / cœur CPU**
  - **AMD Fusion, Nvidia**

Exemple d'une évolution annoncée

~ 2015
Version « socket » du Xeon Phi



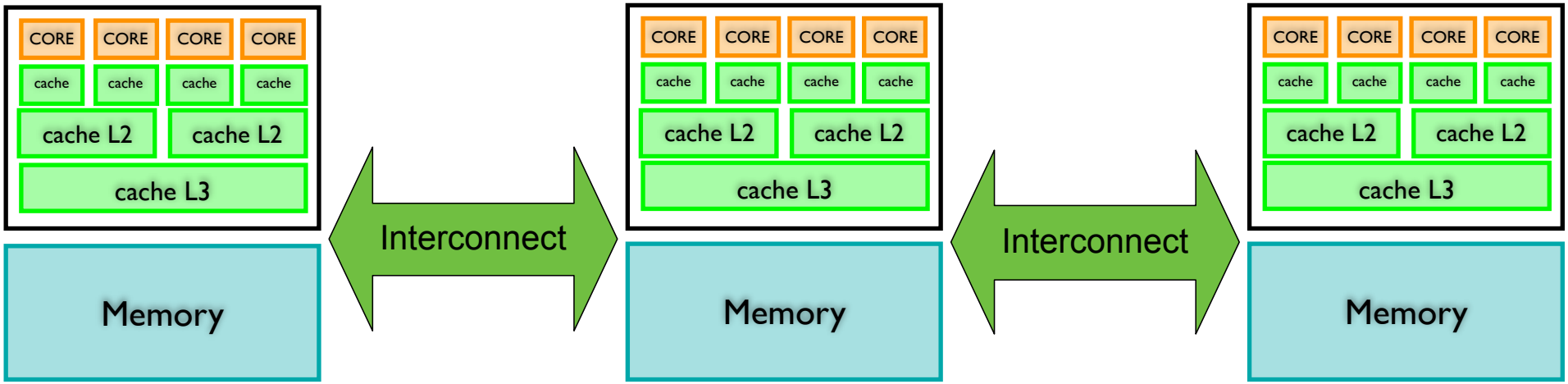
Suppression du « offloading »
- transfert de données hôte - Phi



- **Machine 4 sockets Intel Xeon Phi ?**
  - **~ 4 * 72 = 288 coeurs matériels**
  - **~ 4 * 3 Tflop/s**
  - **cohérence mémoire ?**

Attendons !

Caractéristiques des multicœurs

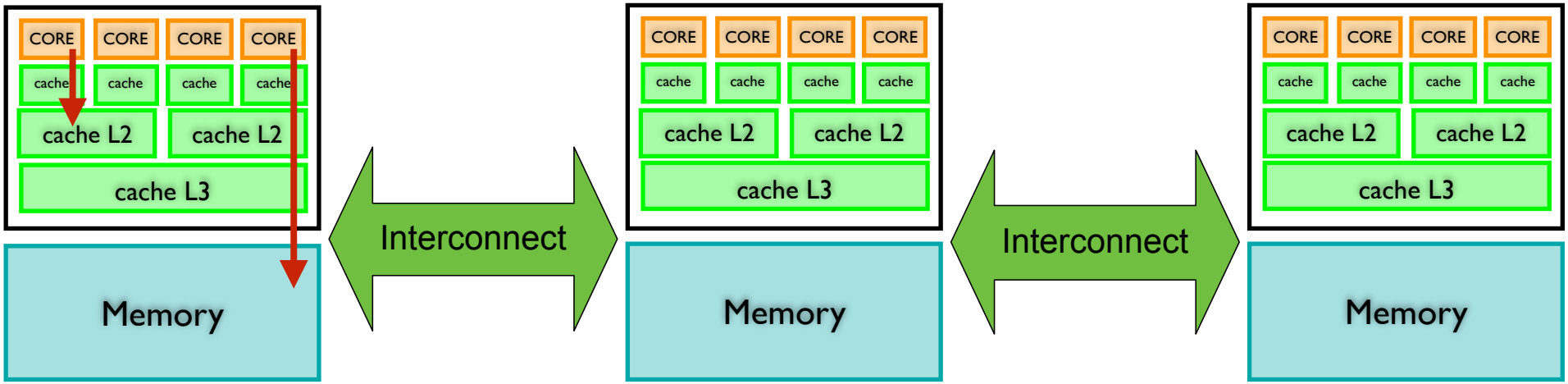


« nœud NUMA »

- **Architecture parallèle de type NUMA (Non Uniform Memory Access)**
  - **les mémoires importantes sont lentes, les mémoires petites rapides**
  - **les cœurs possèdent, ensemble, un grand cache rapide**

➡**Les algorithmes doivent faire travailler les processeurs sur des données locales**

Caractéristiques des multicœurs

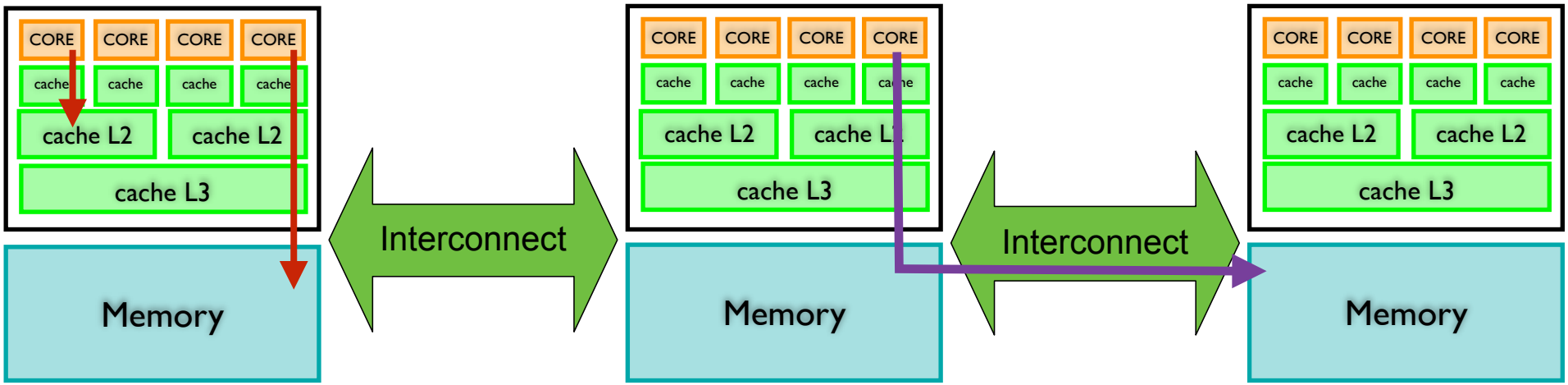


« nœud NUMA »

- **Architecture parallèle de type NUMA (Non Uniform Memory Access)**
  - **les mémoires importantes sont lentes, les mémoires petites rapides**
  - **les cœurs possèdent, ensemble, un grand cache rapide**

➡**Les algorithmes doivent faire travailler les processeurs sur des données locales**

Caractéristiques des multicœurs

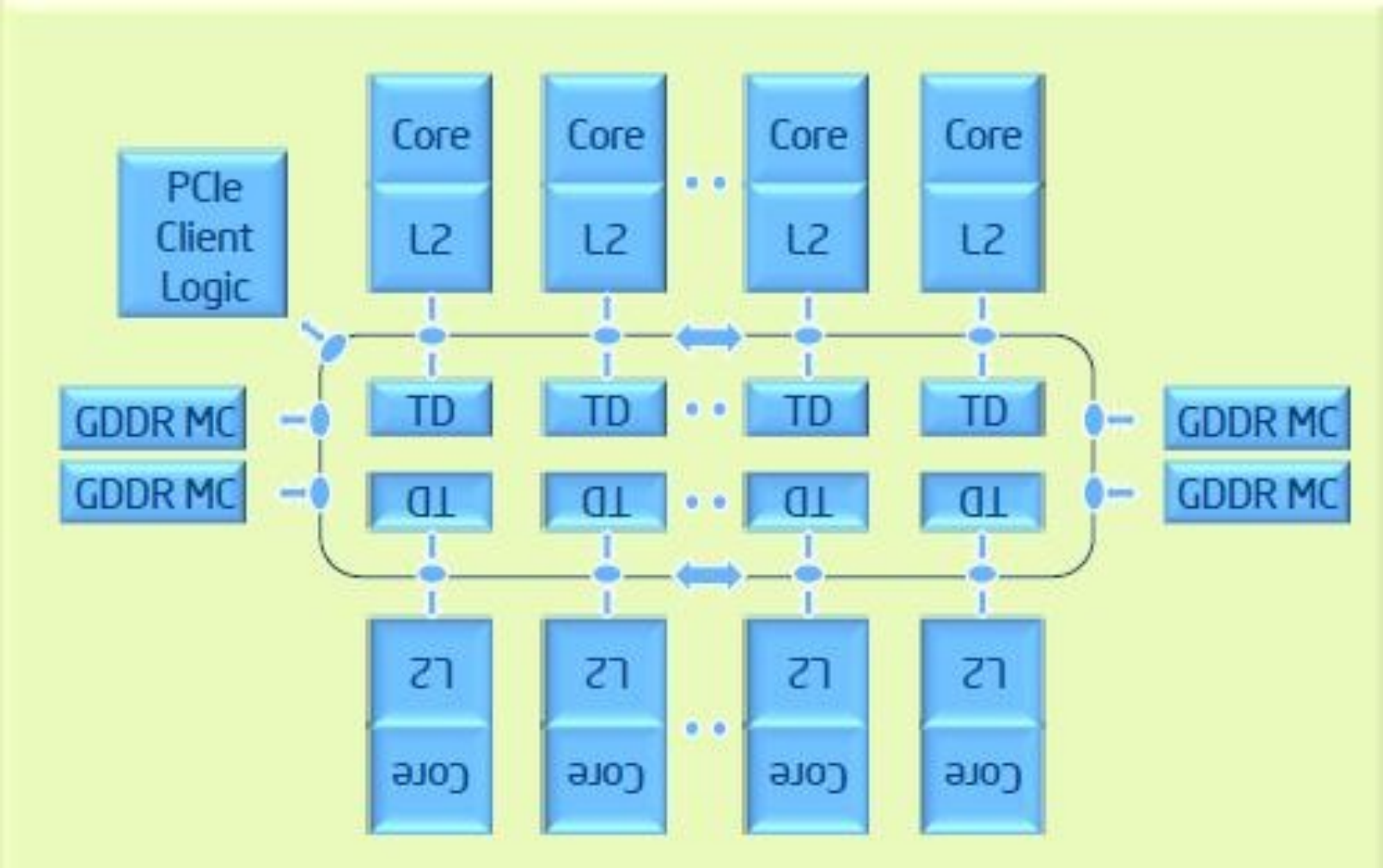


« nœud NUMA »

- **Architecture parallèle de type NUMA (Non Uniform Memory Access)**
  - **les mémoires importantes sont lentes, les mémoires petites rapides**
  - **les cœurs possèdent, ensemble, un grand cache rapide**

➡**Les algorithmes doivent faire travailler les processeurs sur des données locales**

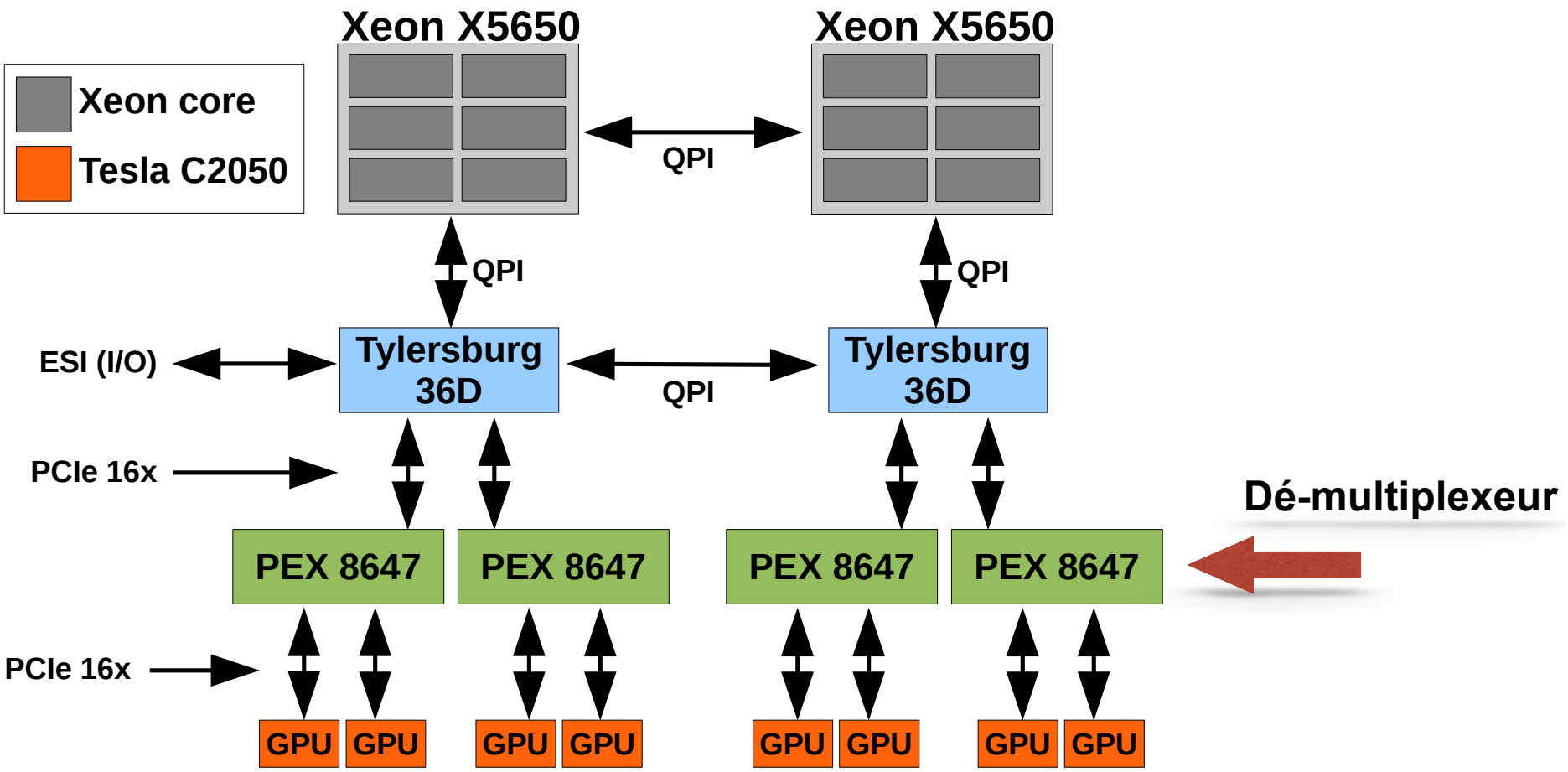
Architecture de l'Intel Xeon Phi



- Jusqu'à 61 cœurs
- Caches L2 cohérents entre les cœurs
- 1TFlop/s double precision

Une architecture multi-cœur hybride

- 8 GPUs et 12 CPUs



2. Comment programmer ces architectures ?

- ✓ - restriction aux multi-cœurs et many-cœurs Xeon Phi
- ✓ - mardi 14h-16: exposé sur la programmation des GPUs

Différents niveaux de parallélisme

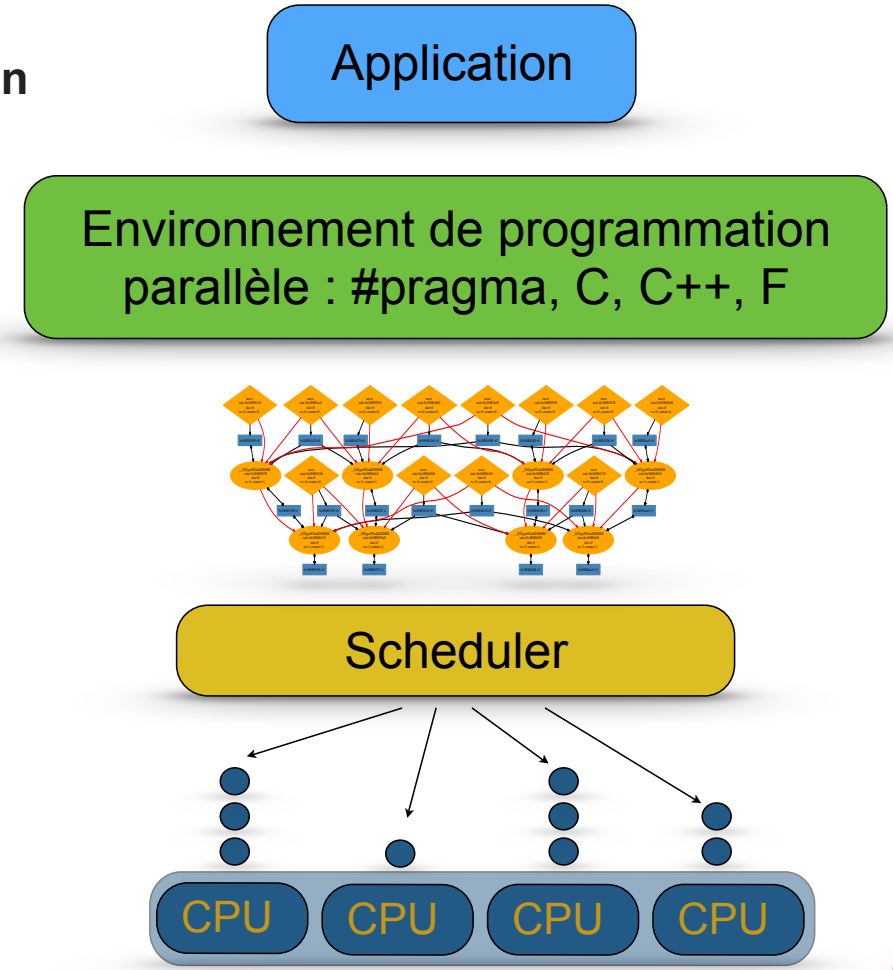
- **Des super-calculateurs vers les unités SIMD**
  - **réseau**
    - MPI, PGAS (UPC), X10
  - **multi-cœurs**
    - OpenMP, X10, Cilk, Intel TBB, XKaapi,…
  - **accélérateurs**
    - Cuda, OpenCL, OpenACC, OpenMP (4.0)
  - **vecteur / unité SIMD**
    - compilateur, type étendu, OpenMP (4.0)

- **Quel environnement de programmation parallèle ?**
  - **De 2 à 4 environnements différents…**

Objectif : la portabilité des performances!

