OpenMP
www.openmp.org

- Une API pour la programmation parallèle en mémoire partagée
  - C, C++, Fortran
  - Portable
  - Porté par l'*Architecture Review Board* (Intel, IBM, AMD, Microsoft, Cray, Oracle, NEC…)
- Basé sur des annotations : #pragma omp *directive*
- et des fonctions: omp_*fonction*()
- Permet de paralléliser un code de façon plus ou moins intrusive un code
  - Plus on en dit plus on a de performance
  - Facile à mettre en œuvre par un non spécialiste… Trop facile ?
- Ne permet pas de créer ses propres outil de synchronisation
- https://computing.llnl.gov/tutorials/openMP/

Hello world !

```
int main()
{
  printf("bonjour\n");
  printf("au revoir\n") ;

  return EXIT_SUCCESS;
}
```

```
> gcc bon.c
>./a.out
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;

  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;

  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>OPENMP_NUMTHREADS=3 ./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>

int main()
{
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;

  return EXIT_SUCCESS;
}
```

```
> export OPENMP_NUMTHREADS=3
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Hello world !

```
#include <omp.h>
int main()
{
#pragma omp parallel num_threads(3)
  printf("bonjour\n");

 printf("au revoir\n") ;
  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

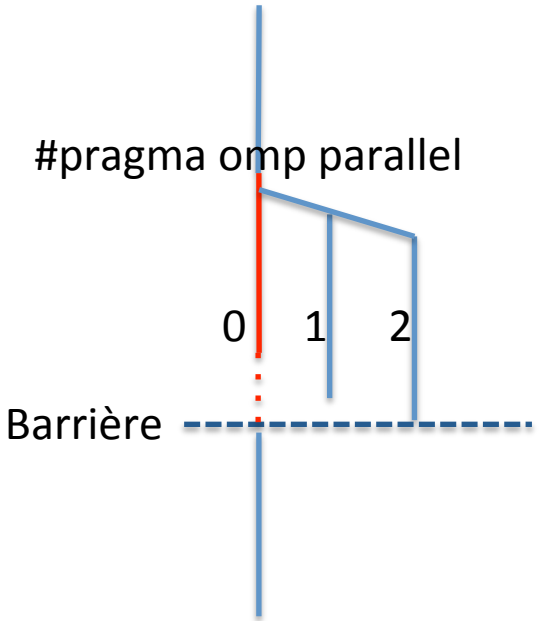
Hello world !

```
#include <omp.h>
int main()
{
omp_set_num_threads(3);
#pragma omp parallel
  printf("bonjour\n");

 printf("au revoir\n") ;
  return EXIT_SUCCESS;
}
```

```
> gcc –fopenmp bon.c
>./a.out
bonjour
bonjour
bonjour
au revoir
>
```

Parallélisme fork-join

- Un unique thread exécute séquentiellement le code de *main*.
- Lors de la rencontre d'un bloc parallèle, tout thread
  - crée une *équipe* de threads
  - la charge d'exécuter une fonction correspondant au bloc parallèle
  - rejoint en tant que *maître* l'équipe.
- À la fin du bloc parallèle
  - les threads d'une même équipe s'attendent au moyen d'une barrière *implicite*
  - les threads esclaves sont démobilisés
  - le thread maître retrouve son équipe précédente



Hello world !

```
#include <omp.h>
int main()
{
#pragma omp parallel
  {
      printf("bonjour\n");
      // barrière implicite
  }
printf("au revoir\n") ;
  return EXIT_SUCCESS;
}
```

Traduction en pthread

```
#pragma omp parallel
f();
```

```
{
pthread_t tid[K];
for(int i=1;i<k;i++)
 pthread_create(tid+i,0,f,i);
f(0);
for(int i=1;i<k;i++)
 pthread_join(tid[i]);
}
```

Calcul en parallèle de Y[i] = f(T,i) Pthread

```
#define NB_ELEM (TAILLE_TRANCHE *
NB_THREADS)
pthread_t threads[NB_THREADS];
double input[NB_ELEM], output[NB_ELEM];


void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

//  pthread_exit(NULL);
}
```

```
int main()
{
…
  for (int i = 0; i  < NB_THREADS; i++)
   pthread_create(&threads[I], NULL,
                        appliquer_f, (void *)i);

 for (int i = 0; i  < NB_THREADS; i++)
   pthread_join(threads[I], NULL);

 ...
}
```

Parallélisation efficace si équilibrée

Calcul en parallèle de Y[i] = f(T,i) OpenMP

```
#define NB_ELEM (TAILLE_TRANCHE *
NB_THREADS)
pthread_t threads[NB_THREADS];
double input[NB_ELEM], output[NB_ELEM];

void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

//  pthread_exit(NULL);
}
```

```
int main()
{
…
 #pragma omp parallel
   appliquer_f, (omp_get_thread_num());

 ...
}
```

Calcul en parallèle de Y[i] = f(T,i) distribution d'indices

```
#define NB_ELEM (TAILLE_TRANCHE *
NB_THREADS)
pthread_t threads[NB_THREADS];
double input[NB_ELEM], output[NB_ELEM];

/* void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

//  pthread_exit(NULL);
} */
```

```
int main()
{
…
 #pragma omp parallel
 #pragma omp for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 ...
}
```

Calculer Y[i] = f^k(T,i)
pthread

```
#define NB_ELEM (TAILLE_TRANCHE *
NB_THREADS)
pthread_t threads[NB_THREADS];
double input[NB_ELEM], output[NB_ELEM];

void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

//  pthread_exit(NULL);
}

int main()
{
```

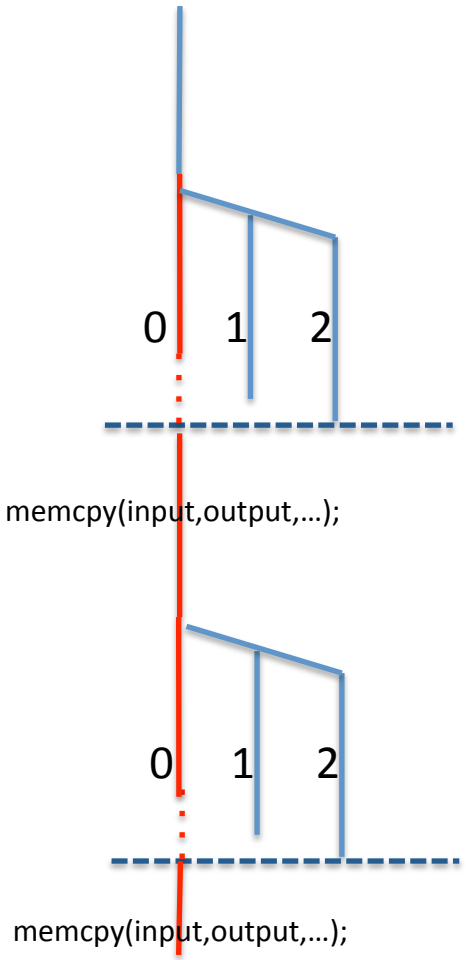
```
…
for (int etape = 0; etape < k; etape++)
{
  for (int i = 0; i  < NB_THREADS; i++)
   pthread_create(&threads[I], NULL,
                        appliquer_f, (void *)i);

 for (int i = 0; i  < NB_THREADS; i++)
   pthread_join(threads[I], NULL);

memcpy(input,output,…);
}
 ...
}
```

Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 ...
}
```

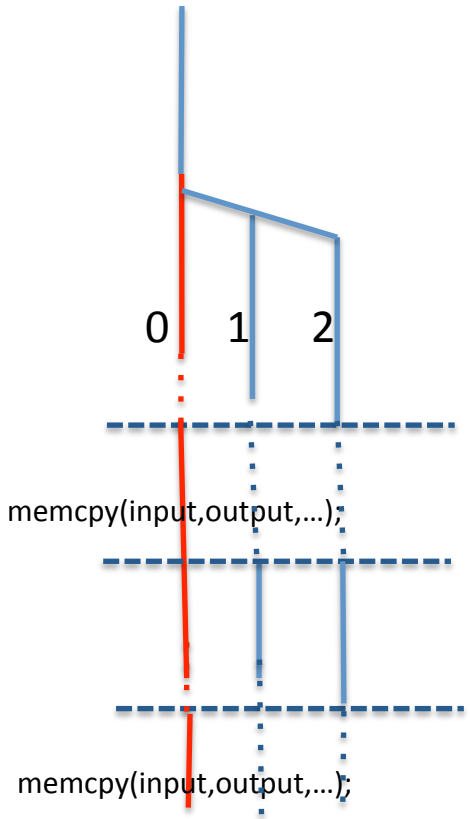
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 ...
}
```



Calculer Y[i] = f^k(T,i)
OpenMP

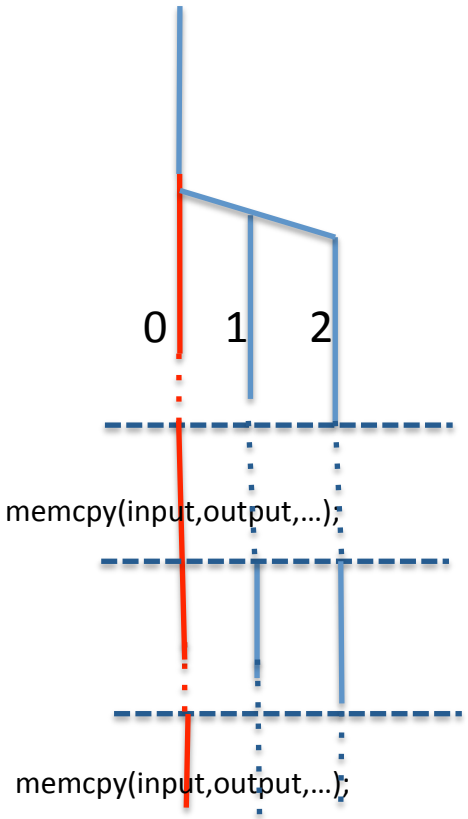
```
int main()
{
…
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

memcpy(input,output,…);
}
 ...
}
```



Réduire le coût de la parallélisation

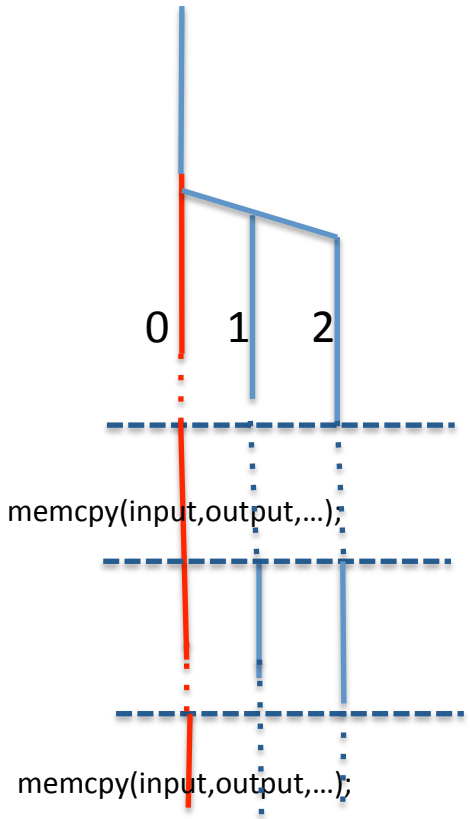
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);
  // barrière implicite
 if (omp_get_thread_num() == 0)
      memcpy(input,output,…);
  #pragma omp barrier
}
 ...
}
```



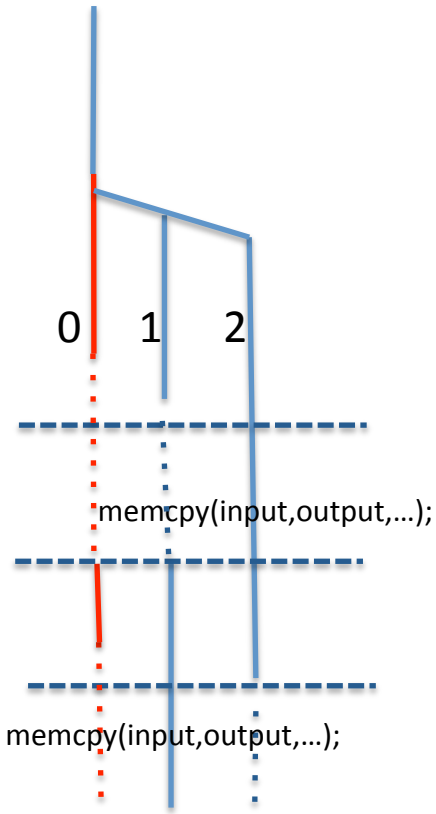
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);
  // barrière implicite
#pragma omp master
 memcpy(input,output,…);
 #pragma omp barrier
}
 ...
}
```



Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
…
#pragma omp parallel
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