- **Garantir des performances d'une application**
  - **sur plusieurs architectures !**
  - **avec des variations de charge en cours d'exécution**
    - OS jitter, charge variable de l'application

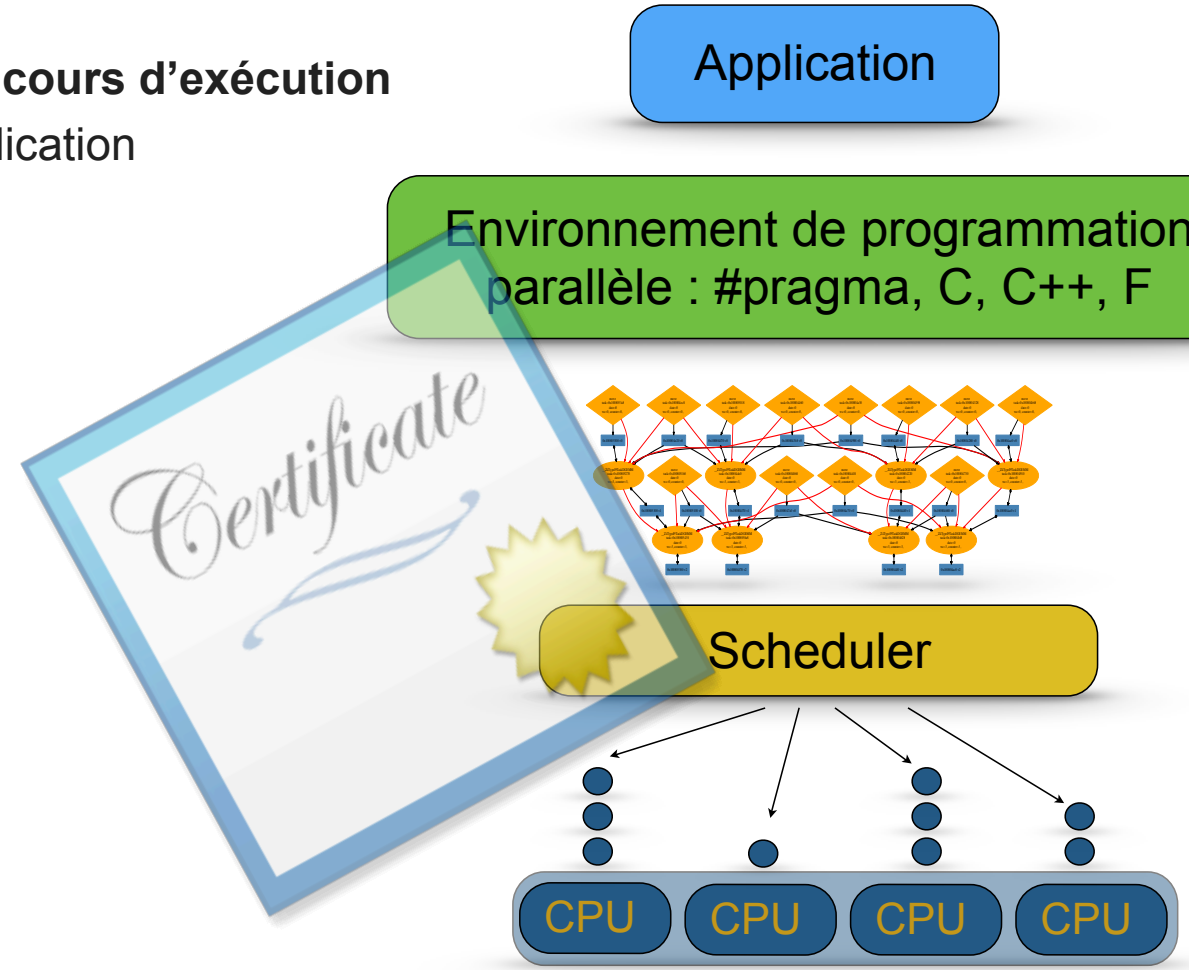
- **Une solution en deux étapes**
  1. **parallélisation de l'application**
     - Attention aux communications
  2. **ordonnancement efficace**



Objectif : la portabilité des performances!

- **Garantir des performances d'une application**
  - **sur plusieurs architectures !**
  - **avec des variations de charge en cours d'exécution**
    - OS jitter, charge variable de l'application

- **Une solution en deux étapes**
  1. **parallélisation de l'application**
     - Attention aux communications
  2. **ordonnancement efficace**



Comment programmer ces architectures?

1. Parallélisation de l'algorithme de calcul

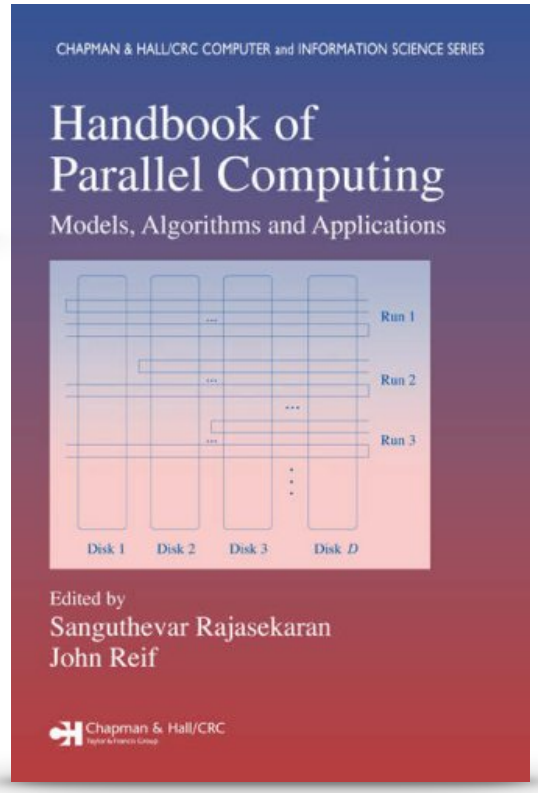
- **décomposition du problème en sous problèmes indépendants**
- **un « bon algorithme parallèle »**
  - efficace (~ même nombre d'opérations qu'en séquentiel)
  - très parallèle afin d'exploiter tout le parallélisme disponible

• Loi d'Amdahl

- **s: fraction du programme exécuté en séquentiel**
- **(1-s): fraction du programme exécuté en parallèle**
- **p nombre de cœurs**
- **Alors le speedup est borné par :** $Speedup(p) = \frac{T_{séquentiel}}{T_p} \leq \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$

Comment programmer ces architectures?

1. **Parallélisation de l'algorithme de calcul**
   - **décomposition du problème en sous problèmes indépendants**
   - **un « bon algorithme parallèle »**
     - efficace (~ même nombre d'opérations qu'en séquentiel)
     - très parallèle afin d'exploiter tout le parallélisme disponible



- **Loi d'Amdahl**
  - **s: fraction du programme exécuté en séquentiel**
  - **(1-s): fraction du programme exécuté en parallèle**
  - **p nombre de cœurs**
  - **Alors le speedup est borné par :** $Speedup(p) = \frac{T_{séquentiel}}{T_p} \leq \frac{1}{s + \frac{1-s}{p}} \leq \frac{1}{s}$

Comment programmer ces architectures?

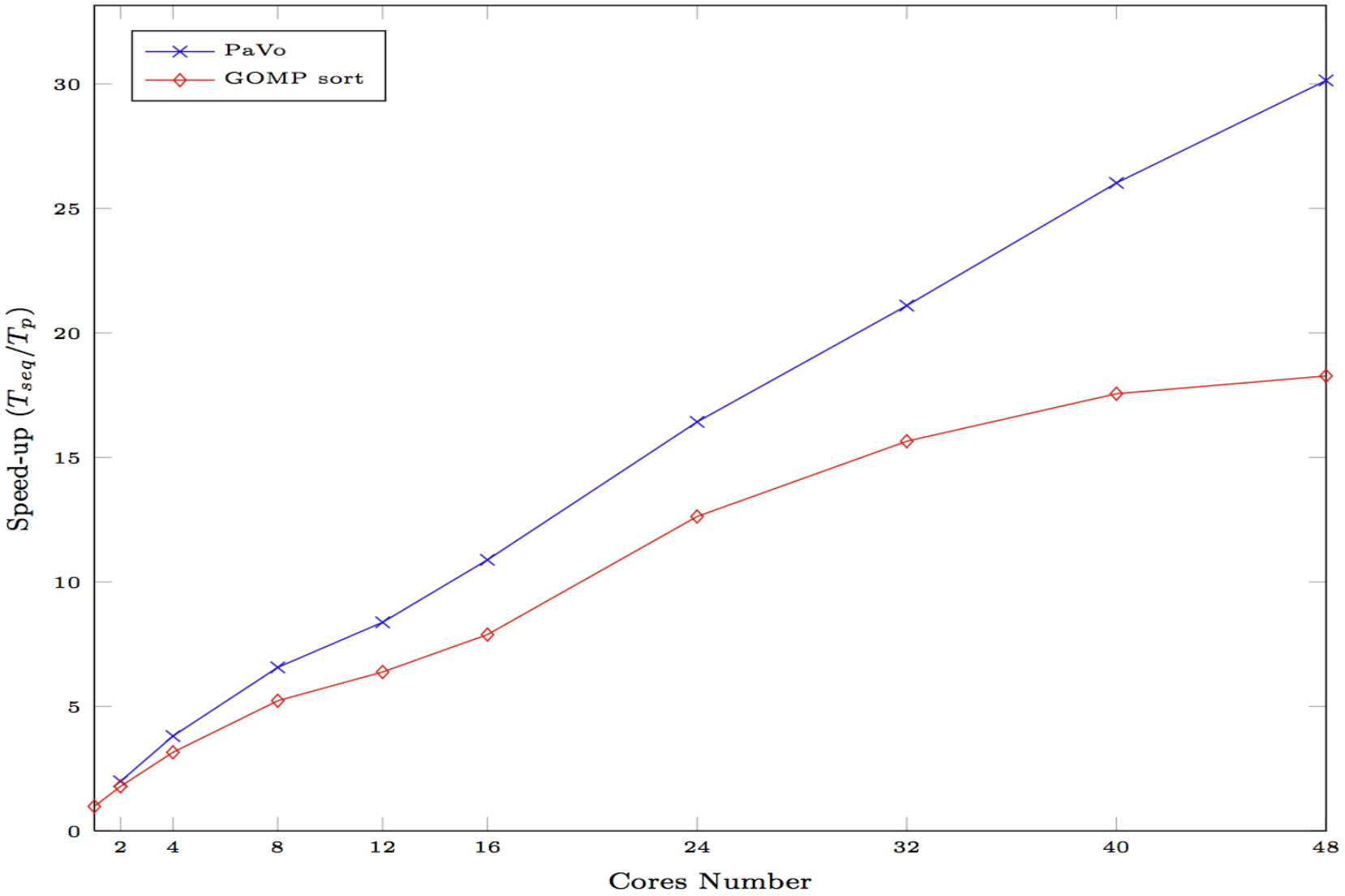
1.(suite) ATTENTION aux communications !

- **communications =**
  - envoi de message d'un cœur à un autre
  - lecture ou écriture à distance d'une données non dans le cache, …
  - faux partage, …

- **outils du système pour la gestion de la localité**
  - contrôle du placement des tâches sur les ressources
    - « thread affinity », GOMP_CPU_AFFINITY (OpenMP/GCC), KMP_AFFINITY(Intel OpenMP), hwloc-bind, numactl - - physcpubin
  - et des données en mémoire
    - libnuma : numa_alloc_on_node, numactl - - membind, hwloc-bind

- **outils algorithmiques (ex. *communication avoiding algorithms*)**
  - homepage de James Demmel : http://www.cs.berkeley.edu/~demmel/

Tri de séquences presque triées



Communication de M.Durand

- AMD Magny-Cours 48 cœurs
- PaVo = un algorithme spécialisé tenant compte du caractère presque trié

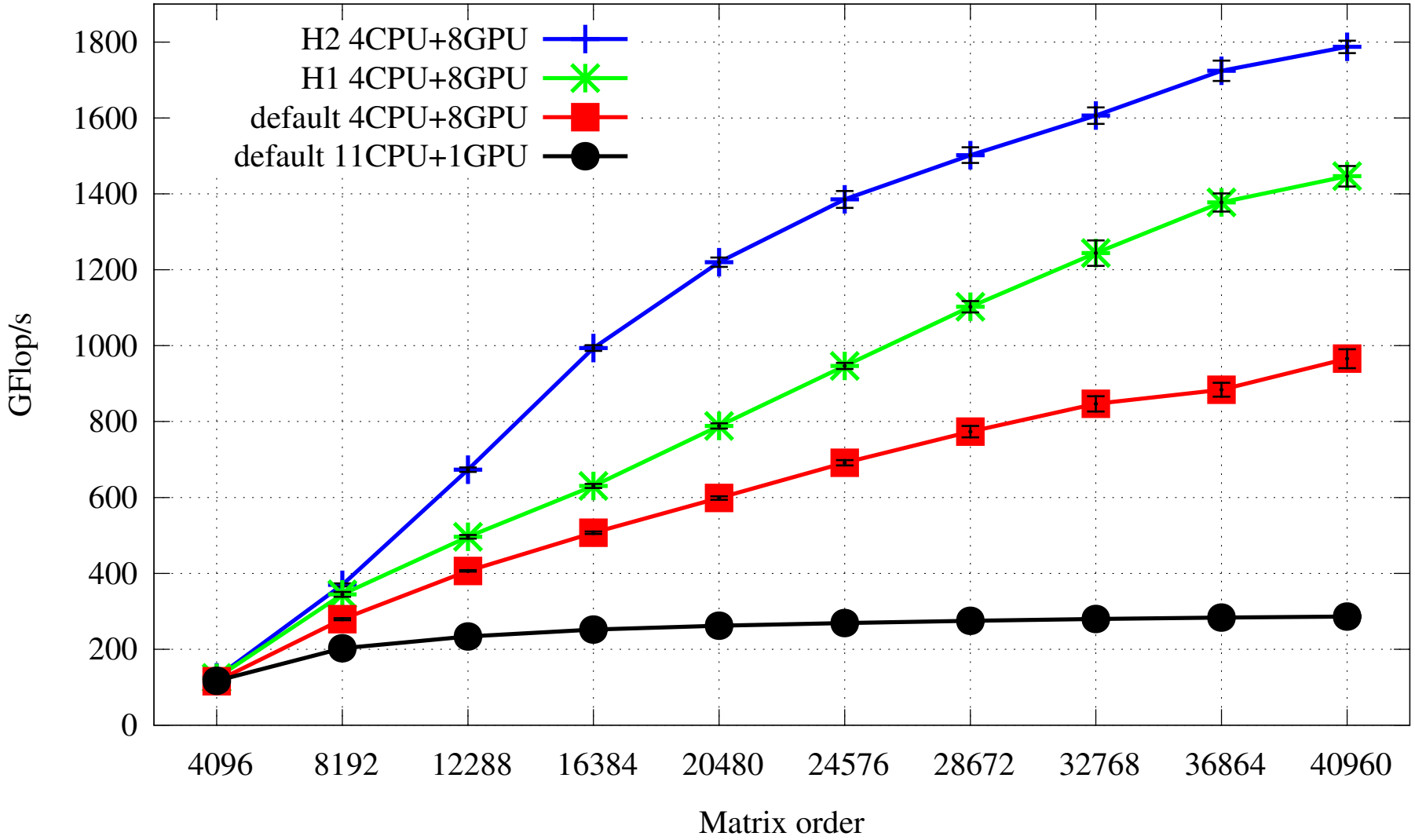
Comment exécuter ces programmes ?

2.Réguler la charge de travail sur les cœurs

- **problème d'ordonnancement souvent très compliqué !**
  - nécessite un modèle d'application : classiquement un graphe de tâche
  - un modèle de performance : connaître le temps d'exécution d'une tâche sur une ressource
  - un algorithme qui calcule une date et un site d'exécution pour chaque tâche afin d'optimiser une fonction objectif (temps d'exécution)

- **problème plus ou moins entaché d'« erreurs » dû à la possible variation entre :**
  - les temps d'exécution prévus lors du découpage de l'application
  - et les temps d'exécution réels mesurés en cours d'exécution

Exemple de 3 ordonnancements



[IPDPS 2013] DPOTRF (Cholesky), architecture hybride idgraf

- **default = standard work stealing**
- **H1, H2 = 2 heuristiques de prise en compte de la localité des données**

3.Environnement de programmation parallèle

Un environnement = des architectures

	OpenMP	Intel TBB	Cilk	CUDA	OpenCL	MPI
Muticœurs	*yes*	*yes*	*yes*	*no*	*yes*	*yes*
Xeon Phi	*yes*	*yes*	*yes*	*no*	*yes*	*yes*
GPU Nvidia	*OMP-4.0*	*no*	*no*	*yes*	*yes*	*no*
GPU AMD	*OMP-4.0*	*no*	*no*	*no*	*yes*	*no*
Mémoire Distribué	*no*	*no*	*no*	*no*	*no*	*yes*

- (1) Extension OpenMP-4.0 pour les accélérateurs
- (2) Possible utilisation excessive de la mémoire

- **Intel TBB : cet après midi 14h-16h, Thierry Dumont**
- **Programmation sur GPU : mardi 25/02, 14h-16h**

- **OpenMP (4.0)**
  - **visent à exploiter tous les niveaux jusqu'au multi-cœur (comme OpenCL)**
  - **exemples futurs en OpenMP**
- **… Sans GPU moins de problème de programmation !**

- **Portabilité du code. Portabilité des performances ?**

Parallélisme de données (data parallelism)

- **Distribution de données et placement des opérations là où se trouve les données**
  - **typiquement traitement par des boucles**

```
#pragma omp for shared(A,B)
for (size_t i=0; i<N; ++i)
  A[i] = do_computation(B[i]);
```

OpenMP

```
tbb::parallel_for( tbb::blocked_range<size_t>(0,n),
    [&](const tbb::blocked_range<size_t>& r) {
      for(size_t i=r.begin(); i!=r.end(); ++i)
        A[i] = do_computation( B[i] );
      }
);
```

Intel TBB

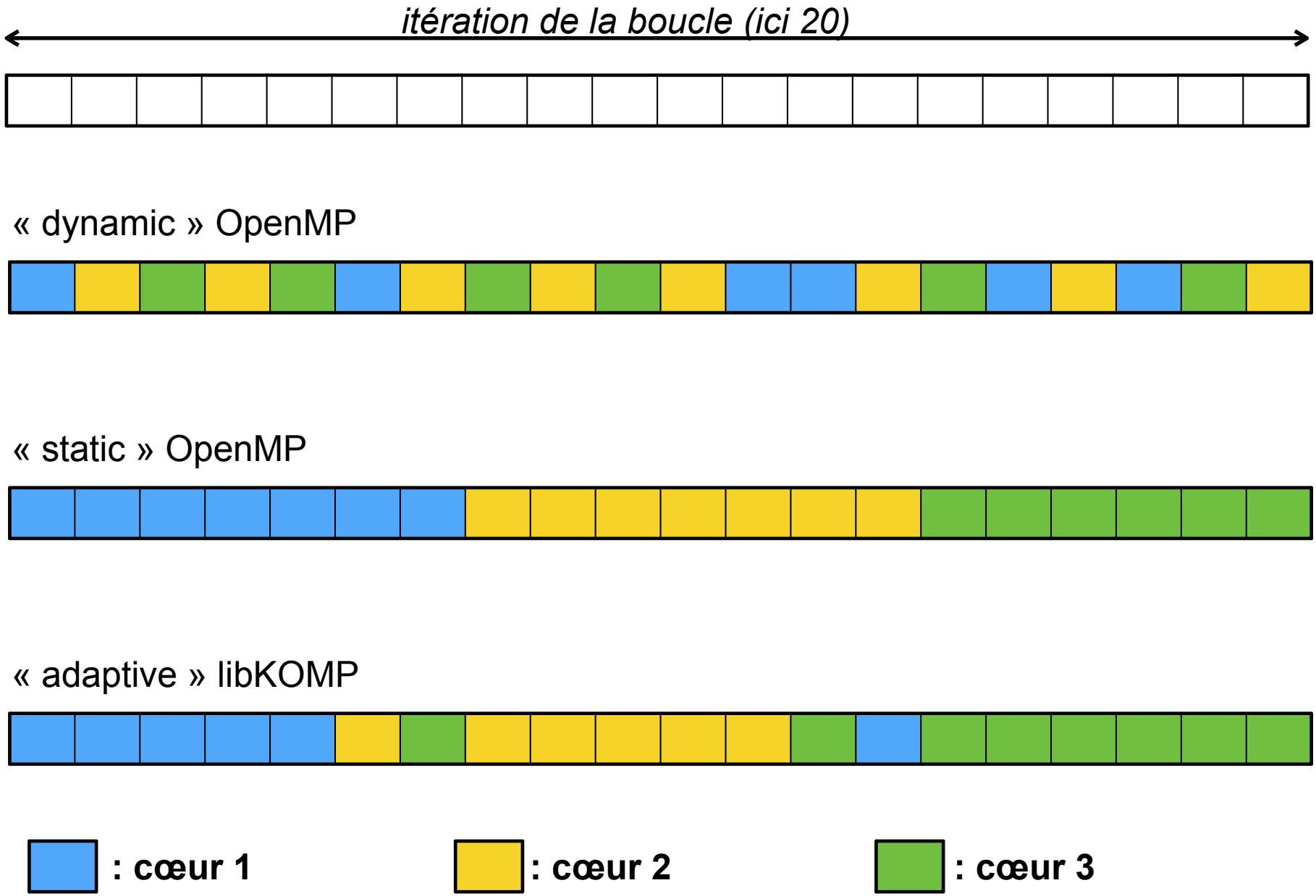
```
cilk_for (size_t i=0; i<N; ++i)
  A[i] = do_computation(B[i]);
```