#pragma omp single
 memcpy(input,output,…);
// barrière implicite
}
 ...
}
```



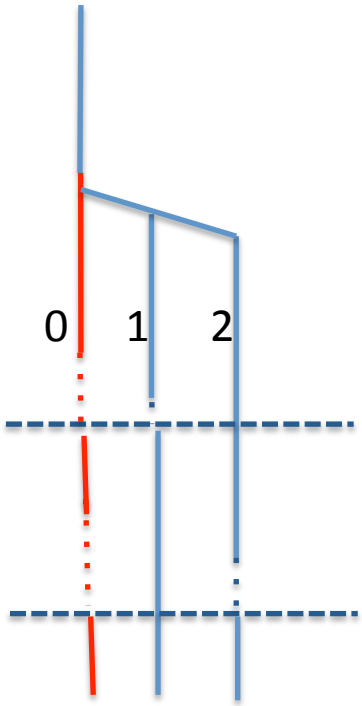
Calculer Y[i] = f^k(T,i)
Pthread

```
void appliquer_f(void *i)
{
 int debut = (int) i * TAILLE_TRANCHE;
 int fin = ((int) i+1) * TAILLE_TRANCHE;

 double *entree = input;
 double *sortie = output;

for(int etape=0; etape < k; etape++)
{
 for( int n= debut; n < fin; n++)
  sortie[n] = f(entree,n);
  echanger(entree,sortie);
  barrier_wait(&b); // attendre
}
```

```
}

int main()
{
…
for (int i = 0; i  < NB_THREADS; i++)
   pthread_create(&threads[I], NULL,
                        appliquer_f, (void *)i);

 for (int i = 0; i  < NB_THREADS; i++)
   pthread_join(threads[I], NULL);

…
}
```

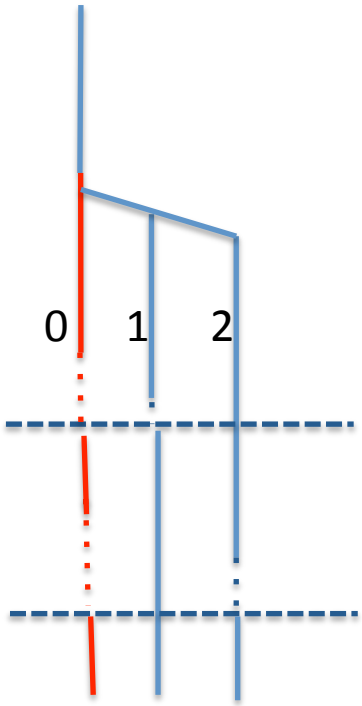
Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
  double *entree = input;
 double *sortie = output;
…
#pragma omp parallel private(entree, sortie)
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 echange (&entree,&sortie);

}
 ...
}
```



Calculer Y[i] = f^k(T,i)
OpenMP

```
int main()
{
  double *entree = input;
 double *sortie = output;
…
#pragma omp parallel firstprivate(entree, sortie)
for (int etape = 0; etape < k; etape++)
{
#pragma omp parallel for
for( int n= debut; n < fin; n++)
  output[n] = f(input,n);

 echange (&entree,&sortie);

}
 ...
}
```



omp parallel

#pragma omp parallel

- barrière implicite à la fin de la section parallèle
- nombre de threads : num_threads(n)
  - Dépendant de l'implémentation
  - Au maximum n threads exécuteront la section

- clauses sur le partage des variables
  - default (none | shared | private | firstprivate)
  - private(…) shared(…)
  - fisrtpivate(…) : la valeur de la variable est recopiée à l'initialisation du thread
  - lastprivate(…) : affecté par le thread exécutant la dernière affectation du programme séquentiel

#omp threadprivate()

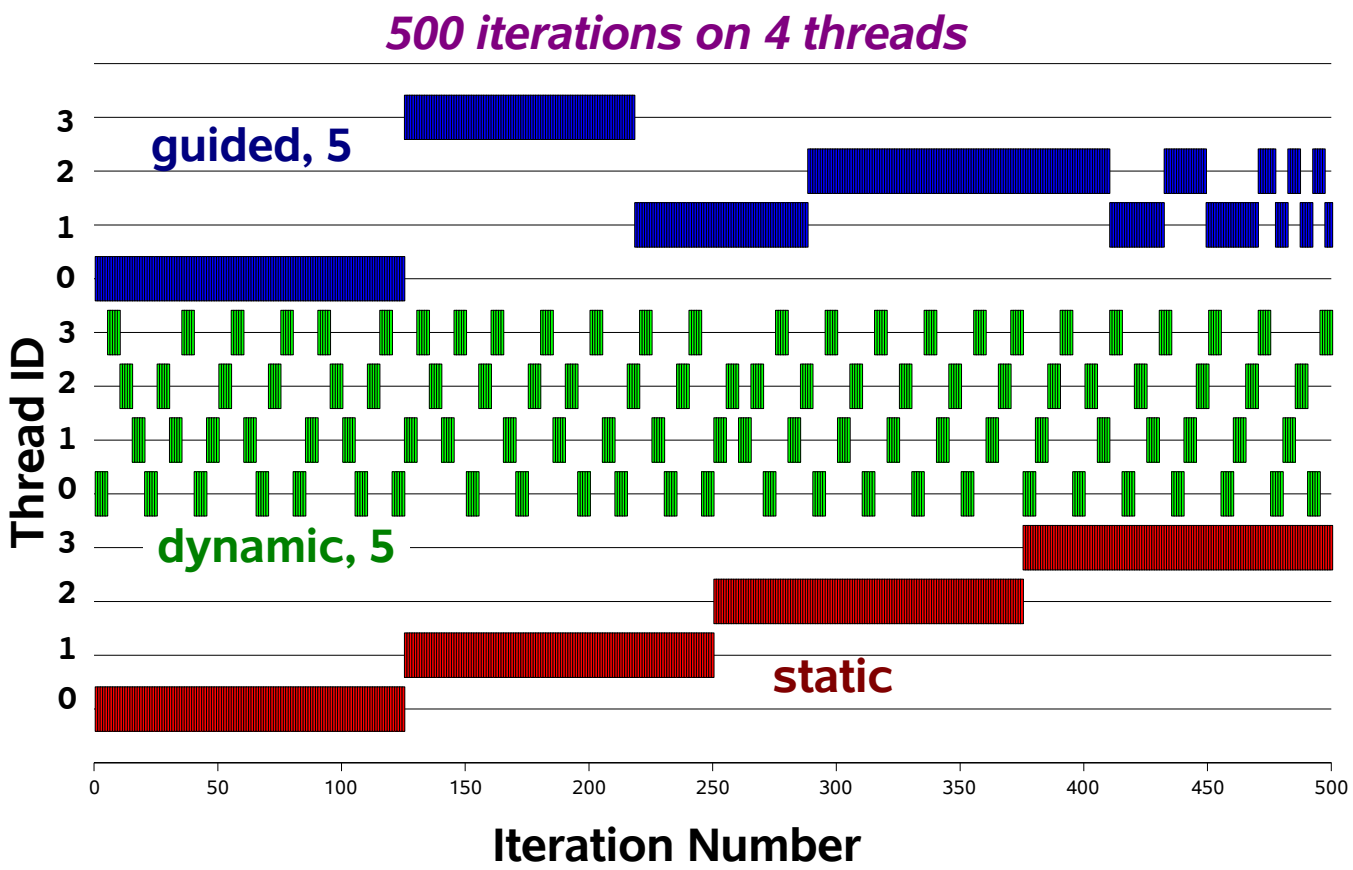
  - variable persistente conservée de section parallèle en section parallèle

omp for

#pragma omp for

- nowait
  - Suppression de la barrière implicite
- schedule(mode,taille)
  - (**static**) : distribution par bloc
  - (static,1) : distribution cyclique
  - (static,n) : cyclique par tranche de n
  - (**dynamic**,n) : à la demande par tranche de n
  - (**guided**,n) : à la demande, par tranches de tailles décroissantes = MAX(n, (nb indices restants) / nb_threads)
  - (**runtime**) : suivant la valeur de la variable OMP_SCHEDULE
  - (**auto**) : suivant le compilateur et/ou le support d'exécution

Schedule (d'après sun)



omp for collapse

```
#pragma parallel omp for collapse(2) schedule(runtime)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            t[i][j] = omp_get_thread_num();
```

OMP_SCHEDULE=static OMP_NUM_THREADS=5 ./a.out

```
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
```

OMP_SCHEDULE=guided OMP_NUM_THREADS=4

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3
3 3 3 3 3 2 2 2 2 1 1 1 3 3 0 0 2 1 3 3
```

OMP_SCHEDULE=dynamic,2 OMP_NUM_THREADS=5

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 1 1 0 0 1 1 0 0 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 1 1 3 3 0 0 1 1 2 2 0 0 1 1
0 0 3 3 0 0 2 2 1 1 3 3 0 0 2 2 4 4 1 1
0 0 3 3 2 2 4 4 0 0 1 1 3 3 2 2 0 0 4 4
1 1 3 3 0 0 2 2 4 4 1 1 3 3 0 0 2 2 4 4
1 1 0 0 3 3 2 2 4 4 0 0 1 1 3 3 2 2 0 0
4 4 1 1 3 3 0 0 2 2 4 4 1 1 0 0 3 3 2 2
4 4 0 0 1 1 3 3 2 2 0 0 4 4 1 1 3 3 0 0
2 2 4 4 1 1 0 0 3 3 2 2 4 4 0 0 1 1 3 3
2 2 0 0 4 4 1 1 3 3 2 2 0 0 4 4 1 1 3 3
0 0 2 2 4 4 1 1 0 0 3 3 2 2 4 4 0 0 1 1
3 3 2 2 0 0 4 4 1 1 3 3 0 0 2 2 4 4 1 1
0 0 3 3 2 2 4 4 0 0 1 1 3 3 2 2 0 0 4 4
1 1 3 3 0 0 2 2 4 4 1 1 0 0 3 3 2 2 4 4
0 0 1 1 3 3 2 2 0 0 4 4 1 1 3 3 0 0 2 2
4 4 1 1 0 0 3 3 3 3 2 2 4 4 0 0 1 1 3 3
2 2 0 0 4 4 1 1 3 3 0 0 2 2 4 4 1 1 0 0
3 3 2 2 4 4 0 0 1 1 3 3 2 2 0 0 4 4 1 1
```

Paralléliser le jeu de la vie

```
int T[2][DIM][DIM];
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;

         for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                        T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                        if (T[out][i][j] > 0)
                                nb_cellules++;
                }
      printf("%d => %d", etape, nb_cellules);

}
```

Paralléliser le jeu de la vie

```
#pragma omp parallel shared(nb_cellules) private(in,out)
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;
#pragma omp for collapse(2)
         for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                      T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                      if (T[out][i][j] > 0)
                            nb_cellules++;
                }
#pragma omp single
     printf("%d => %d", etape, nb_cellules);
}
```

Paralléliser le jeu de la vie

```
#pragma omp parallel shared(nb_cellules) private(in,out)
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;
#pragma omp for collapse(2)
          for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                      T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                      if (T[out][i][j] > 0)
#pragma omp critical
                                nb_cellules++;
                }
#pragma omp single
     printf("%d => %d", etape, nb_cellules);
}
```

Paralléliser le jeu de la vie

```
#pragma omp parallel shared(nb_cellules) private(in,out)
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;
#pragma omp for collapse(2)
          for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                      T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                      if (T[out][i][j] > 0)
#pragma omp atomic
                                nb_cellules++;
                }
#pragma omp single
     printf("%d => %d", etape, nb_cellules);
}
```

Critical vs atomic

#pragma omp critical [*identificateur*]

- Sans identificateur un mutex par défaut est utilisé
- Pas de barrière implicite

- #pragma atomic
  - Remplacé au besoin par un critical

```
#pragma omp parallel for collapse(2)
    for (i = 0; i < 2000; i++)
     for (j = 0; j < 2000; j++)
#pragma omp atomic
        x++;
```

Nb threads	critical	atomic
sans openMP	0,018	0,018
1	0,2	0,15
2	1,7	0,7
3	2,4	0,9
6	6	1,2
12	12	1,2
24	24	1,2
48	47	1,2

Paralléliser le jeu de la vie

```
#pragma omp parallel shared(nb_cellules) private(in,out)
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in;
        out = 1 - out;
        nb_cellules = 0;
#pragma omp for collapse(2) reduction(+:nb_cellules)
         for(i=1; i < DIM-1; i++)
                for(j=1; j < DIM-1; i++)
                {
                      T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                      if (T[out][i][j] > 0)
                            nb_cellules++;
                }
#pragma omp single
     printf("%d => %d", etape, nb_cellules);
}
```

Paralléliser le jeu de la vie

```
#pragma omp parallel shared(nb_cellules) private(in,out)
for(etape = 0; etape < ETAPE; etape++)
{
        in = 1-in; out = 1 – out; nb_cellules1 = 0; nb_cellules2 = 0;
#pragma omp for collapse(2) reduction(+:nb_cellules1,nb_cellules2)
          for(i=1; i < DIM-1; i++)
                  for(j=1; j < DIM-1; i++) {
                          if(etape%2)
                                  T[out][i][j] =f(T[in][i][j], T[in][i-1][j], ...)
                                  if (T[out][i][j] > 0)
                                  nb_cellules1++;
                          else …
                  }
#pragma omp single nowait
        if(etape % 2) {
                  printf("%d => %d", etape, nb_cellules1);
                  nb_cellules1=0;
        } else { ….
}
```

OpenMP détails
sections

```
#pragma omp sections
{
  #pragma omp section
     {
      ...
     }

     #pragma omp section
     {
     ...
     }
}
```

```
#pragma omp for schedule (dynamic)
for(i = 0; i < = S; i++)
  switch(i)
     {
       case 0: … ; break ;
       case 1: … ; break ;
       …
       case S: … ; break ;
       default : abort();
     }
}
```