Cilk

  - **distribution de l'espace d'itération**
    - statique : chaque thread reçoit a peu près la même quantité à traiter avant exécution
    - dynamique : l'espace d'itération est distribué en fonction de l'activité des threads

distribution	OpenMP	Intel TBB	Cilk
statique	*yes*	*no*	*no*
dynamique	*yes*	*yes*	*yes*

Exemples de distribution



Parallélisme de tâche (task parallelism)

- Un programme parallèle est divisé en plusieurs tâches
- Tâches indépendantes

```
void fibonacci( int n, int *result )
{
  if (n <2) *result = n;
  else {
    int r1, r2;
    #pragma omp task shared(r1)
    r1 = fibonacci(n-1);
    #pragma omp task shared(r2)
    r2 = fibonacci(n-2);
    #pragma omp taskwait
    *result = r1+r2;
  }
}
```

OpenMP

```
void fibonacci( int n, int *result )
{
  if (n <2) *result = n;
  else {
    int r1, r2;
    r1 = cilk_spawn fibonacci(n-1);
    r2 = cilk_spawn fibonacci(n-2);
    cilk_sync;
    *result = r1+r2;
  }
}
```

Cilk

```
printf("Cet après midi de 14h à 16h !\n »);
```

TBB

- Tâches dépendantes

```
void fibonacci( int n, int* result )
{
  if (n <2) return n;
  int r1, r2;
  #pragma omp task shared(r1) depend(out: r1)
  fibonacci(n-1, &r1);
  #pragma omp task shared(r2) depend(out: r2)
  fibonacci(n-2, &r2);
  #pragma omp task shared(r1,r2) depend(in: r1, r2) depend(out: result)
  *result = r1+r2;
  #pragma omp taskwait
}
```

OpenMP-4.0

Produit de matrice en OpenMP-4.0

```
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N] )
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
        #pragma omp task depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
                          depend ( inout: C[i:BS][j:BS] )
        for (ii = i; ii < i+BS; ii++ )
          for (jj = j; jj < j+BS; jj++ )
            for (kk = k; kk < k+BS; kk++ )
              C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
      }
    }
  }
}
```

C++

OpenMP-4.0, example 16.19c

Fortran

```
subroutine matmul_depend (N, BS, A, B, C)
   integer :: N, BS, BM
   real, dimension(N, N) :: A, B, C
   integer :: i, j, k, ii, jj, kk
   BM = BS -1
   do i = 1, N, BS
      do j = 1, N, BS
         do k = 1, N, BS
!$omp task depend ( in: A(i:i+BM, k:k+BM), B(k:k+BM, j:j+BM) ) &
!$omp depend ( inout: C(i:i+BM, j:j+BM) )
            do ii = i, i+BS
               do jj = j, j+BS
                  do kk = k, k+BS
                     C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
                  end do
               end do
            end do
!$omp end task
         end do
      end do
   end do
end subroutine
```

OpenMP-4.0, example 16.19f

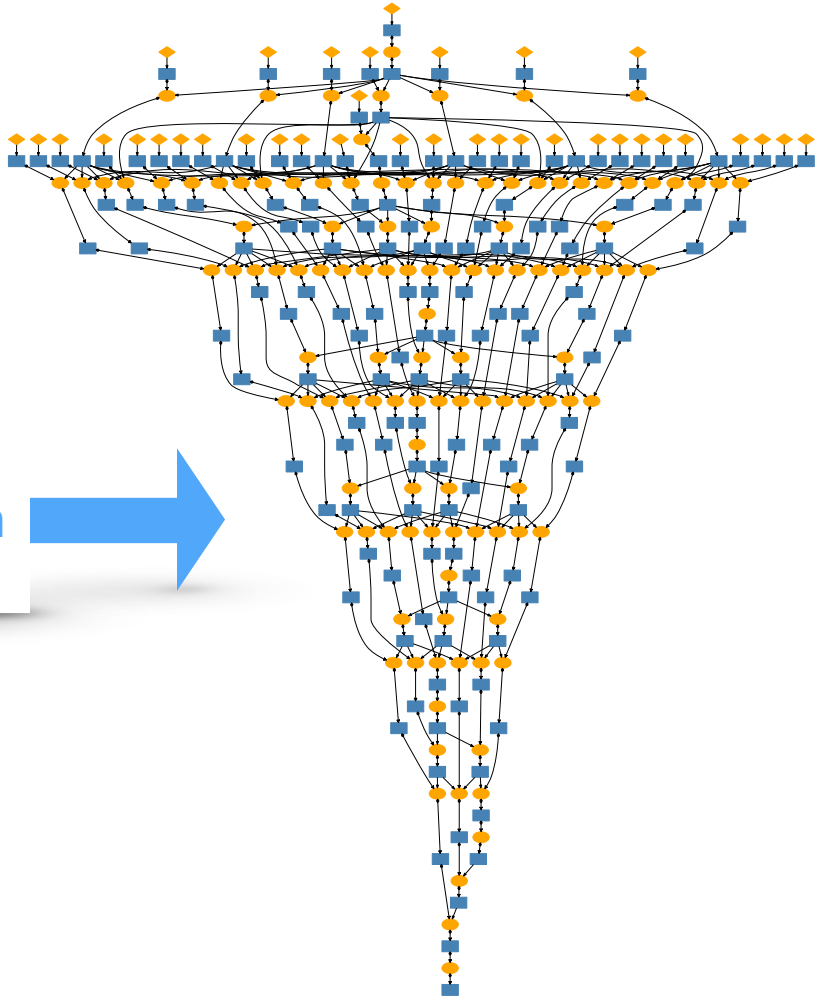
Factorisation de Cholesky

```
#include <cblas.h>
#include <clapack.h>

void Cholesky( int N, double A[N][N], size_t NB )
{
  for (size_t k=0; k < N; k += NB)
  {
#pragma omp task depend(inout: A[k:NB][k:NB]) shared(A)
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[k:NB][k:NB]) \
                   depend(inout: A[m:NB][k:NB]) shared(A)
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[m:NB][k:NB]) \
                   depend(inout: A[m:NB][m:NB]) shared(A)
     cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma kaapi task depend(in: A[m:NB][k:NB], A[n:NB][k:NB])\
                   depend(inout: A[m:NB][n:NB]) shared(A)
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
  }
}
```

Factorisation de Cholesky

```
#include <cblas.h>
#include <clapack.h>

void Cholesky( int N, double A[N][N], size_t NB )
{
  for (size_t k=0; k < N; k += NB)
  {
#pragma omp task depend(inout: A[k:NB][k:NB]) shared(A)
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[k:NB][k:NB]
                  depend(inout: A[m:NB][k:
      cblas_dtrsm ( CblasRowMajor, CblasLeft, Cblas
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[m:NB][k:NB]) \
                  depend(inout: A[m:NB][m:NB]) shared(A)
     cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma kaapi task depend(in: A[m:NB][k:NB], A[n:NB][k:NB])\
                  depend(inout: A[m:NB][n:NB]) shared(A)
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
  }
}
```

A l'exécution



Résumé

	OpenMP	Intel TBB	Cilk
Boucle parallèle	*yes*	*yes*	*yes*
Tâches indépendantes	*yes*	*yes*	*yes*
Tâches dépendantes	*OMP-4.0*	*no*	*no*
Langage	*C,C++,Fortran*	*C++*	*C,C++*

- **(1) Extension OpenMP-4.0 pour les tâches dépendantes**
  - spécification certainement légèrement modifiée dans la version 4.1

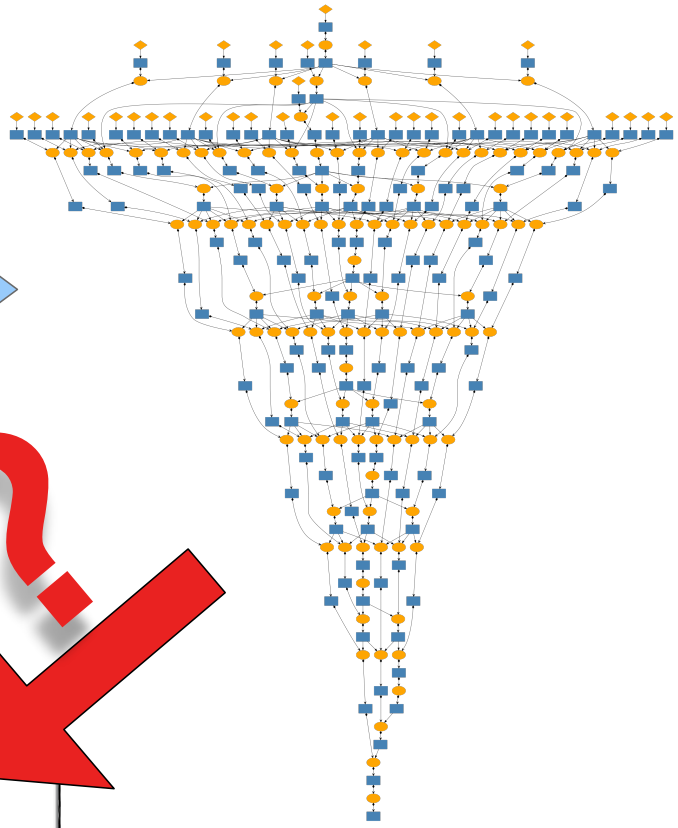
- **Portabilité du code sur quelques architectures**
- **Portabilité des performances ?**

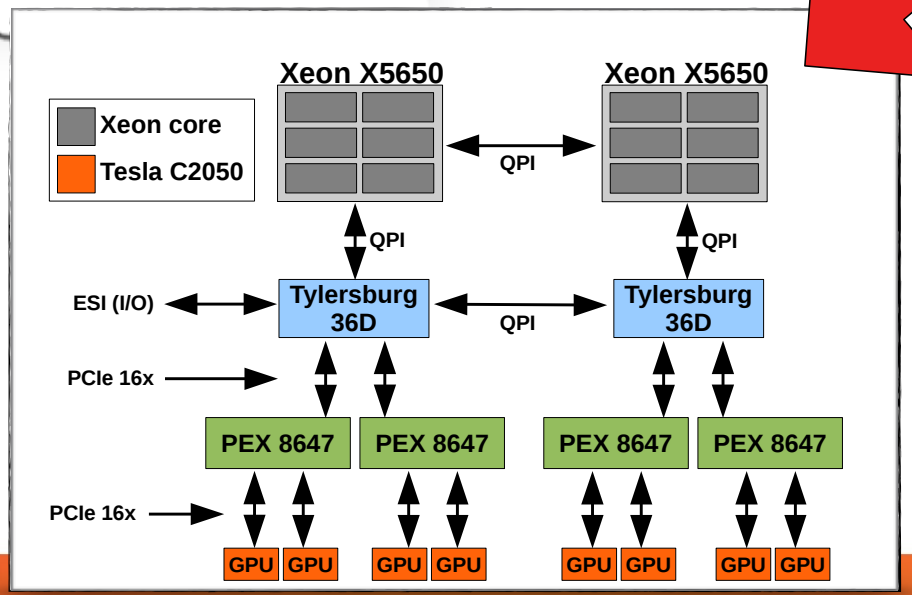
4.Ordonnancement d’application parallèle

- problématique
- algorithme d’ordonnancement de liste
- HEFT
- Vol de travail

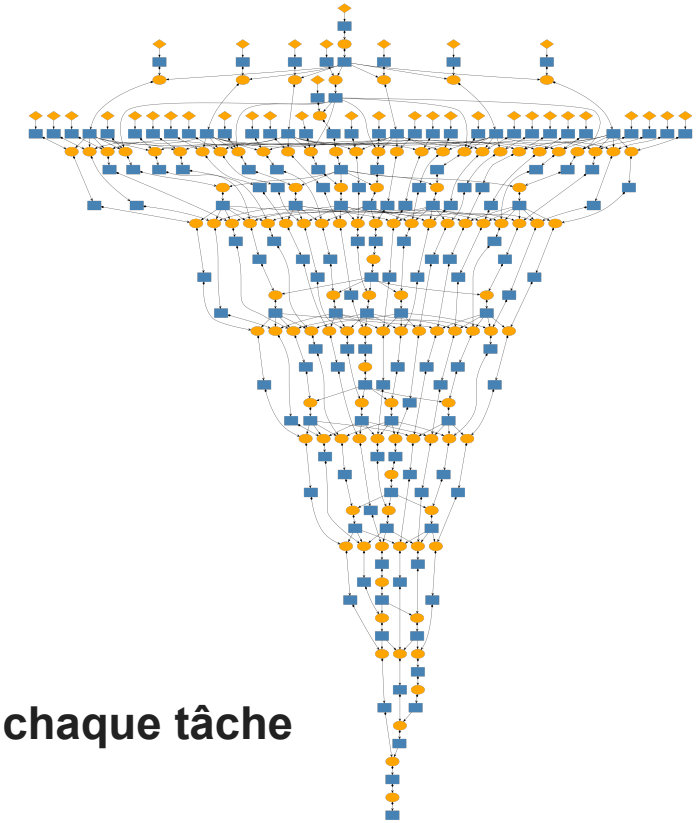
Problématique

```
#include <cblas.h>
#include <clapack.h>

void Cholesky( int N, double A[N][N], size_t NB )
{
  for (size_t k=0; k < N; k += NB)
  {
#pragma omp task depend(inout: A[k:NB][k:NB]) shared(A)
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[k:NB][k:NB]) \
                   depend(inout: A[m:NB][k:NB]) shared(A)
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task depend(in: A[m:NB][k:NB]) \
                   depend(inout: A[m:NB][m:NB]) shared(A)
     cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma kaapi task depend(in: A[m:NB][k:NB], A[n:NB][k:NB])\
                   depend(inout: A[m:NB][n:NB]) shared(A)
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
  }
}
```





Ordonnancement



- **Une application parallèle = un graphe de tâches**
  - **orienté (avec dépendances)**
  - **sans cycle**

- **L'ordonnancement consiste à**
  - **décider d'une ressources et d'une date d'exécution pour chaque tâche**

- **La valeur de l'ordonnancement est la valeur de la fonction objectif choisie**
  - **temps d'exécution (makespan) noté** $C_{\max}$
  - **temps d'exécution optimal noté** $C^*_{\max}$

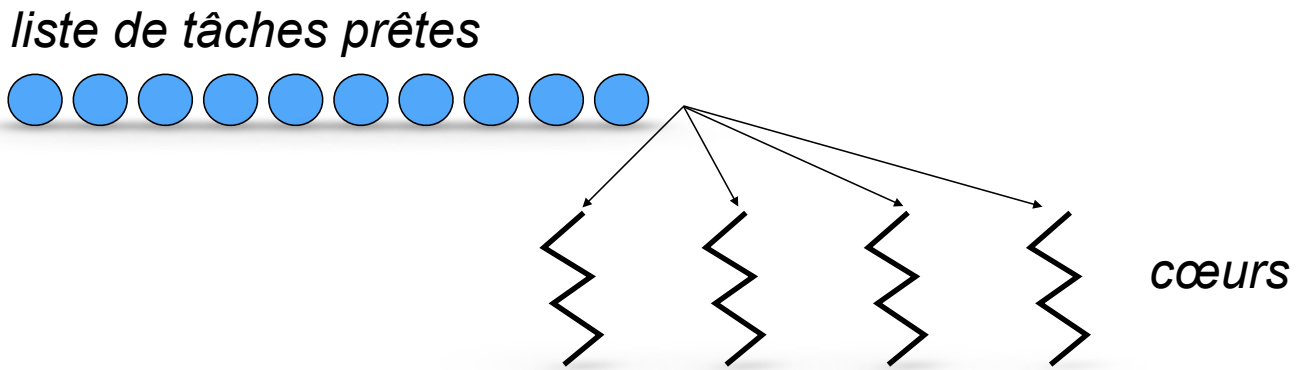
Algorithmes d'ordonnancement

- **Gestion de listes de tâches prêtes**
  - **une ou plusieurs liste de tâches prêtes**
  - **les tâches qui s'exécutent rendent prêtes leurs successeurs**

- **Eventuellement un modèle de performance** ⟵ **Optionnel !**
  - **informations sur l'application**
    - connaissance ou non des temps d'exécution des tâches sur les ressources
    - connaissance ou non des volumes de données nécessaire pour exécuter les tâches
  - **informations sur le matériel**
    - modèle du temps de transfert d'un volume de données

- **La garantie de performance d'un algorithme d'ordonnancement**
  - **repose sur la *qualité d'une solution***
    - ratio (≥ 1) entre la valeur de l'ordonnancement calculé et la meilleure solution possible sur la même instance
  - **garantie de performance ρ = le maximum des ratios pour toutes les instances**

Exemple : un algorithme de liste



- **Idée simple**
  - **un cœur inactif prend une tâche dans la liste des tâches prêtes et l'exécute**
    - l'exécution d'une tâche peut rendre prêtes les successeurs de la tâche
  - **une liste pour tous les cœurs**

- **[Coffman-Graham 72]**
  - **la garantie de performance sur p cœurs de cet algorithme est** $\rho = 2 - \frac{1}{p}$
    - soit $C_{\max} \leq (2 - \frac{1}{p}) C^*_{\max}$

HEFT

- **Heterogeneous Earliest Finish Time**

- **Pour ressources hétérogène : architecture hybride**
  - **utilise un modèle de performance**
    - on suppose connu, pour chaque tâche, ses temps d'exécution sur CPU et GPU
    - on suppose connu le temps nécessaire pour ramener les données avant d'exécuter la tâche

- **Principe**
  - **on trie la liste de tâches prêtes en fonction d'un critère « rang »**

$$Rang_i = \frac{T_i^{CPU} + T_i^{GPU}}{2} + \max_{j \in succ(i)} (Comm_{ij} + Rang_j)$$

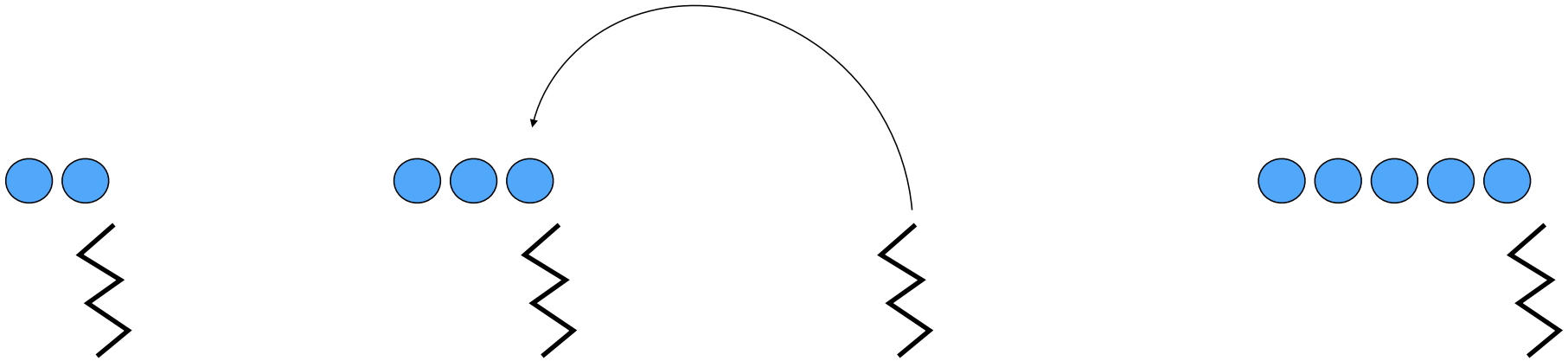
  - **et on place la première tâche de la liste sur la ressource où elle termine le plus tôt, y compris le coût de la communication**

- **Propriétés**
  - **faible complexité**
  - **mais pas de garantie de performance** ⟸ **mais semble bien marcher en pratique !**