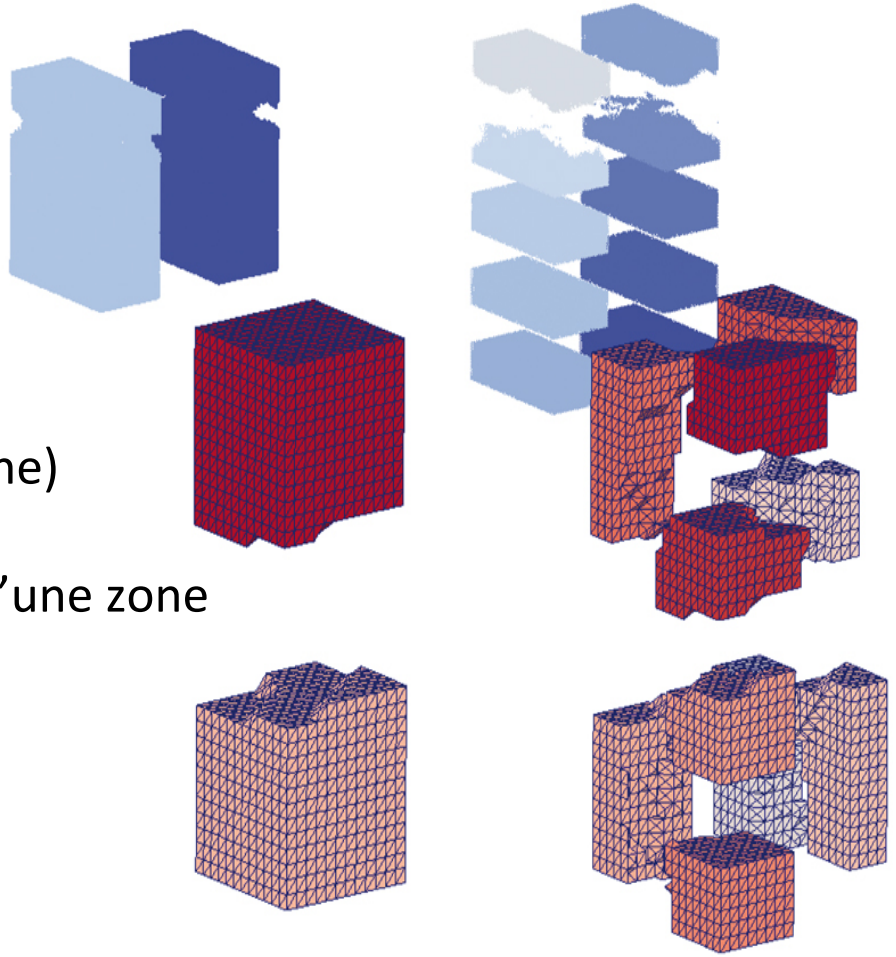
OpenMP détails

- clause nowait
  - Supprime la barrière par défaut
    - for, single, section,…
  - La mémoire n'est plus « synchronisée » à la fin de la construction
    - #pragma omp flush()

Parallélisme imbriqué

- Imbriquer des équipes de threads

```
#omp parallel num_threads(externe)
{
// distributions des zones de travail

#omp parallel for num_threads(interne)
  {
  // parallélisation du travail au sein d'une zone
   }
```



Parallélisme imbriqué

- Exprimer plus de parallélisme
  - Adapter le parallélisme au problème
  - Méthode récursive « diviser pour régner »

- Difficultés:
  - Surcoût à la création lié à la création / d'équipe
  - Stabilité d'exécution (support d'exécution, ordonnanceur du système)
  - Qualité du partionnement des threads

Parallélisme imbriqué
Qualité du support d'exécution

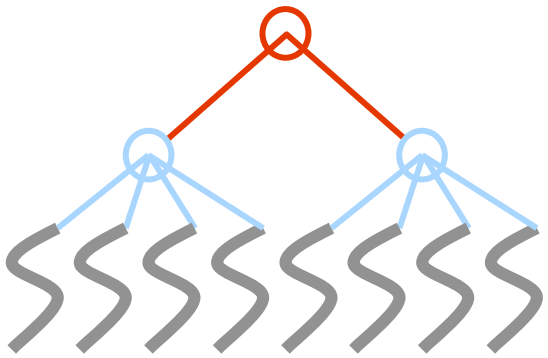
	GOMP 3	ICC	Forest
atomic	0,52	0,87	0,49
barrier	75,51	26,98	27,56
critical	12,90	39,01	4,13
for	80,44	28,17	27,33
lock	4,69	4,41	4,06
parallel	3209,75	304,94	171,66
parallel for	3222,49	311,58	170,56
reduction	3220,41	454,20	171,58

Benchmark Nested-EPCC : surcout (µs) de l'invocation des mot-clés OpenMP en contexte de parallélisme imbriqué, sur une machine à 16 coeurs

Parallélisme imbriqué
le benchmark BT-MZ

- Parallélisation à deux niveaux
  - Plusieurs simulations sur des zones différentes
  - Plusieurs threads pour traiter chacune des zones
  - Imbrication de régions parallèles OpenMP
  - Allocation mémoire: *first-touch*

- Parallélisme externe irrégulier
  - Différentes charges de travail selon les zones
- Parallélisme interne régulier
  - Les threads d'une même zone ont la même quantité de travail à effectuer



Arbre de threads obtenu lors d'une exécution « 2x4 » du benchmark BT-MZ

BT-MZ: Résultats expérimentaux

Outer x Inner	GOMP 3	Intel	Cache	
			Original	Info de charge
4 x 4	9.4	13.8	14.1	14.1
16 x 1	**14.1**	**13.9**	14.1	14.1
16 x 4	11.6	6.1	14.1	14.9
16 x 8	11.5	4.0	14.4	**15.0**
32 x 1	12.6	10.3	13.5	13.8
32 x 4	11.2	3.4	14.3	14.8
32 x 8	10.9	2.8	**14.5**	14.7

Accélérations obtenues sur la machine Kwak avec la classe C du benchmark BT-MZ

Parallélisme imbriqué
Diviser pour régner

- Extraire beaucoup de parallélisme
- Pouvoir traiter des pb irréguliers

```
void fun ( int p)
{
    #pragma omp parallel for
    for(i …)
        …
        fun(p+1) ;

}
```

Parallélisme imbriqué
Diviser pour régner

- Extraire beaucoup de parallélisme
- Pouvoir traiter des pb irréguliers

```
void fun ( int p)
{
    #pragma omp parallel for if (p < PROFMAX)
    for(i …)
        …
        fun(p+1) ;

}
```