Vol de travail

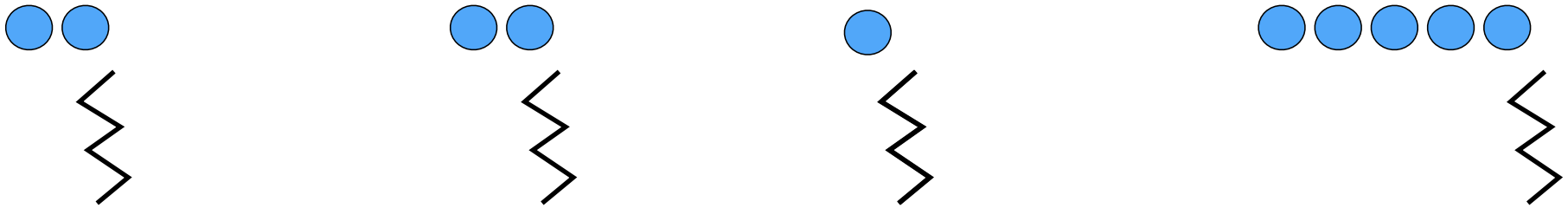
- **Pour ressources homogènes ou hybrides**
- **Principe**
  - **chaque cœur possède sa liste de tâches prêtes**
    - l'exécution d'une tâche peut rendre prêtes les successeurs de la tâche
  - **lorsqu'un cœur est inactif il cherche à voler une tâche prêtes chez un autre cœur**
    - choix aléatoire de la victime
- **Le vol = communication**

Vol de travail



- **Pour ressources homogènes ou hybrides**
- **Principe**
  - **chaque cœur possède sa liste de tâches prêtes**
    - l'exécution d'une tâche peut rendre prêtes les successeurs de la tâche
  - **lorsqu'un cœur est inactif il cherche à voler une tâche prêtes chez un autre cœur**
    - choix aléatoire de la victime
- **Le vol = communication**

Vol de travail

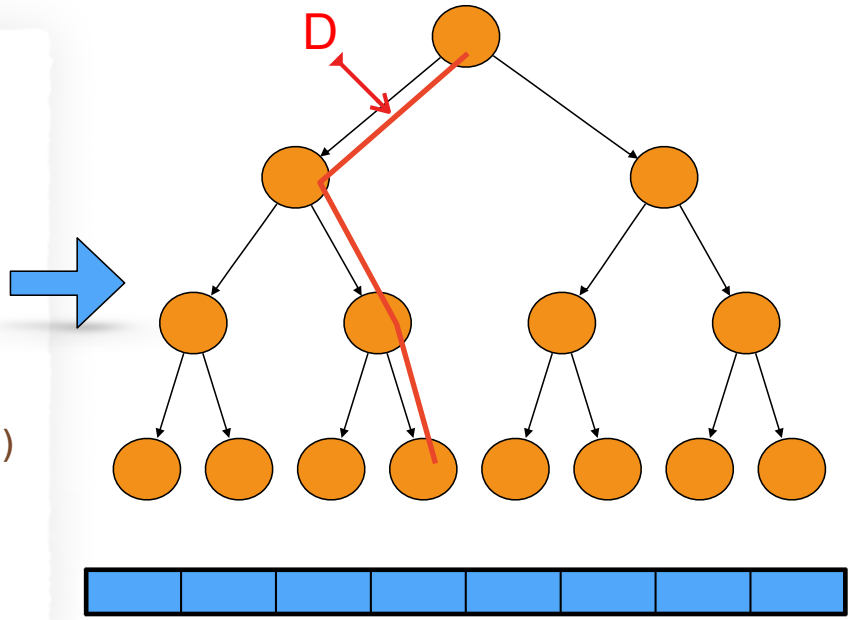


- **Pour ressources homogènes ou hybrides**
- **Principe**
  - **chaque cœur possède sa liste de tâches prêtes**
    - l'exécution d'une tâche peut rendre prêtes les successeurs de la tâche
  - **lorsqu'un cœur est inactif il cherche à voler une tâche prêtes chez un autre cœur**
    - choix aléatoire de la victime
- **Le vol = communication**

W et D

- W (le *work*) est le nombre d'opération pour exécuter le programme parallèle
  - W - $W_{séquentiel}$ = surcoût dû à l'utilisation du parallélisme
- D (le depth) est le chemin critique (plus long chemin) nécessaire pour exécuter le programme parallèle
  - ~ temps parallèle sur un nombre infini de cœurs

```
void foreach(int N, float* A)
{
  if (N<2)
    do_compute(A[0]);
  else
  {
    int mid = N/2;
    #pragma omp task depend(in: A[0:mid]) shared(A)
    foreach(mid, A);
    #pragma omp task depend(in: A[mid:N-mid]) shared(A)
    foreach(N-mid, &A[mid]);
  }
}
```



Vol de travail

- **Performance théorique**
  - **Avec de « bonnes chances » [Cilk 96] :** $C_{\max} \leq \frac{W}{p} + O(D)$
    - machine homogène (p CPUs)
  - **Avec de « bonnes chances » [Blender, Rabin 02] :** $C_{\max} \leq \frac{W}{p\Pi} + O\left(\frac{D}{\Pi}\right)$
    - où π = puissance moyenne
    - préemption des tâches
- **Nombre moyen de vols :** $N_p = O(pD)$

Implication sur l'algorithmique

- **D'après les résultats de l'algorithme d'ordonnancement par vol de travail**
  - **on recherche un algorithme qui :**
    - possède un travail W petit, idéalement W_{seq}
    - possède une profondeur D petite
      - si D très petit alors speedup quasi linéaire : $C_{\max} \approx \frac{W}{p}$

➡ **Algorithmes récursifs**

- **e.g. découpe récursive en sous problèmes de travaux W_i équivalents**
- **très bonne localité !**

5.Implémentation et optimisation du vol de travail

- ✓ - implémentation
- ✓ - optimisations des coûts liés au parallélisme et à l'ordonnancement des tâches
- ✓ - algorithmes adaptatifs

Implémentation du vol de travail

- **Structure de données fondamentale : la *workqueue***
  - **liste de tâches prêtes**
  - **3 fonctions**
    - push/pop : utilisées par le cœur sur sa propre *workqueue*
    - steal : utilisée par le cœur pour voler une tâche dans la workqueue d'un autre cœur
  - **des implémentations très efficaces**
    - T.H.E [Cilk] : utilisation des registres atomique (lecture/écriture atomique uniquement) + lock dans de rare cas
    - [ABP 98], […] : algorithme non bloquant, utilisation d'instruction atomique compare&swap

- **On retrouve cet algorithme d'ordonnancement à la base des logiciels :**
  - **Intel TBB, Cilk**
  - **OpenMP !**
    - si utilisation des supports exécutifs GCC/libKOMP ou ICC/libIOMP
    - construit au dessus XKaapi (http://kaapi.gforge.inria.fr)

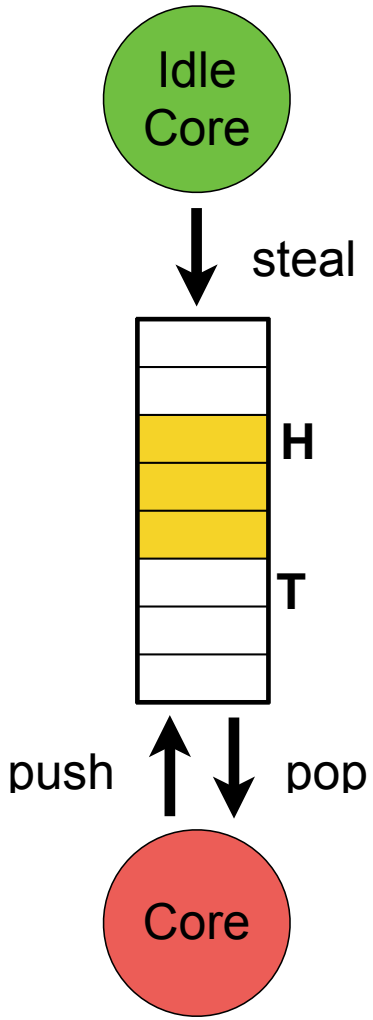
La workqueue « T.H.E. »

Worker/Victim

```
1   push() {
2     T++;
3   }

4   pop() {
5     T--;
6     if (H > T) {
7       T++;
8       lock(L);
9       T--;
10      if (H > T) {
11        T++;
12        unlock(L);
13        return FAILURE;
14      }
15      unlock(L);
16    }
17    return SUCCESS;
18  }
```

Thief

```
1   steal() {
2     lock(L);
3     H++;
4     if (H > T) {
5       H--;
6       unlock(L);
7       return FAILURE;
8     }
9     unlock(L);
10    return SUCCESS;
11  }
```



« The implementation of the Cilk-5 multithreaded language », Frigo et al., PLDI'98, ACM SIGPLAN'98

- Cas standard d'exécution : uniquement lignes 5,6 et 15 de « pop »

Optimisations possibles

- **Objectif**
  - **que le nombre d'opérations exécutées en parallèle soit le même que le nombre d'opérations exécutées par le code séquentiel**

- **D'où viennent les surcoûts ?**
  - **du coût de création d'une tâche**
    - représentation d'une tâche
    - lié à « *ajout d'opérations supplémentaires par rapport au code séquentiel* » ci-dessous
  - **du coût de l'algorithme de vol**
    - nombre important de requêtes N_p=O(pD) de vols, si D grand
    - gestion d'une meilleure localité des données
  - **du parallélisme de l'application**
    - ajout de synchronisations
    - ajout d'opérations supplémentaires par rapport au code séquentiel

Réduire le coût de création d'une tâche

- Exemple naïf

```
void fibonacci( int n, int *result )
{
  if (n <2) *result = n;
  else {
    int r1, r2;
    #pragma omp task shared(r1)
    r1 = fibonacci(n-1);
    #pragma omp task shared(r2)
    r2 = fibonacci(n-2);
    #pragma omp taskwait
    *result = r1+r2;
  }
}
```

OpenMP

Serial	Cilk+	TBB (4.0)	OpenMP (gcc)	XKaapi
0.0905s (x 1)	1.063 (x 11.7)	2.356s (x 26)	2.429s (x 27)	0.728s **(x 8)**

Sur 1 cœur, AMD [2012]

En parallèle avec les BOTS

- **Barcelona OpenMP Task Suite**
  - **un ensemble de benchmarks représentatifs utilisant les tâches OpenMP-3.1**
  - **avec libKOMP = une implémentation de libGOMP au dessus de XKaapi**

Name	*Arguments used*	*Domain*	*Summary*
Alignment	prot100.aa	Dynamic programming	Aligns sequences of proteins
FFT	n=33,554,432	Spectral method	Computes a Fast Fourier Transformation
Floorplan	input.20	Optimization	Computes the optimal placement of cells in a floorplan
NQueens	n=14	Search	Finds solutions of the N Queens problem
MultiSort	n=33,554,432	Integer sorting	Uses a mixture of sorting algorithms to sort a vector
SparseLU	n=128 m=64	Sparse linear algebra	Computes the LU factorization of a sparse matrix
Strassen	n=8192	Dense linear algebra	Computes a matrix multiply with Strassen's method
UTS	medium.input	Search	Computes the number of nodes in an Unbalanced Tree

- **Architecture d'évaluation**
  - **AMD48: 4x12 AMD Opteron (6174) cores**

- **Logiciels**
  - **gcc 4.6.2 + libGOMP**
  - **gcc 4.6.2 + libKOMP**
  - **icc 12.1.2 + Intel OpenMP runtime (KMP)**

Résultats expérimentaux

kernel	***libGOMP***	***libKOMP***	***Intel***
Alignment	38.8	**40.0**	37.0
FFT	0.5	**12.2**	12.0
Floorplan	27.6	**32.7**	29.2
NQueens	43.7	**47.8**	39.0
MultiSort	0.6	**13.2**	11.3
SparseLU	44.1	**44.4**	35.0
Strassen	20.8	**22.4**	20.5
UTS	0.9	**25.3**	15.0

Speed-Up of BOTS kernels on the 48 cores of the AMD48 platform

- **[IWOMP 2012]**
  - **libKOMP au mieux : 28 fois plus rapide que GCC [UTS] et 1.69 fois que ICC [UTS]**

Passage à l'échelle sur Intel Xeon Phi

- Fibonacci (38) naive recursive computation
- 60 cores of Intel Xeon Phi 5100
- [SBAC-PAD 2013]

#threads	Cilk+	OpenMP (icc)	XKaapi
1	33.21	65.64	15.52
10	3.34	33.12	1.58
20	1.66	17.54	0.79
60	0.56	6.30	0.27
120	0.38	3.86	0.18
240	0.37	3.18	0.18

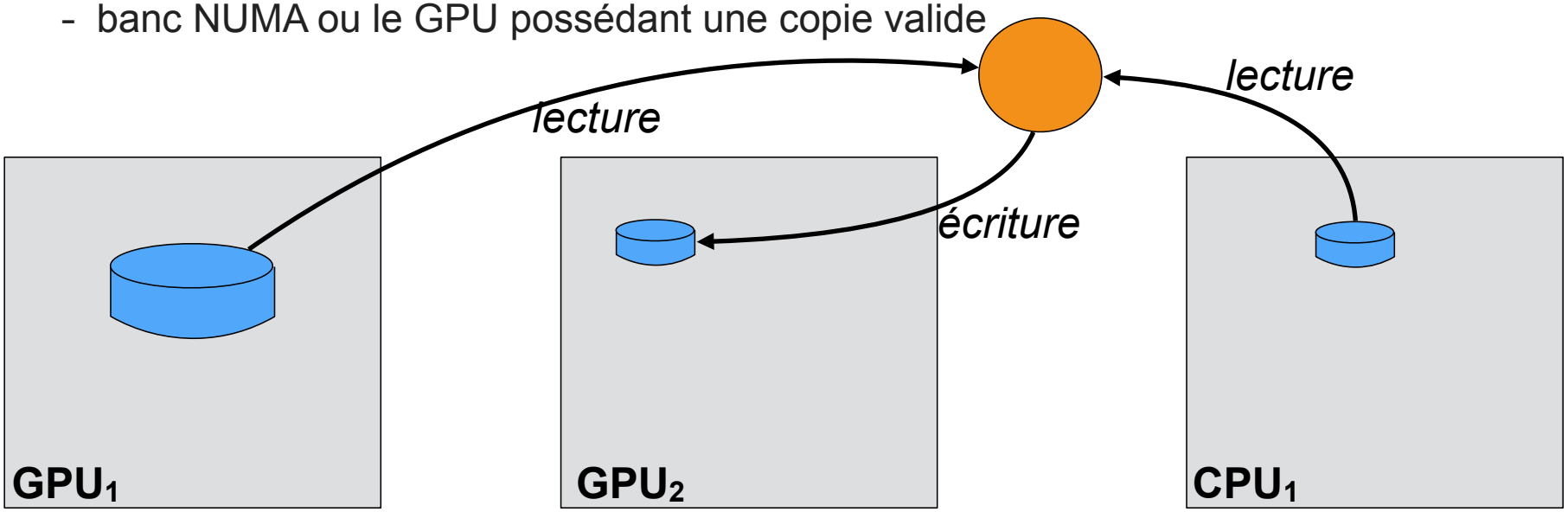
temps en seconde

Vol de travail et localité des données

- De nombreux algorithmes
  - vital pour les architectures ayant un accès mémoire distant important
- 2 Idées principales :
  - voler d'abord localement (proche au sens de la hiérarchie mémoire)
  - pousser une tâche prêtes là où se trouvent ses données

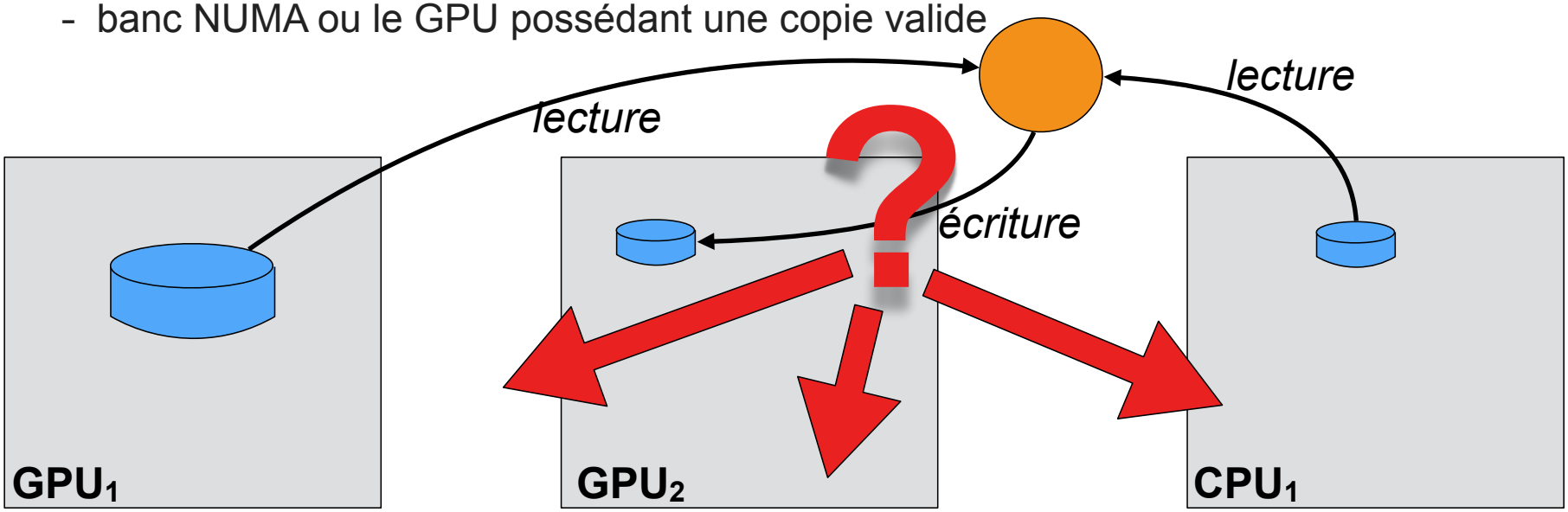
Un exemple XKaapi : heuristiques H1 et H2

- **On suppose que pour chaque tâche, le support exécutif connait :**
  - **la taille des données utilisées**
  - **leur emplacement sur la machine**
    - banc NUMA ou le GPU possédant une copie valide



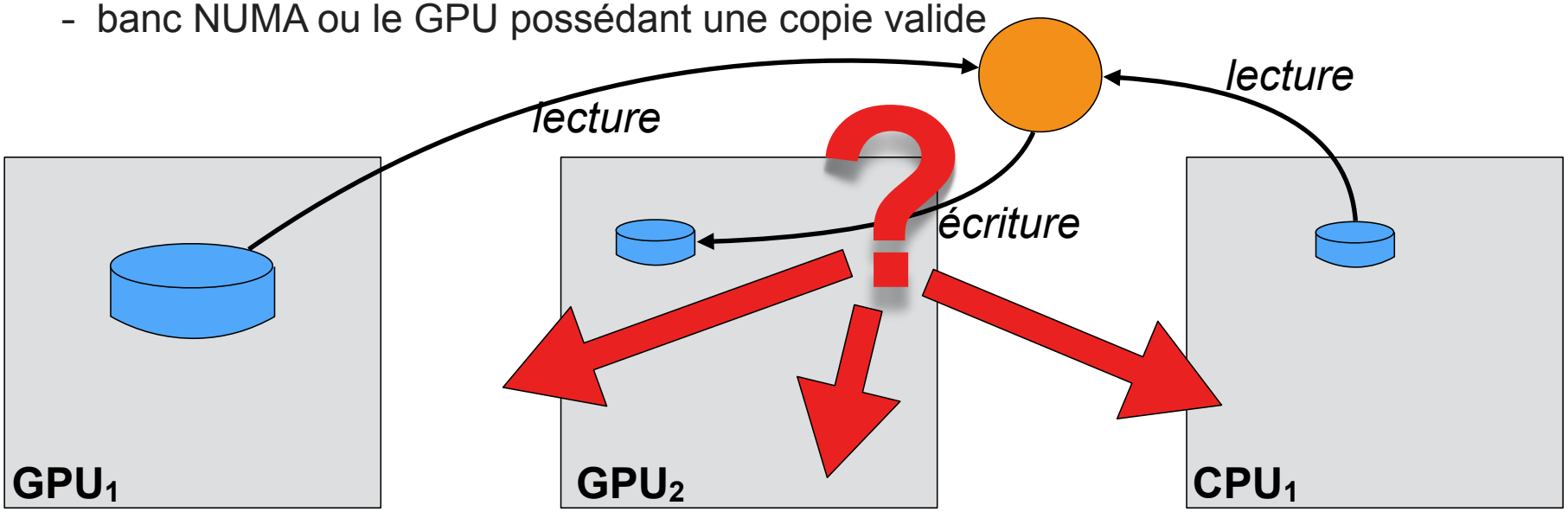
Un exemple XKaapi : heuristiques H1 et H2

- **On suppose que pour chaque tâche, le support exécutif connait :**
  - **la taille des données utilisées**
  - **leur emplacement sur la machine**
    - banc NUMA ou le GPU possédant une copie valide



Un exemple XKaapi : heuristiques H1 et H2

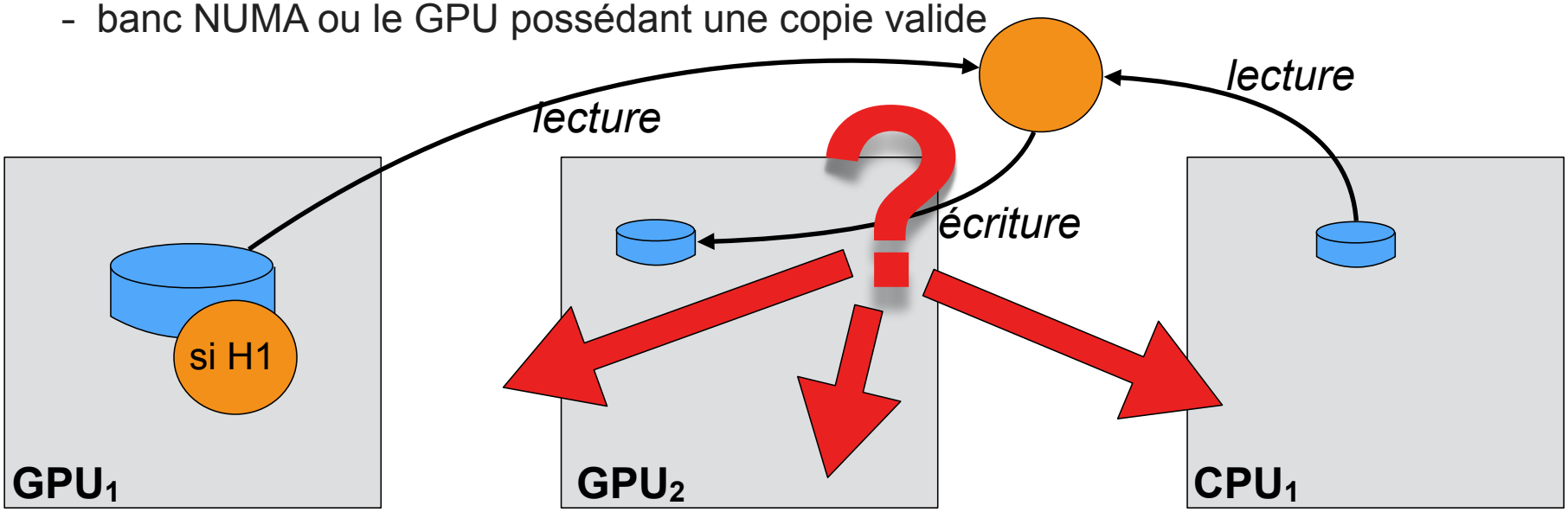
- On suppose que pour chaque tâche, le support exécutif connait :
  - la taille des données utilisées
  - leur emplacement sur la machine
    - banc NUMA ou le GPU possédant une copie valide



- H1 : on pousse une tâche prête sur la ressource qui minimise le volume de données à communiquer
- H2 : on pousse une tâche prête sur la ressource qui possède une donnée qui sera modifiée par la tâche

Un exemple XKaapi : heuristiques H1 et H2

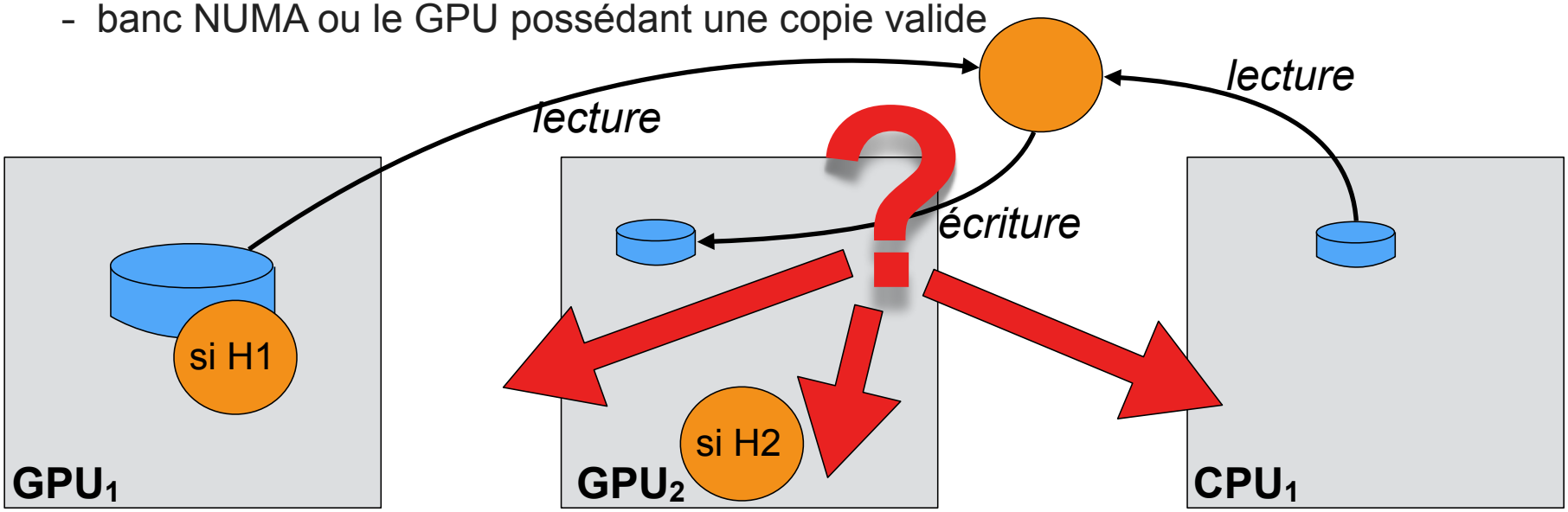
- On suppose que pour chaque tâche, le support exécutif connait :
  - la taille des données utilisées
  - leur emplacement sur la machine
    - banc NUMA ou le GPU possédant une copie valide



- H1 : on pousse une tâche prête sur la ressource qui minimise le volume de données à communiquer
- H2 : on pousse une tâche prête sur la ressource qui possède une donnée qui sera modifiée par la tâche

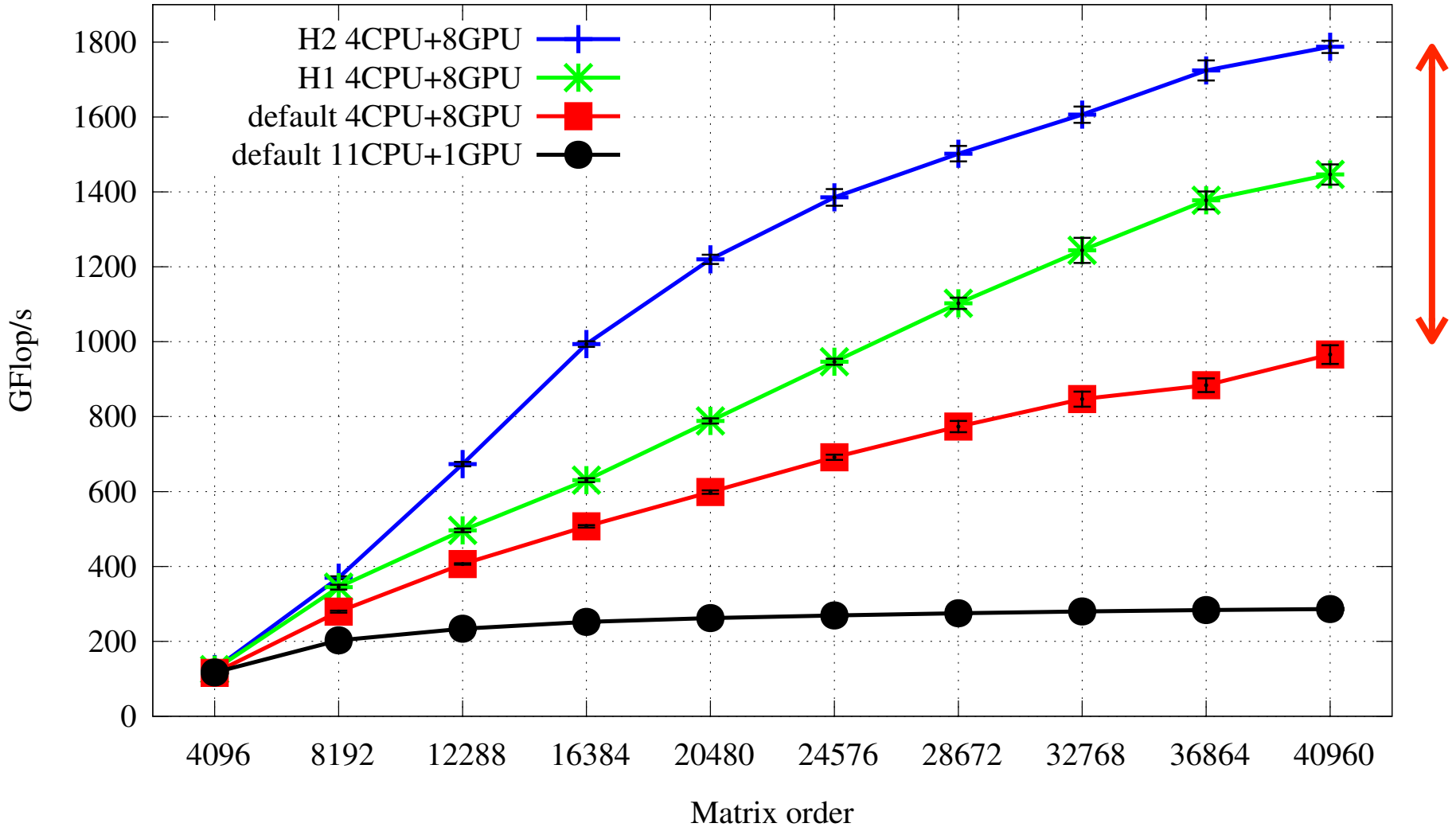
Un exemple XKaapi : heuristiques H1 et H2

- On suppose que pour chaque tâche, le support exécutif connait :
  - la taille des données utilisées
  - leur emplacement sur la machine
    - banc NUMA ou le GPU possédant une copie valide



- H1 : on pousse une tâche prête sur la ressource qui minimise le volume de données à communiquer
- H2 : on pousse une tâche prête sur la ressource qui possède une donnée qui sera modifiée par la tâche

Résultats : de 1TFlops à 1.8TFlops !



- Architecture hybride idgraph, 8 GPU NVidia M2050
- 2.4TFlops pour DGEMM
  - [IPDPS 2013]

Peut-on gagner davantage ?

- **Création d'une tâche = instructions supplémentaires par rapport au séquentiel**
  - **dans les opérations de création**
    - dans XKaapi la création d'une tâche coûte 10 cycles (environ) + 3 cycles par pointeur
  - **dans les opérations nécessaires pour gérer le parallélisme !**

- **Un exemple : calcul du préfixe**
  - **input :** $a_i, i \in \{0, N-1\}$ **output :** $p_i = \sum_{k=0}^{i} a_k, i \in \{0, N-1\}$
  - **W_{seq} = N-1**
  - **[Ladner et Fischer, 80] un algorithme en** $D = 2\log_2 N$ **et** $W = 2N$
  - **[Fich 83] si** $D = \log_2 N$ **alors** $W = 4N$

Peut-on gagner davantage ?

- **Création d'une tâche = instructions supplémentaires par rapport au séquentiel**
  - **dans les opérations de création**
    - dans XKaapi la création d'une tâche coûte 10 cycles (environ) + 3 cycles par pointeur
  - **dans les opérations nécessaires pour gérer le parallélisme !**

- **Un exemple : calcul du préfixe**
  - **input :** $a_i, i \in \{0, N-1\}$ **output :** $p_i = \sum_{k=0}^{i} a_k, i \in \{0, N-1\}$
  - **W_{seq} = N-1**
  - **[Ladner et Fischer, 80] un algorithme en** $D = 2\log_2 N$ **et** $W = 2N$
  - **[Fich 83] si** $D = \log_2 N$ **alors** $W = 4N$

➡**Algorithme adaptatif**

5.Implémentation et optimisation du vol de travail

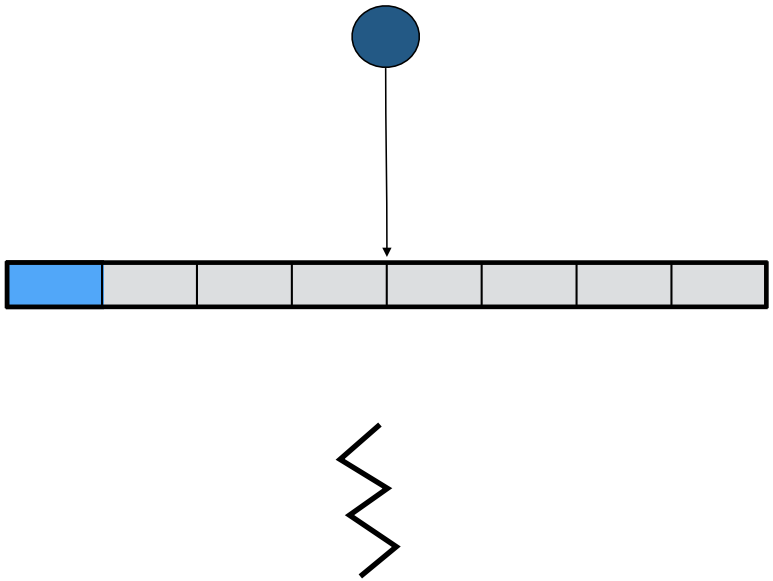
- ✓ - implémentation
- ✓ - optimisations des coûts liés au parallélisme et à l'ordonnancement des tâches
- ✓ - algorithmes adaptatifs

Algorithme adaptatif

- **Adapter le parallélisme du programme aux ressources**
  - **dans le calcule du préfixe de [Ladner & Fischer] :**
    - pourquoi écrire un algorithme pour un nombre très grand de processeurs ?
    - en pratique p=4 à 64 (actuellement)

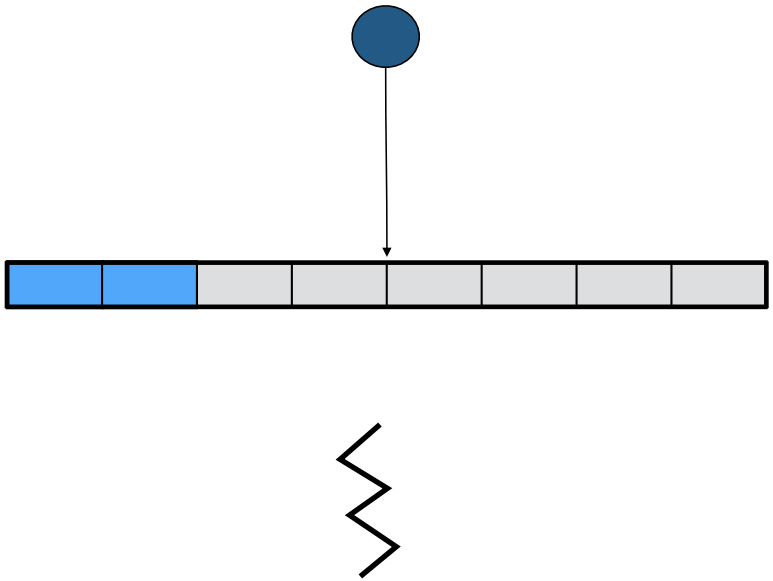
- **Idée : capturer une requête de vol de travail pour créer les tâches de l'application**
  - **[Roch & Traore 06]**
  - **[Roch & Gautier 05]**