Qsort

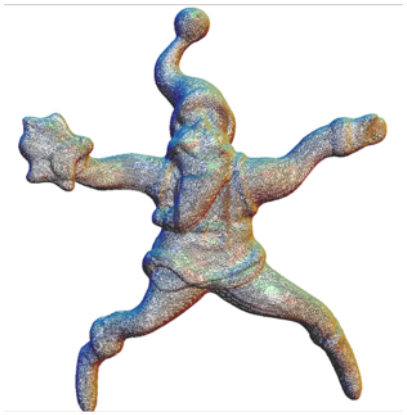
http://wikis.sun.com/display/openmp

```
void quick_sort (int p, int r, float *data)
{
   if (p < r) {
         int q = partition (p, r, data);
         #pragma omp parallel sections firstprivate(data, p, q, r)
         {
            #pragma omp section
            quick_sort (p, q-1, data, low_limit);
            #pragma omp section
            quick_sort (q+1, r, data, low_limit);
         }
      }
   }
}
```

Parallélisme imbriqué
Diviser pour régner

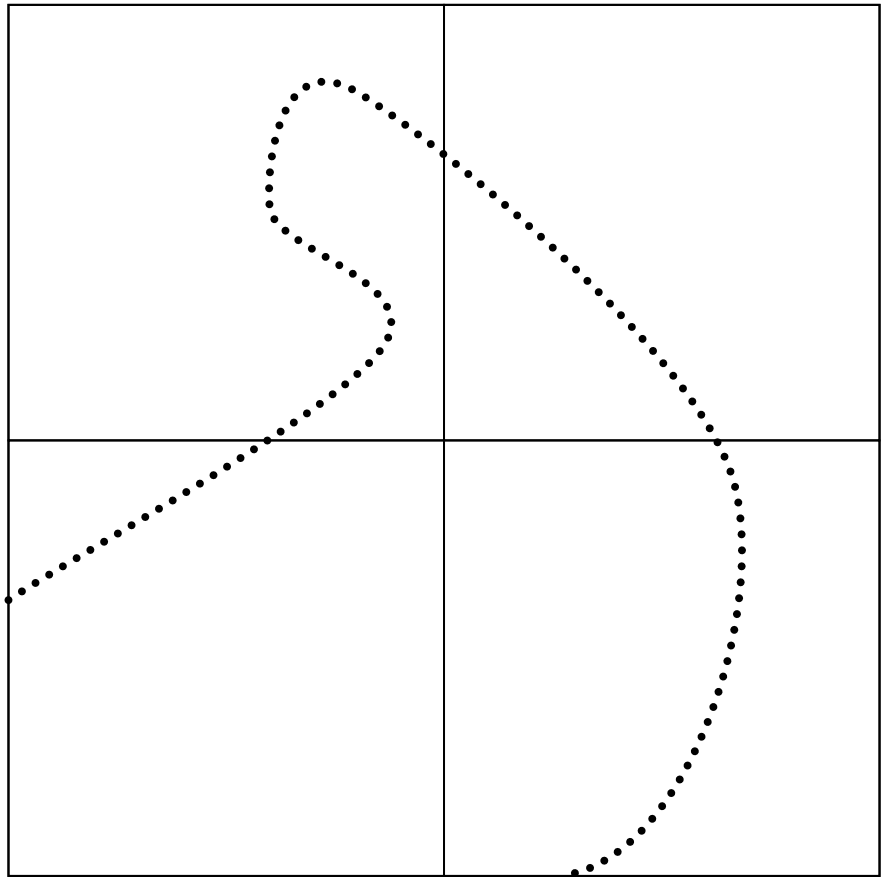
Ordonnancement de l'application MPU

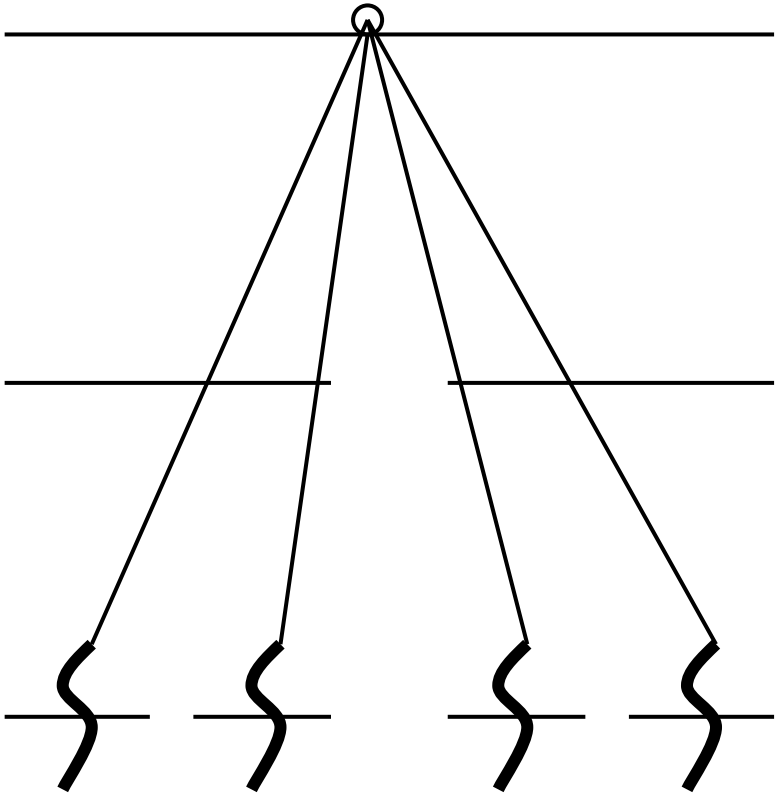
– Objectif : trouver une fonction mathématique approximant la surface d'un objet défini par un nuage de points