Illustration : boucles indépendantes



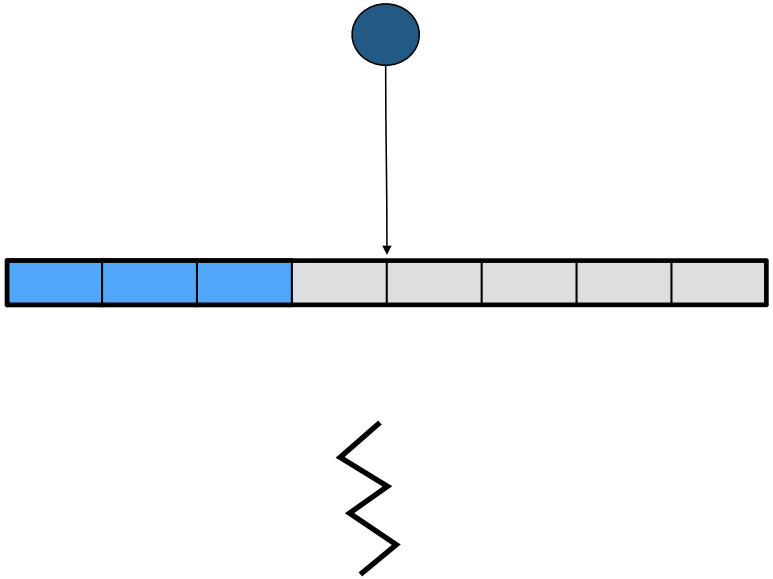
1 cœur

Illustration : boucles indépendantes



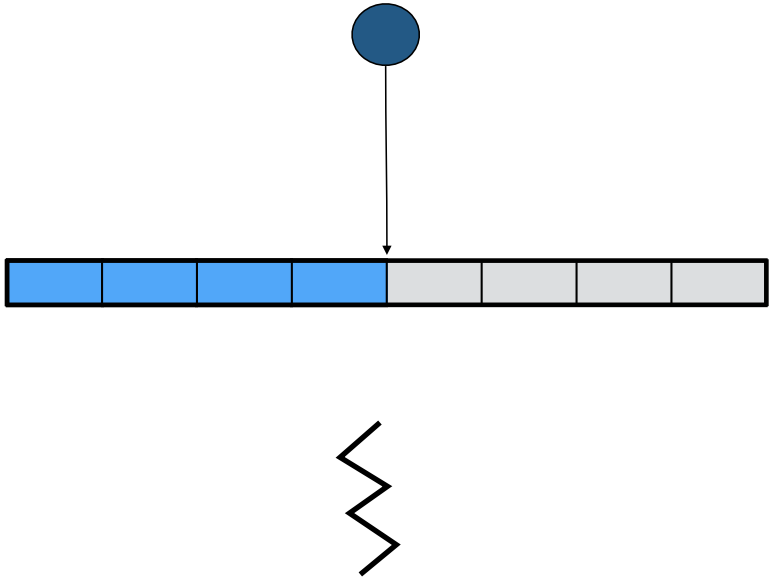
1 cœur

Illustration : boucles indépendantes



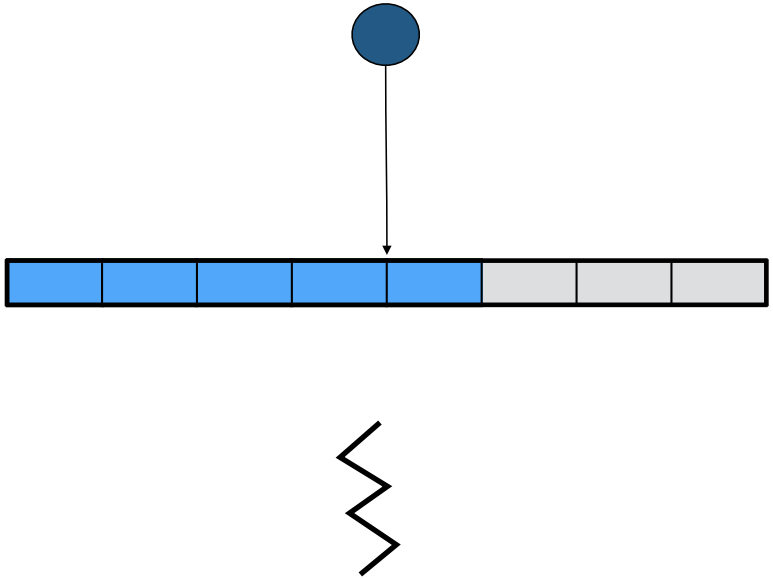
1 cœur

Illustration : boucles indépendantes



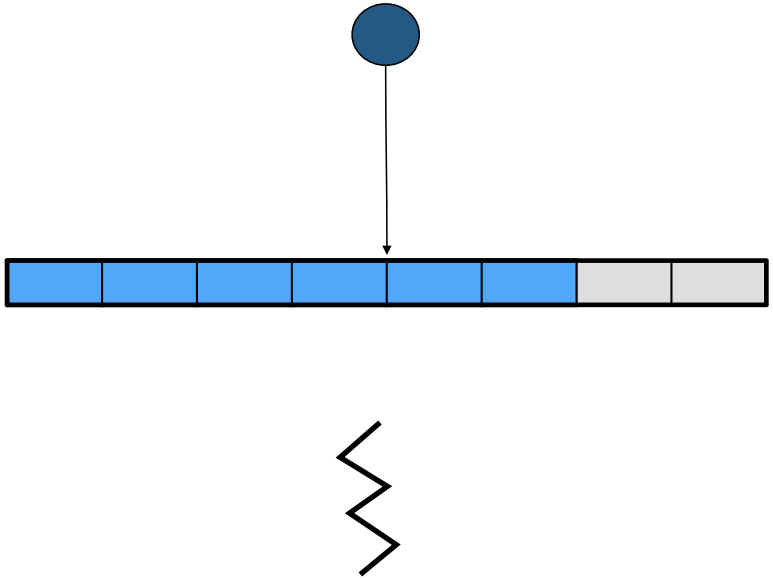
1 cœur

Illustration : boucles indépendantes



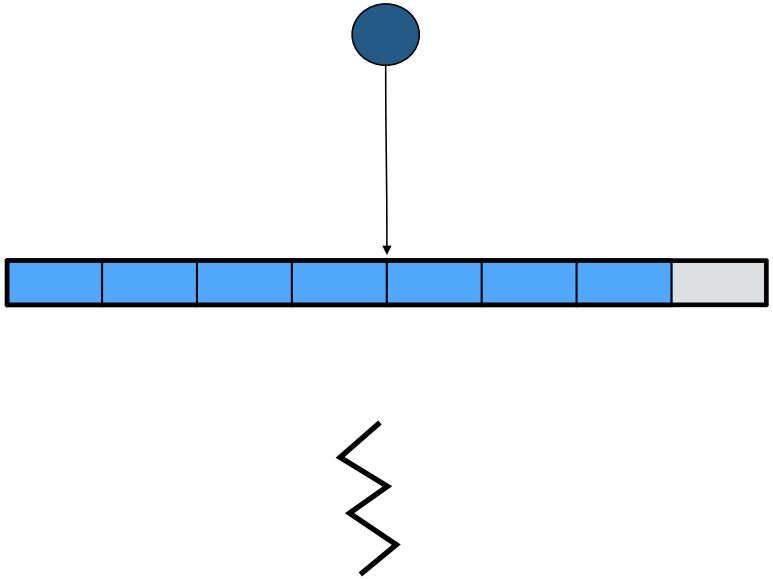
1 cœur

Illustration : boucles indépendantes



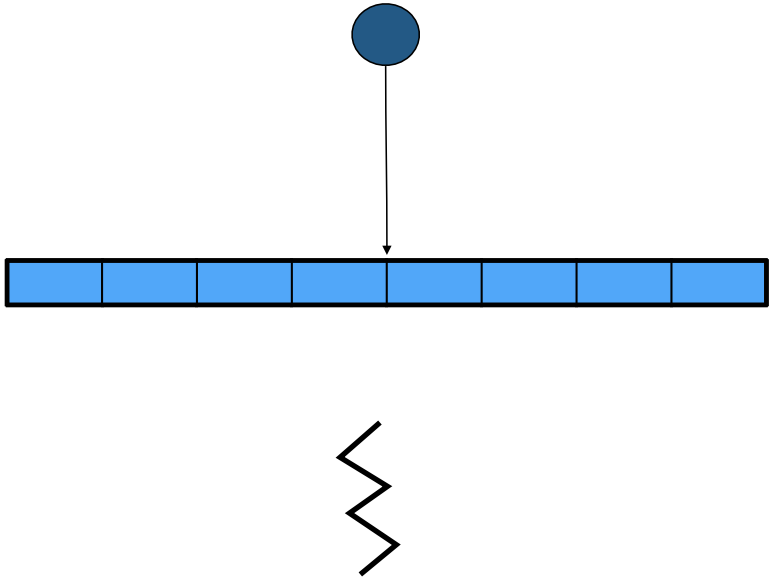
1 cœur

Illustration : boucles indépendantes



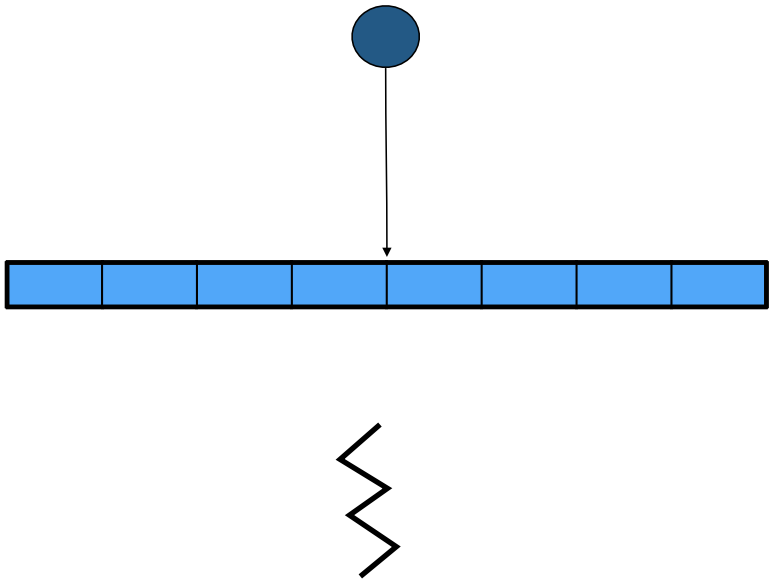
1 cœur

Illustration : boucles indépendantes



1 cœur

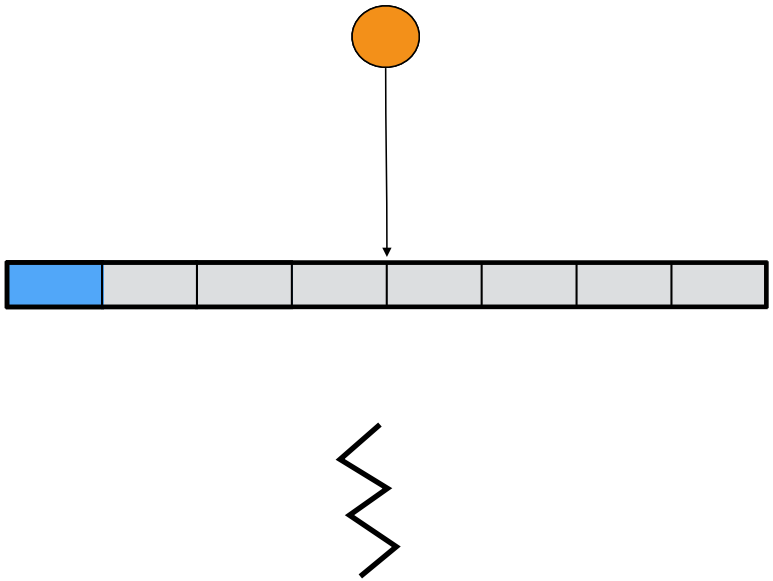
Illustration : boucles indépendantes



1 cœur

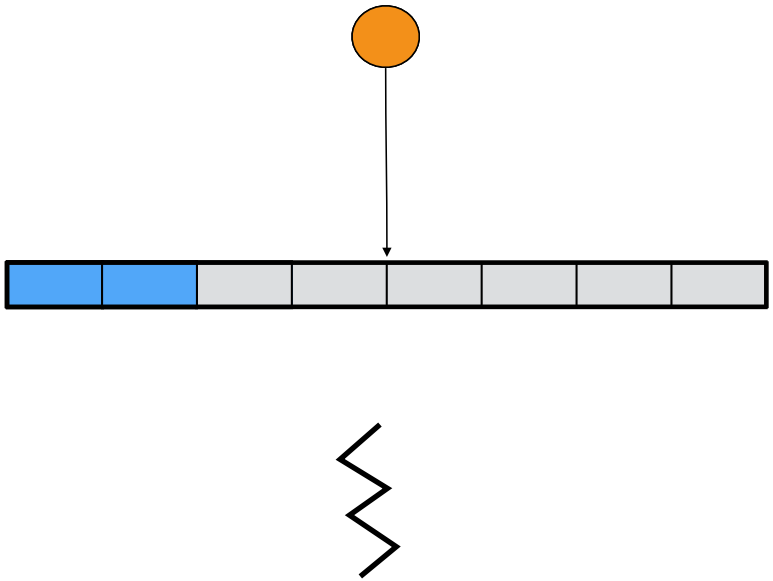
- **Conclusion**
  - **une seule création de tâche**
  - **surcoût très faible W ~ $W_{séquentiel}$**