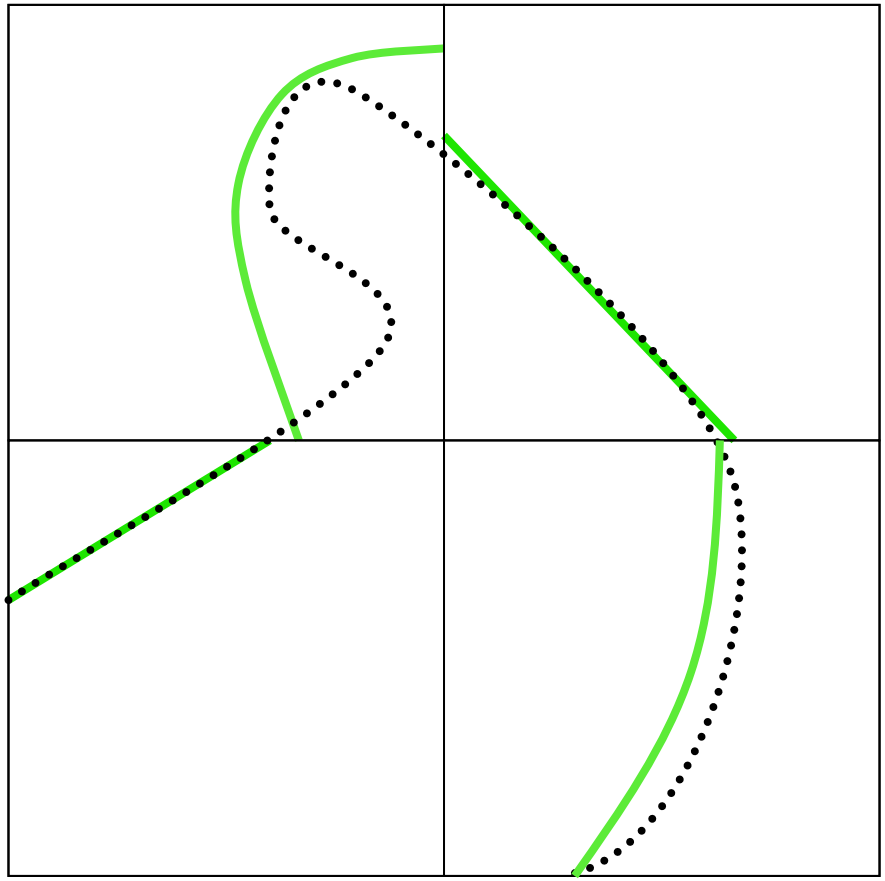
Exécution avec *Cache*

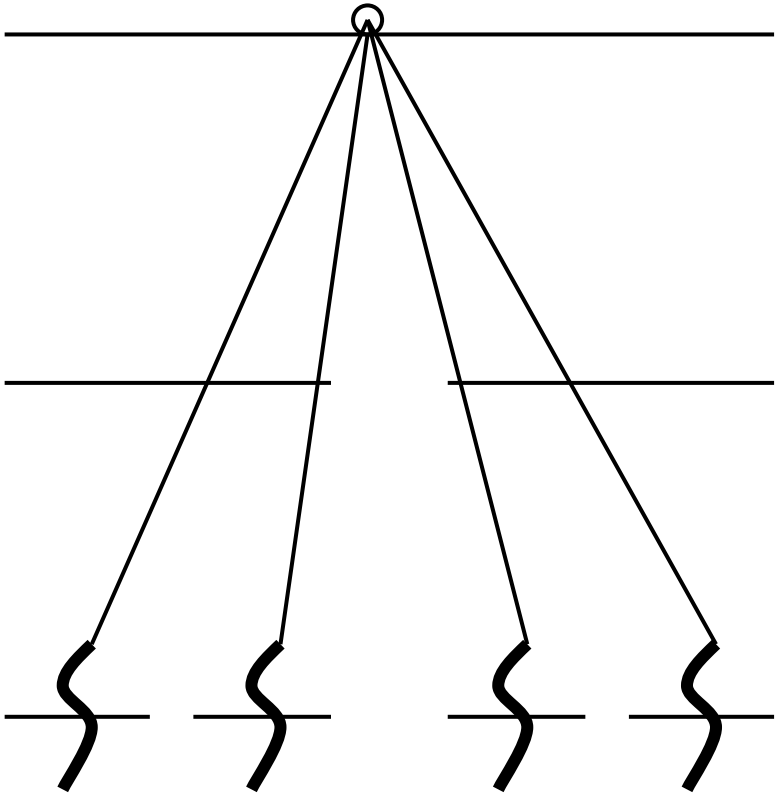
- Distribution gloutonne



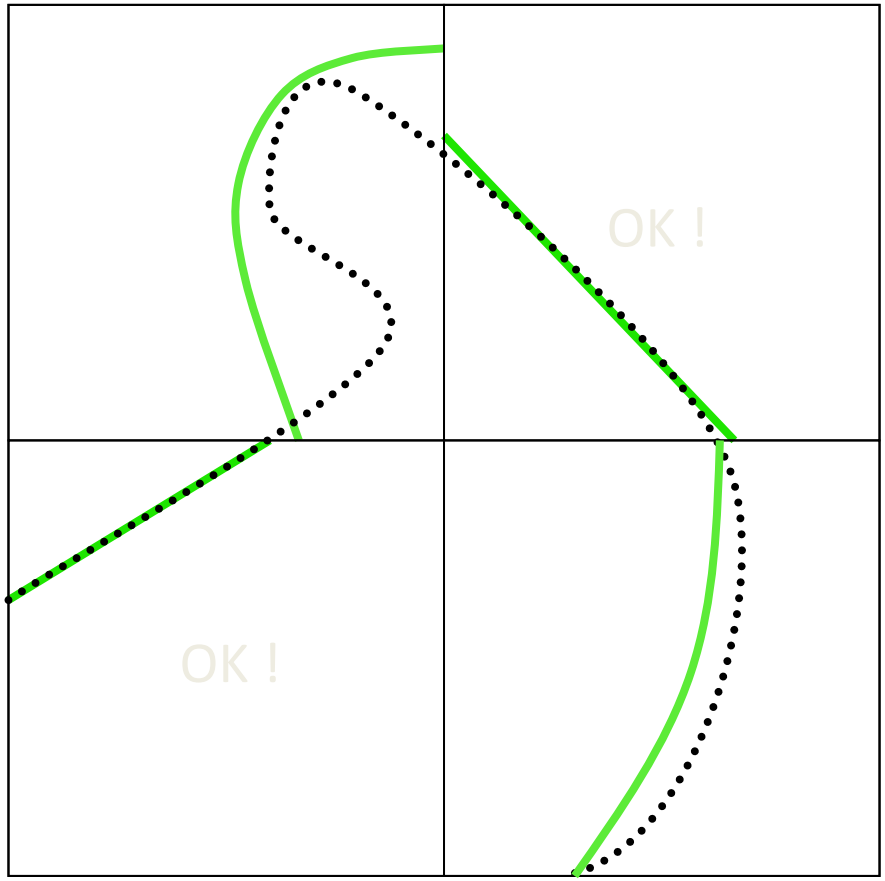