Illustration : boucles indépendantes



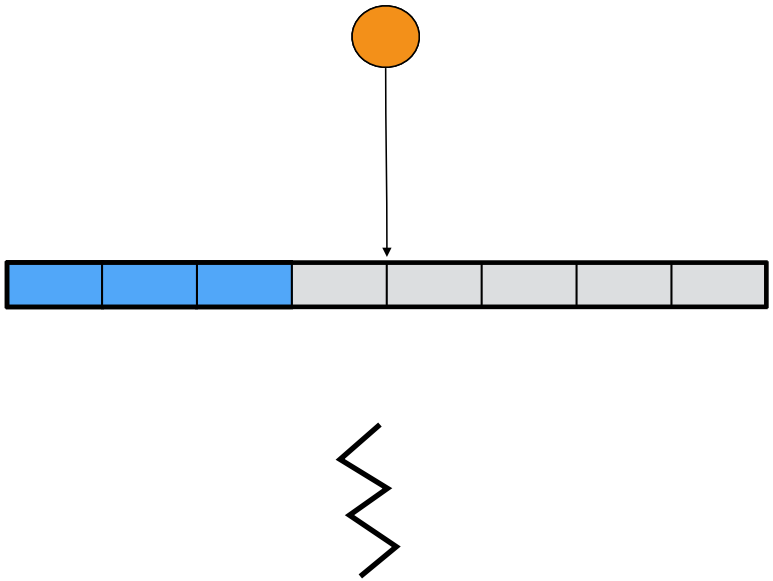
2 cœurs

Illustration : boucles indépendantes



2 cœurs

Illustration : boucles indépendantes



2 cœurs

Illustration : boucles indépendantes

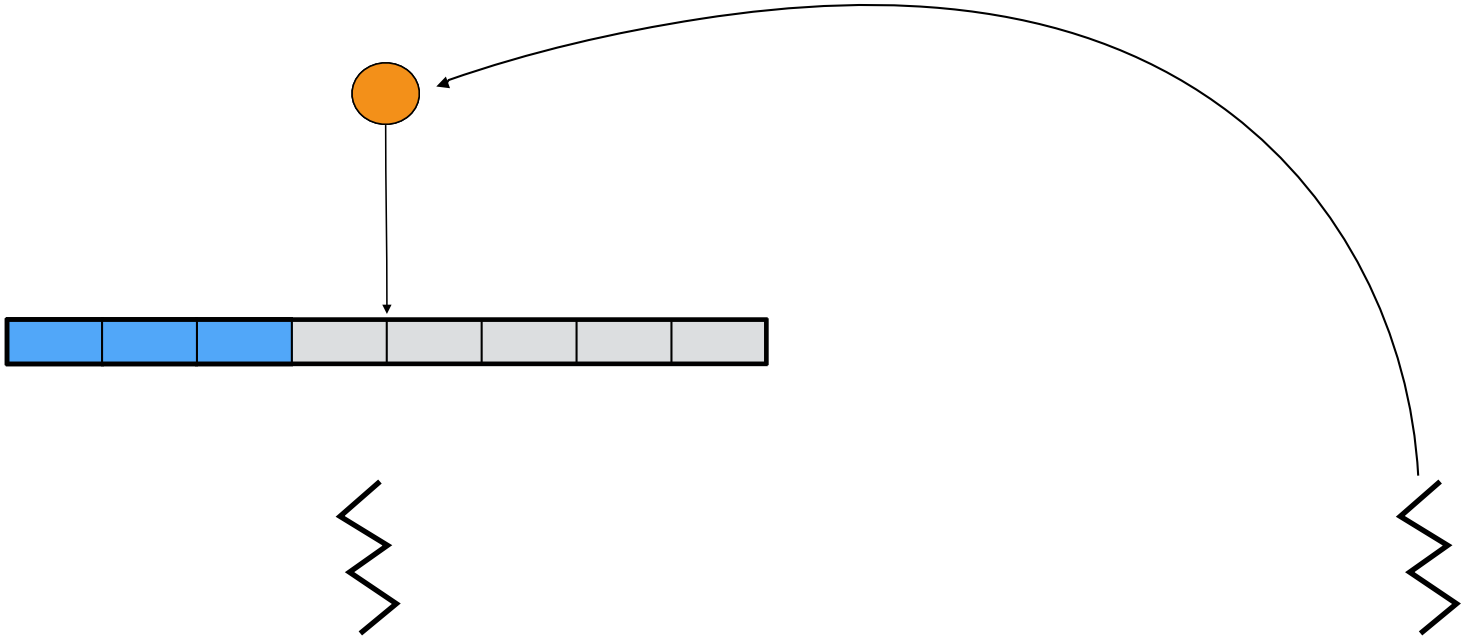


Illustration : boucles indépendantes

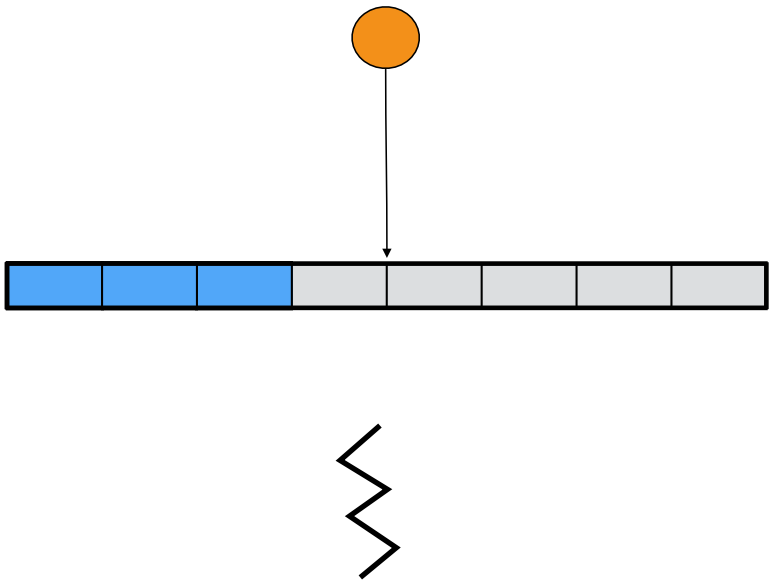


Illustration : boucles indépendantes

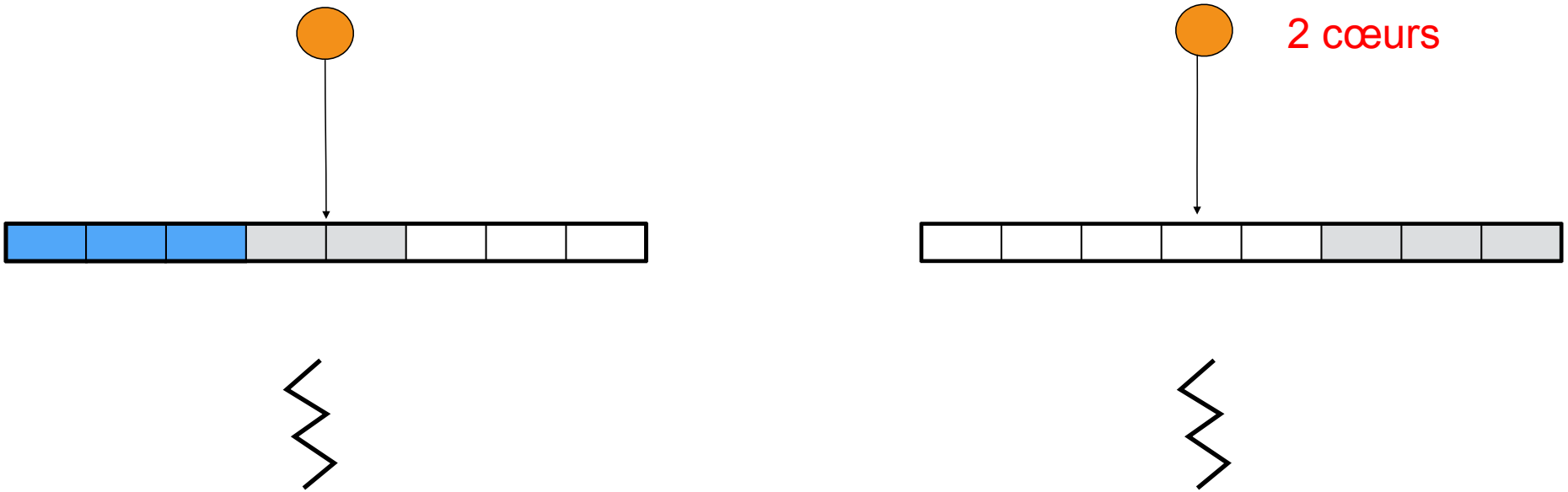


Illustration : boucles indépendantes

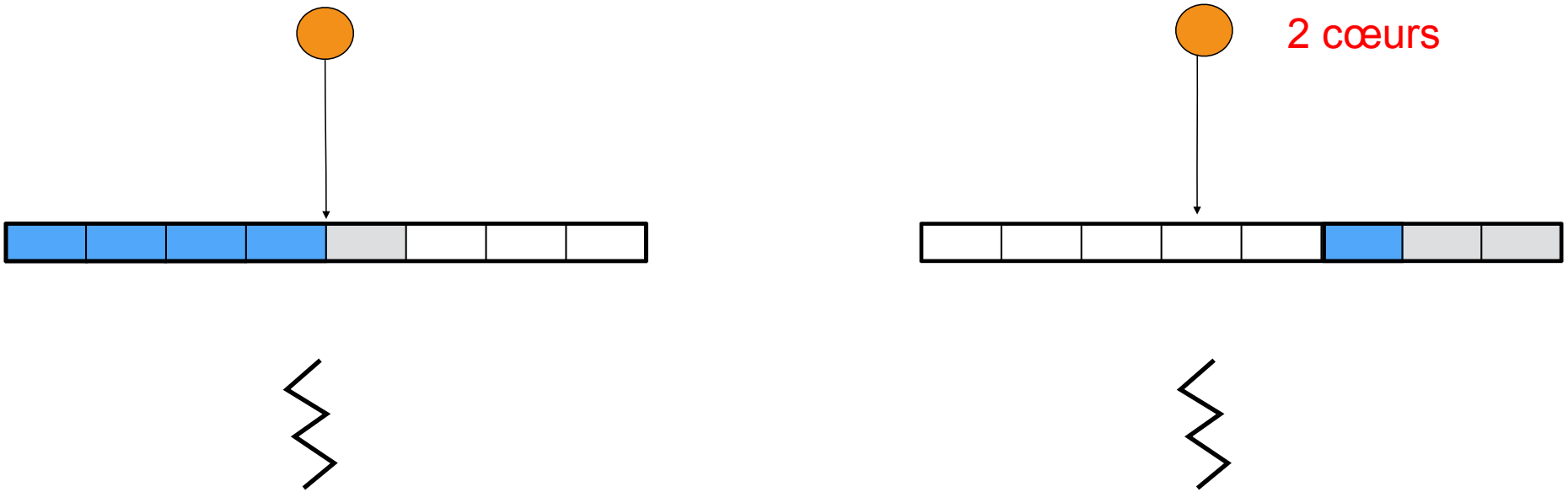


Illustration : boucles indépendantes

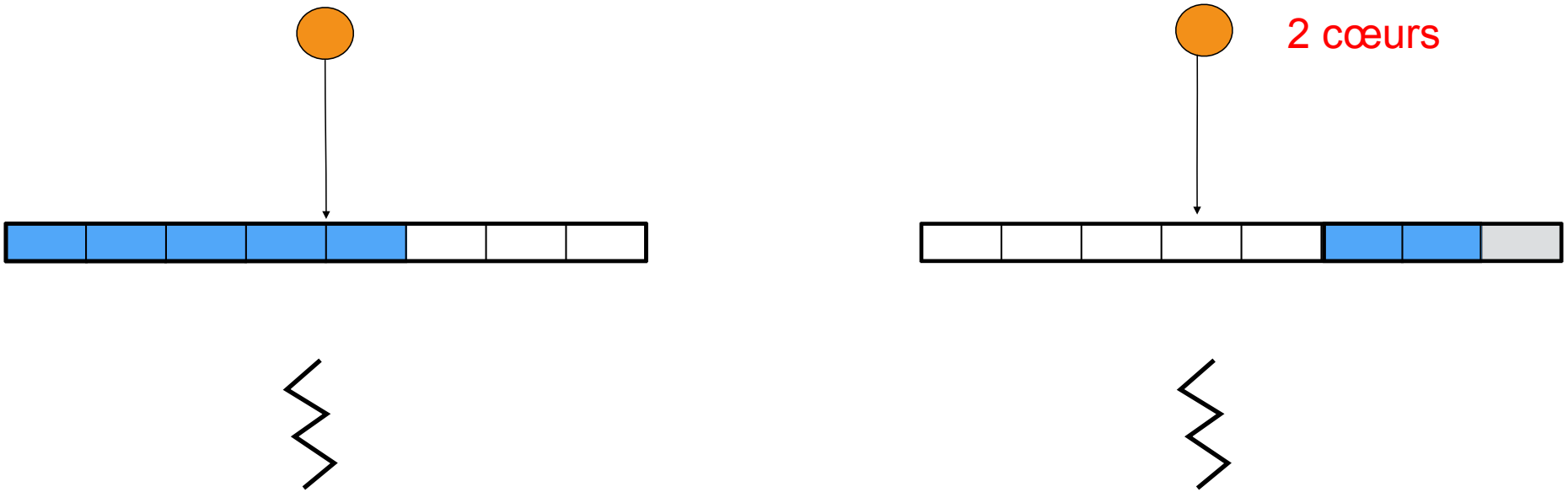


Illustration : boucles indépendantes

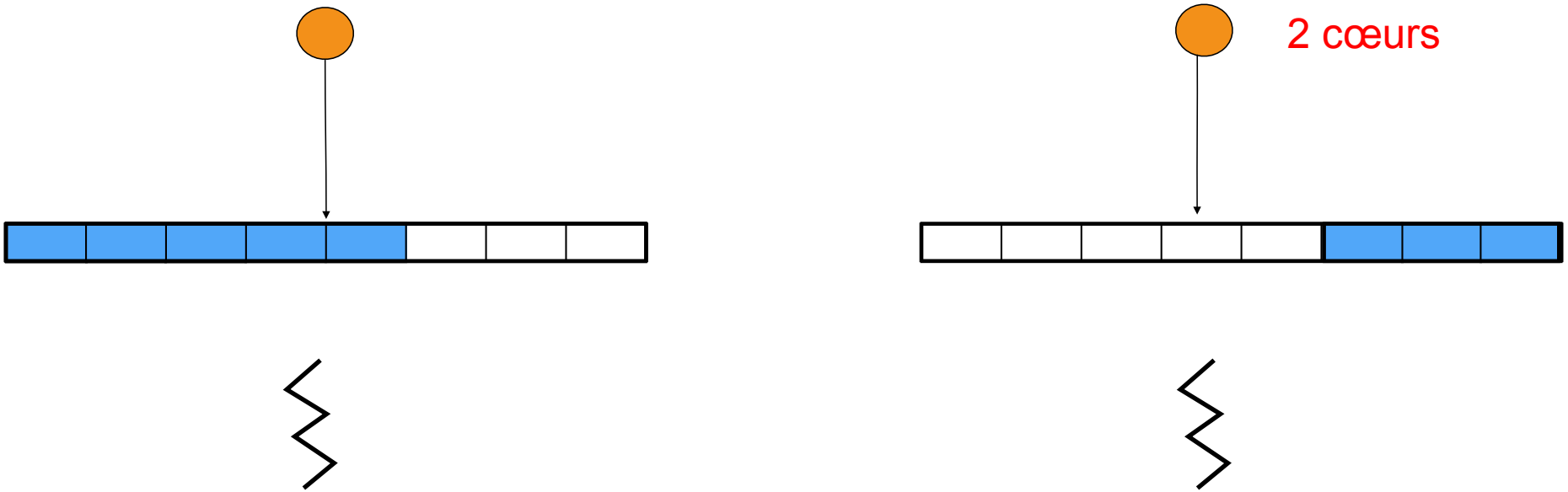
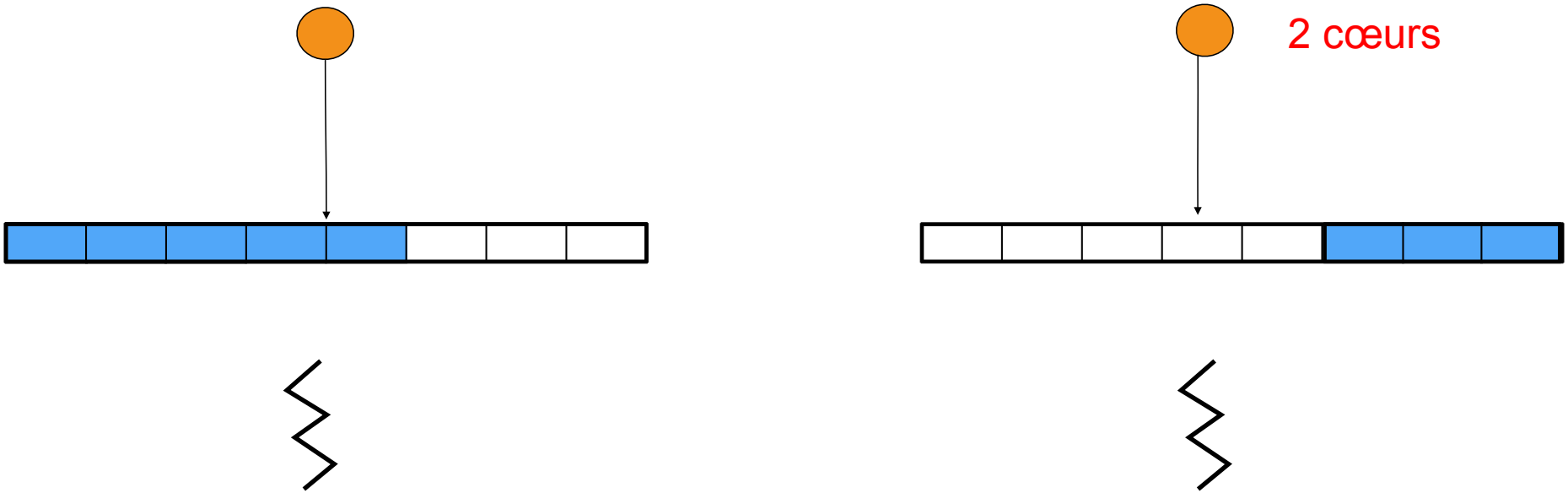
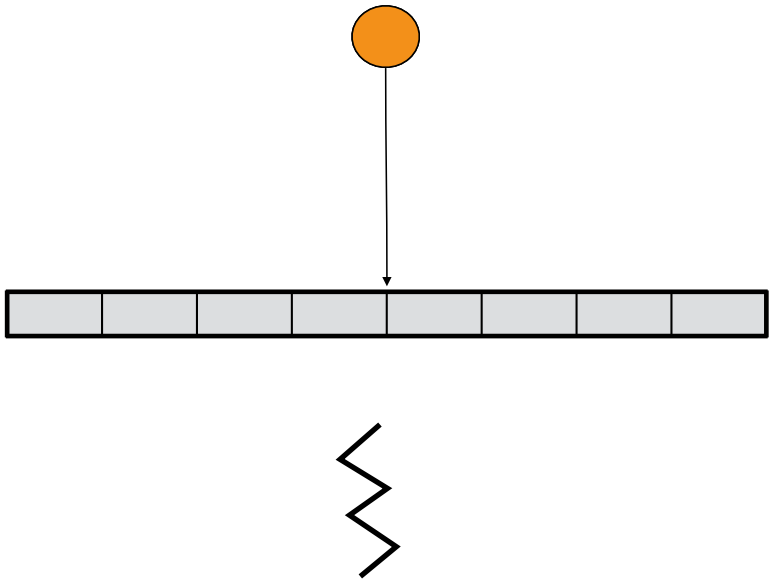


Illustration : boucles indépendantes



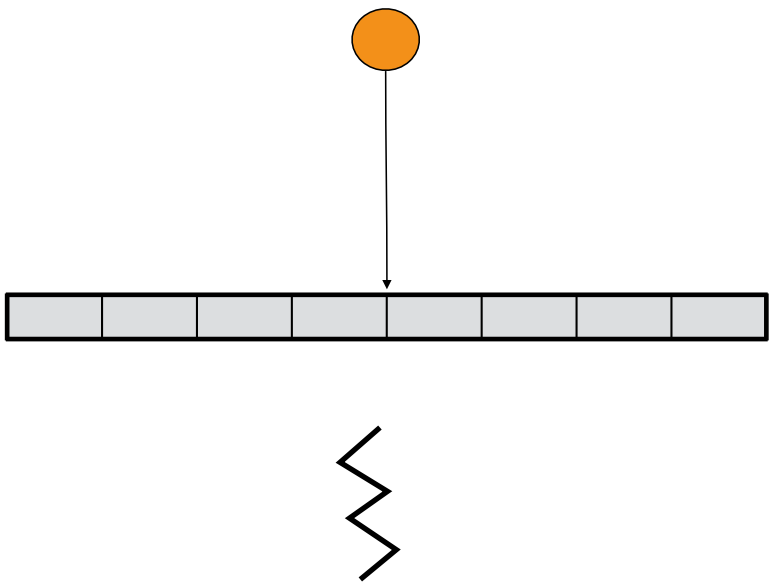
- **Conclusion**
  - **création de tâche lorsque nécessaire : à la demande lors des requêtes de vol**
  - **création de tâche adaptée au parallélisme**
  - **bonus :**
    - parcours linéaire des itérations par les cœurs
    - bonne utilisation des « prefetchers »

Illustration : boucles indépendantes



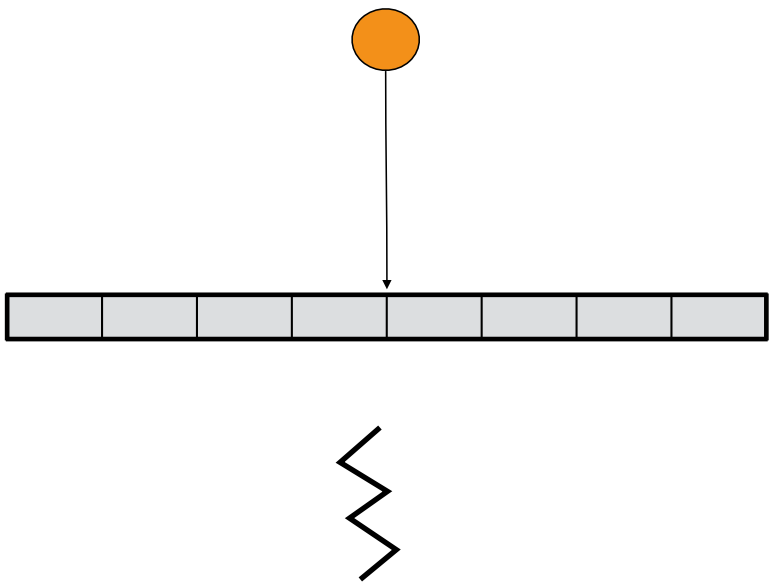
3 cœurs

Illustration : boucles indépendantes



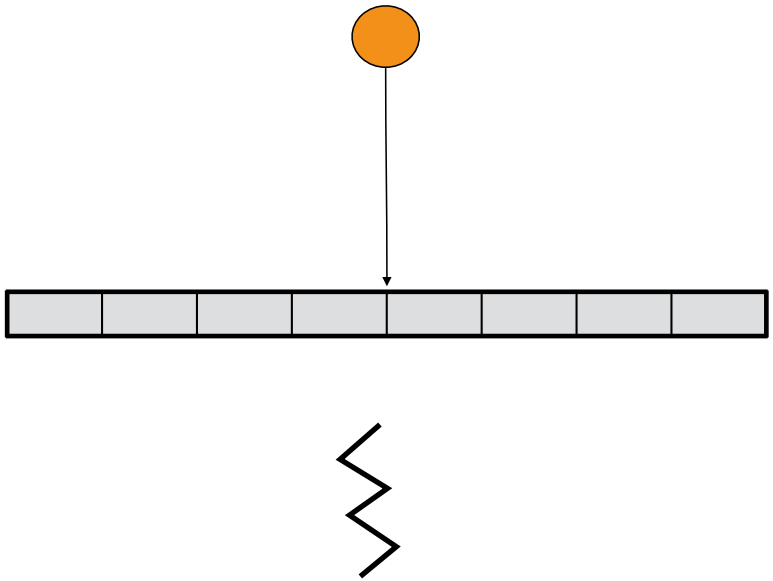
3 cœurs

Illustration : boucles indépendantes



3 cœurs

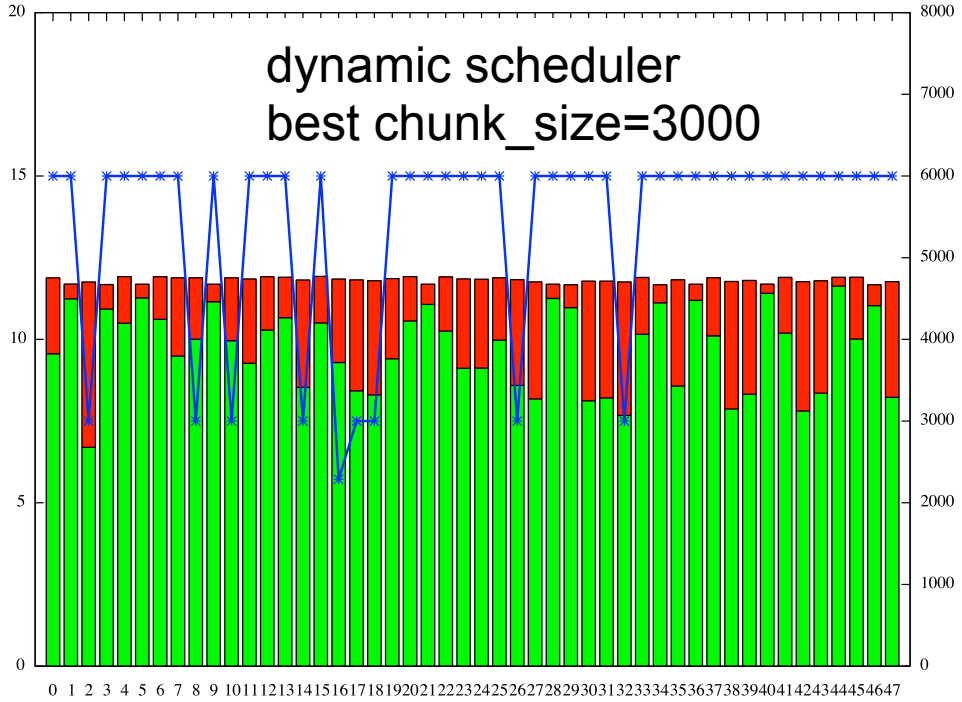
Illustration : boucles indépendantes

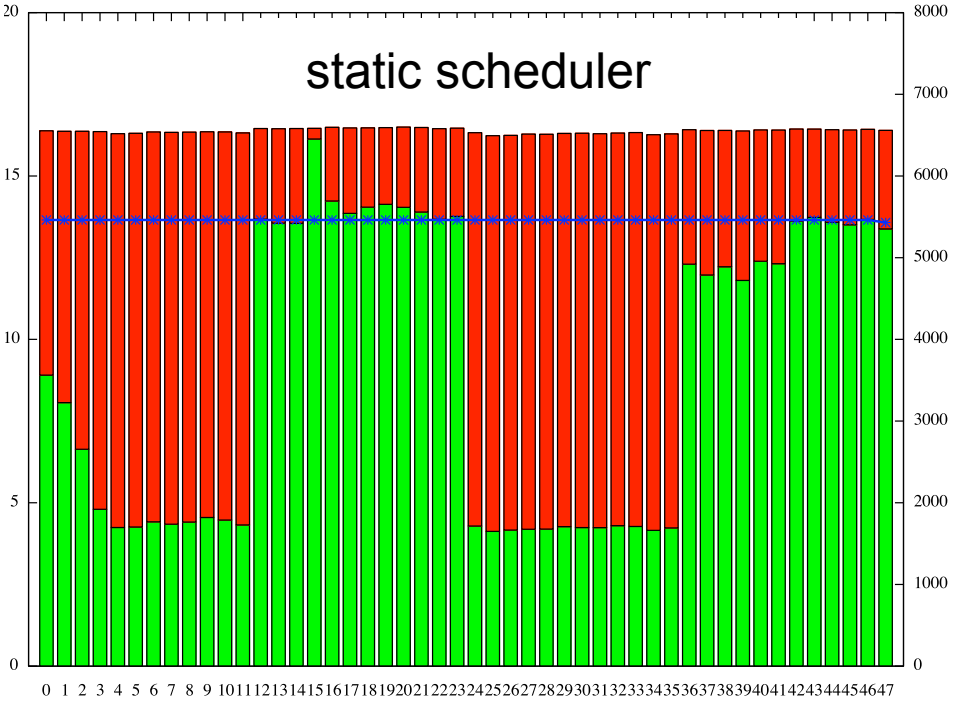


3 cœurs

- …

Extension de GCC/OpenMP [iwomp2013]





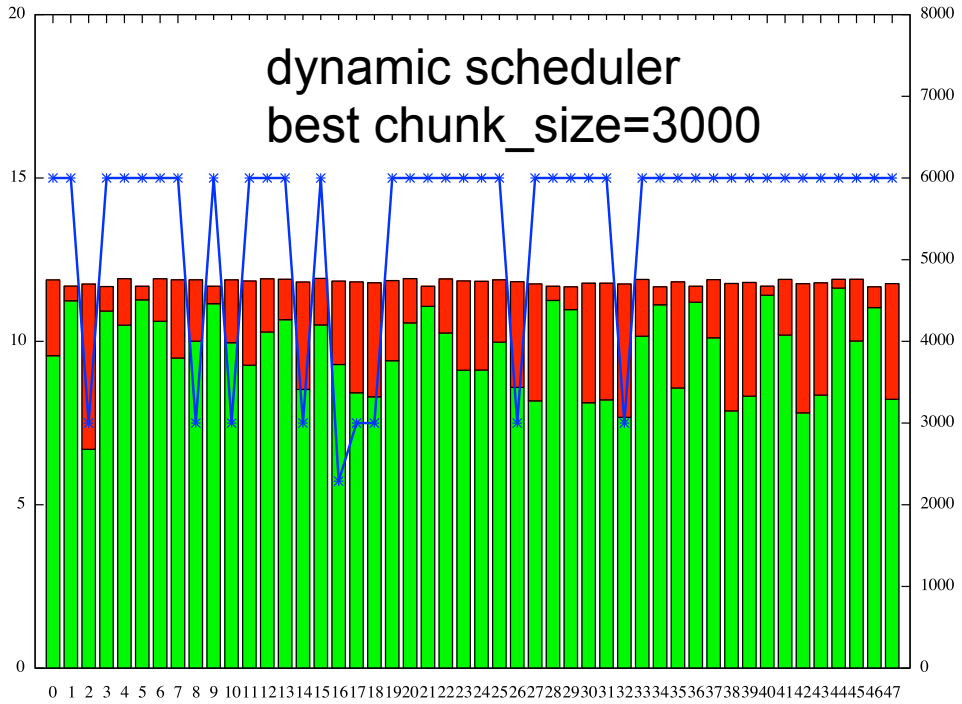
Time spent in computation

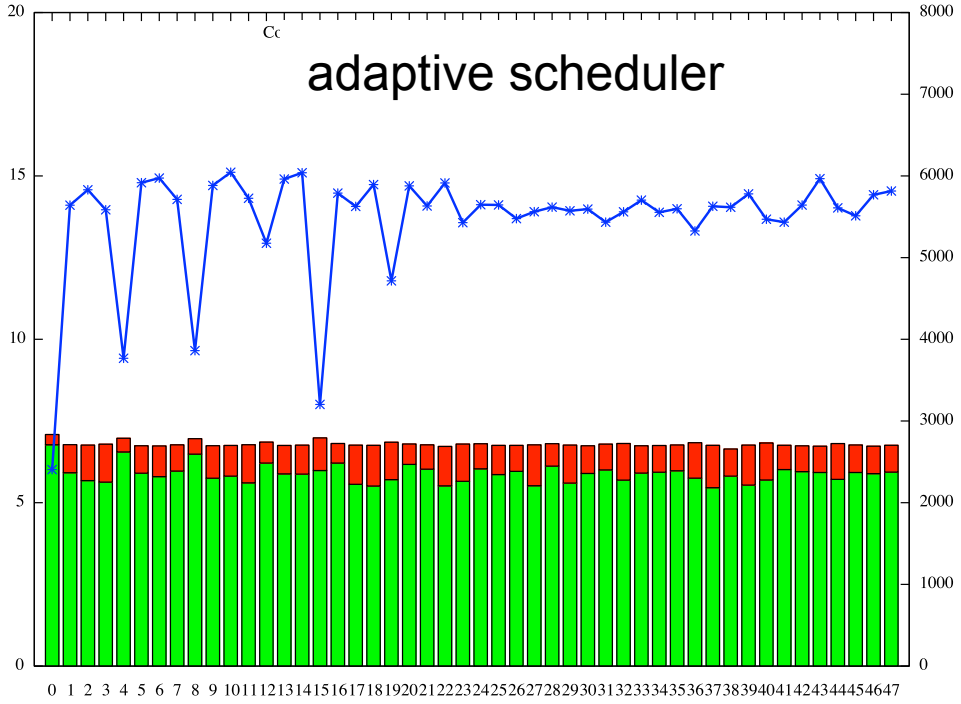
Time spent in scheduling (overhead)

Number of iterations executed by each thread

- **Static scheduler unable to balance the load**
- **Better load balancing with dynamic scheduler, but poor affinity control**

Extension de GCC/OpenMP





Time spent in computation

Time spent in scheduling (overhead)

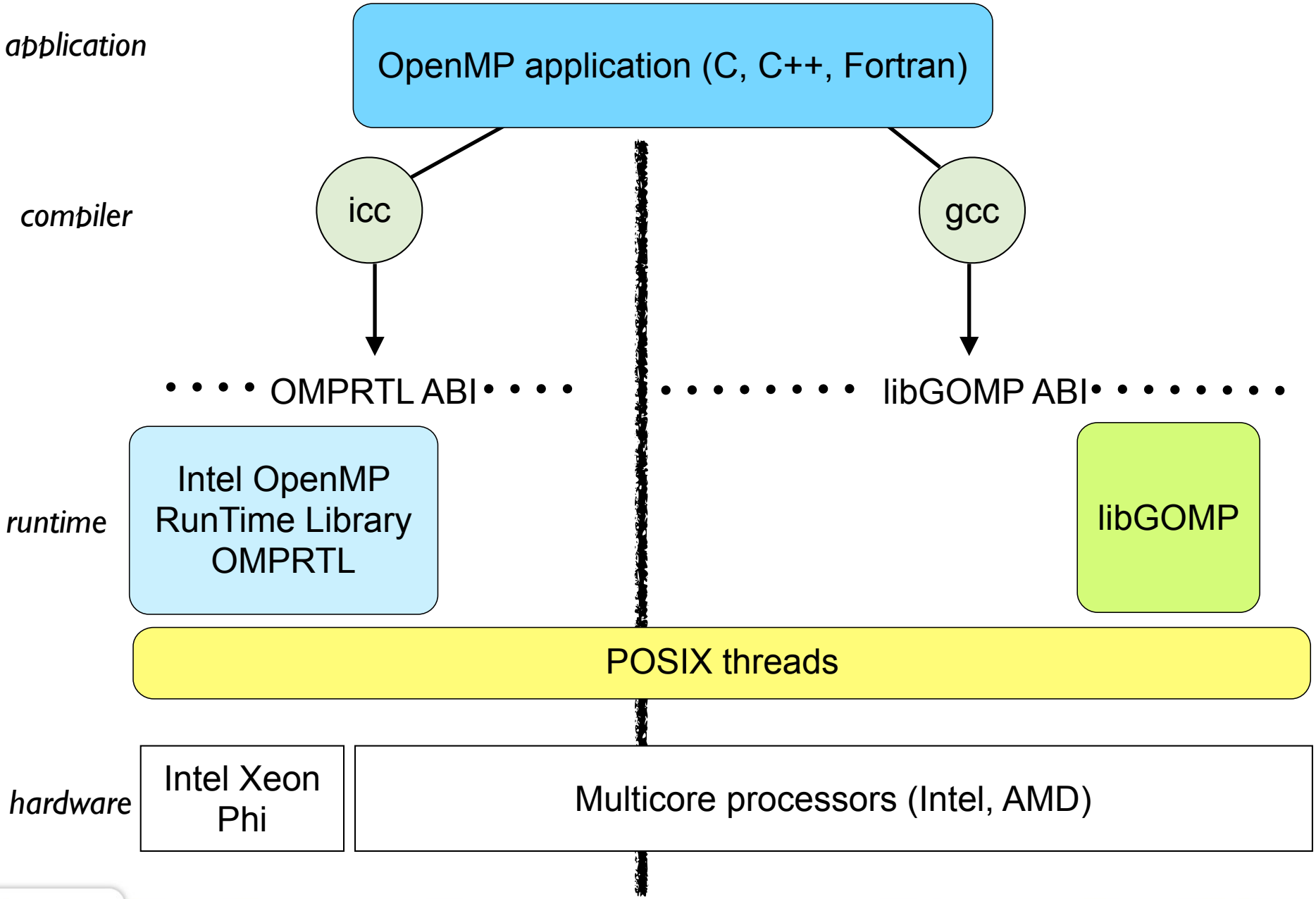
Number of iterations executed by each thread

- **Static scheduler unable to balance the load**
- **Better load balancing with dynamic scheduler, but poor affinity control**
- **Adaptive scheduler: good load balancing + good locality => smaller overheads**

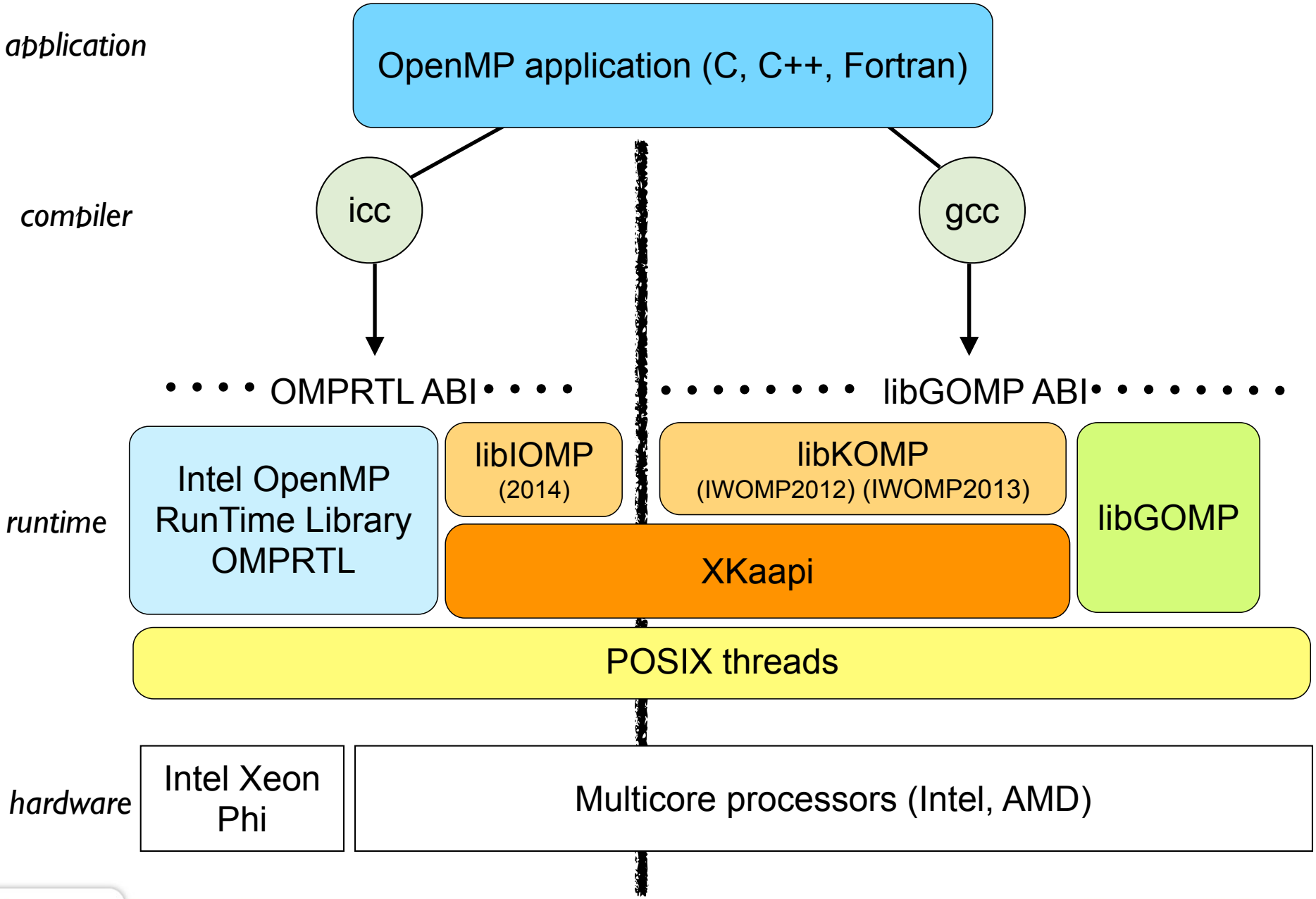
Pour conclure

- ✓ Vol de travail bien adapté pour
  - ordonnancement de programme de tâches ou de boucle !
  - sur architecture homogène ou hétérogène (hybride), multi-cœurs ou many-cœurs
- ✓ Il est possède des coûts faibles d'implémentation
  - il permet de reporter la création des tâches lors de l'inactivité des ressources
- ✓ Comment utiliser un tel outil ?

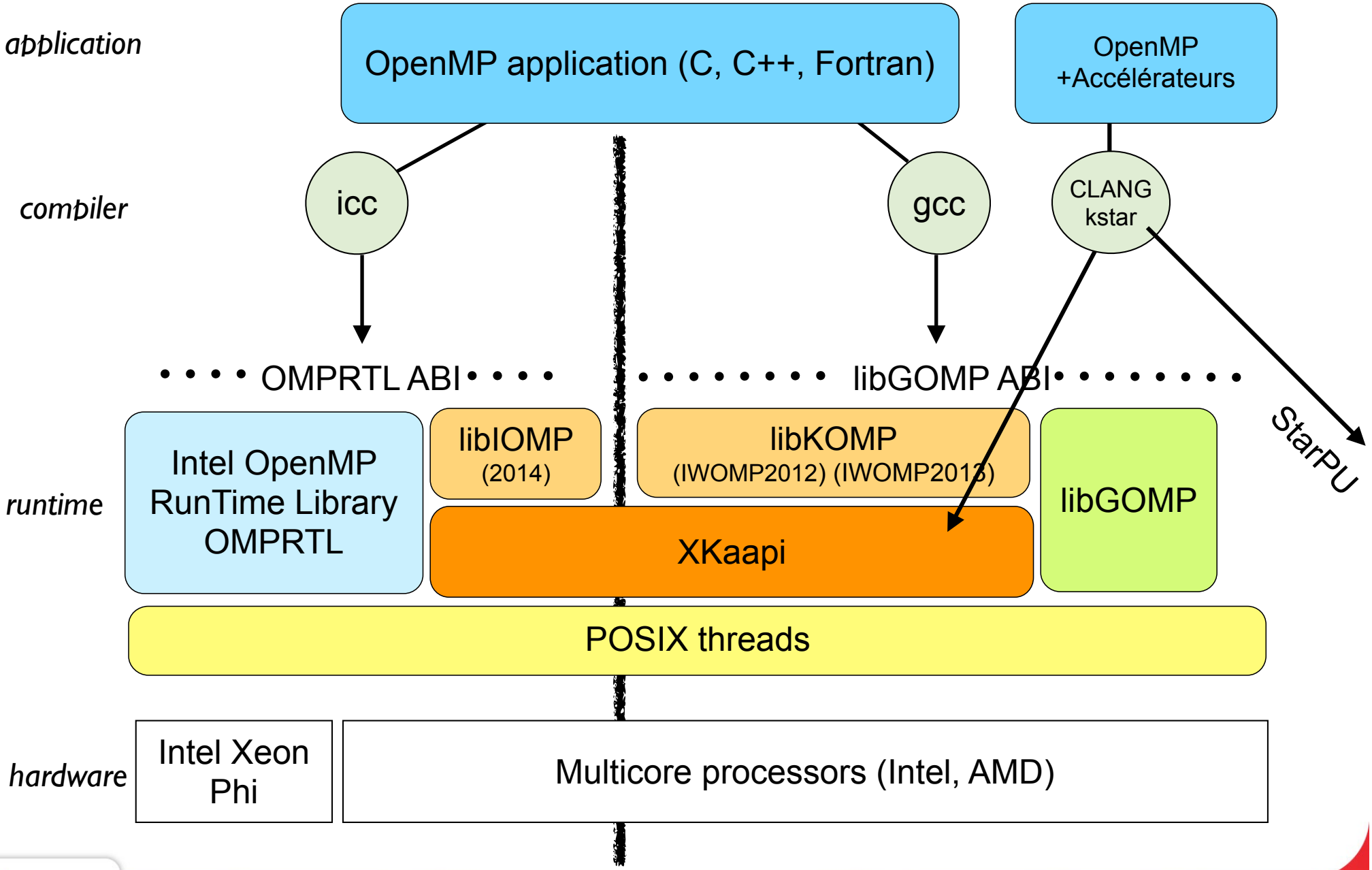
Utilisation à travers OpenMP



Utilisation à travers OpenMP



Utilisation à travers OpenMP



Des questions ?

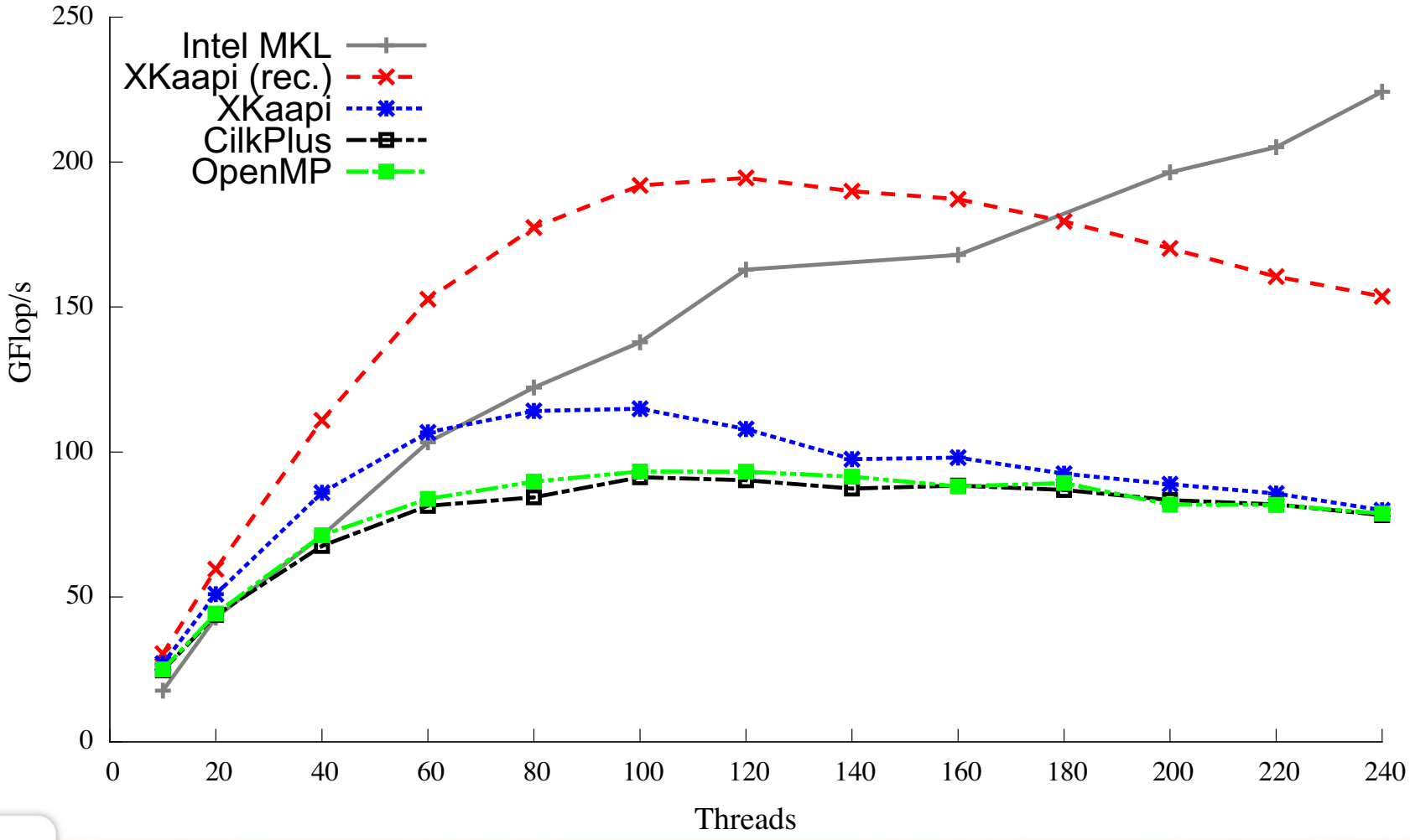
- ✓ http://kaapi.gforge.inria.fr
  - doc, download et mailing list !

- ✓ http://www.openmp.org

Inria

Cholesky factorization on Xeon Phi

- [SBAC-PAD 2013]
- N=8192
- Ongoing work to reduce cache usage



Sparse Cholesky factorization - EPX [CEA]

- ANR REPDYN, V. Faucher
- Comparison with OpenMP code
  - independent task in OpenMP => more synchronization points