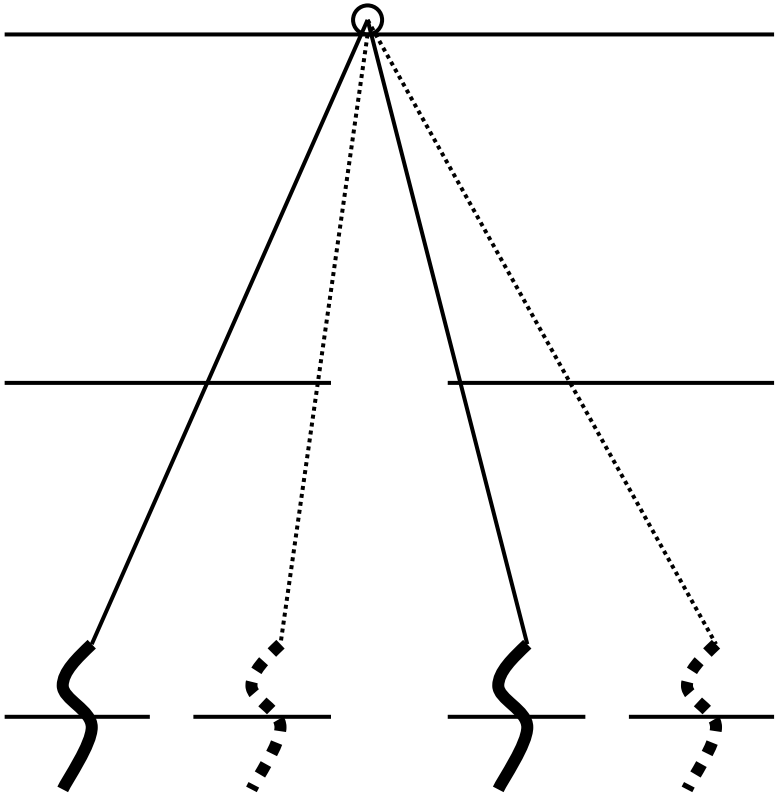
Exécution avec *Cache*



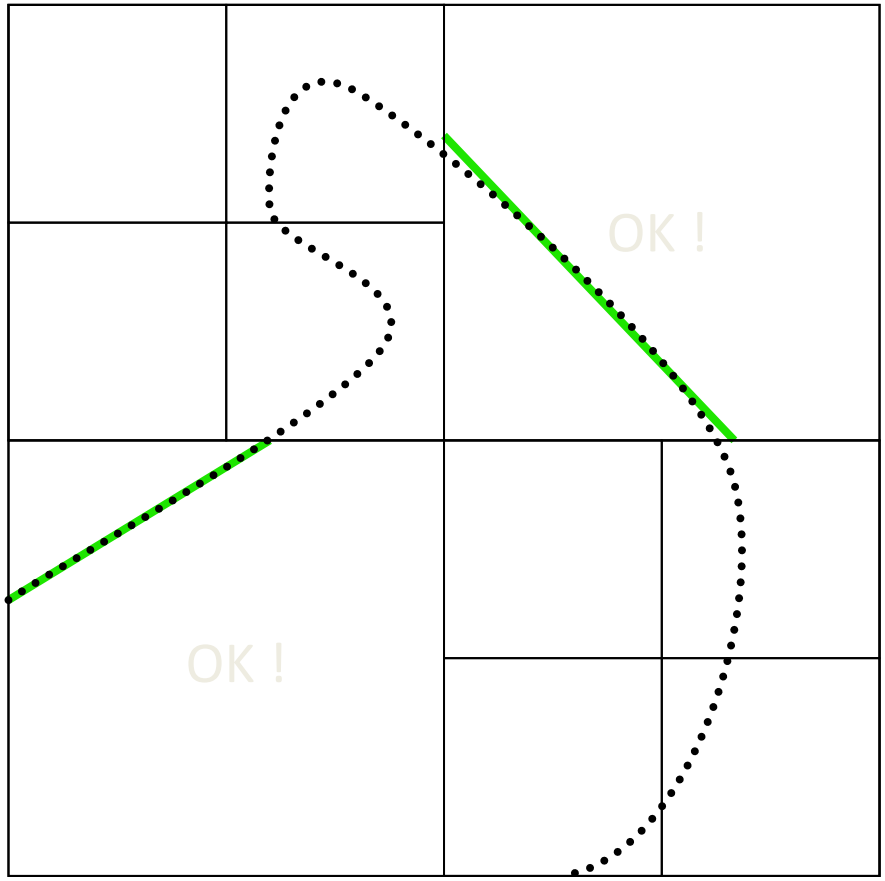


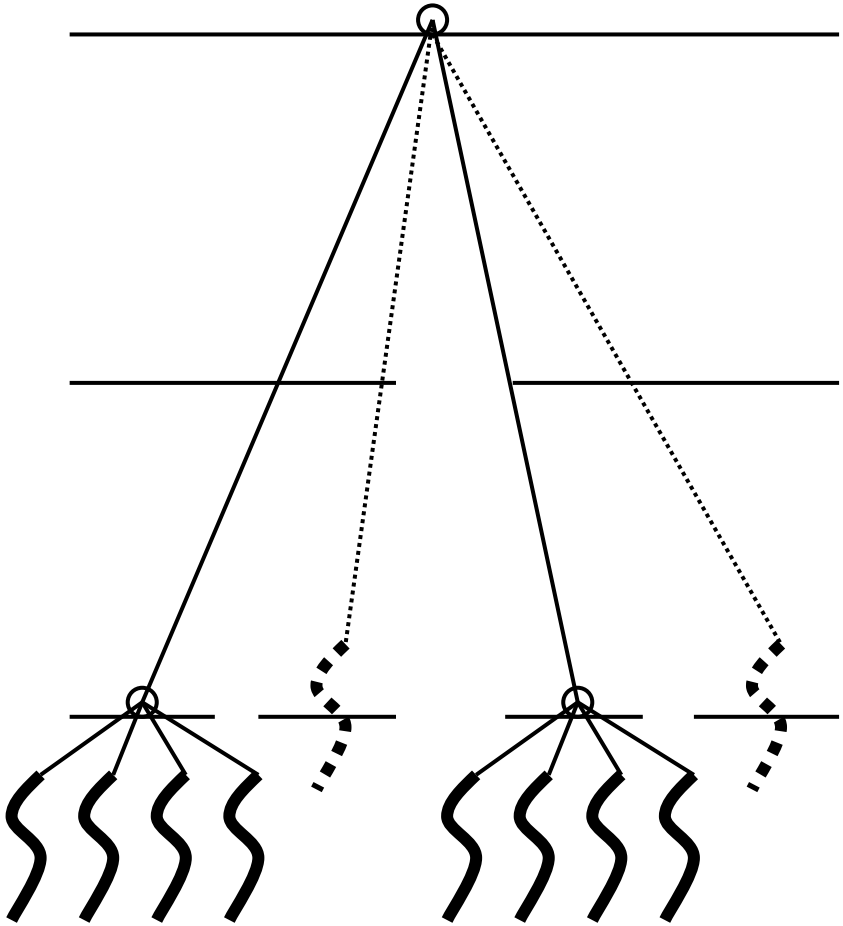
Exécution avec *Cache*





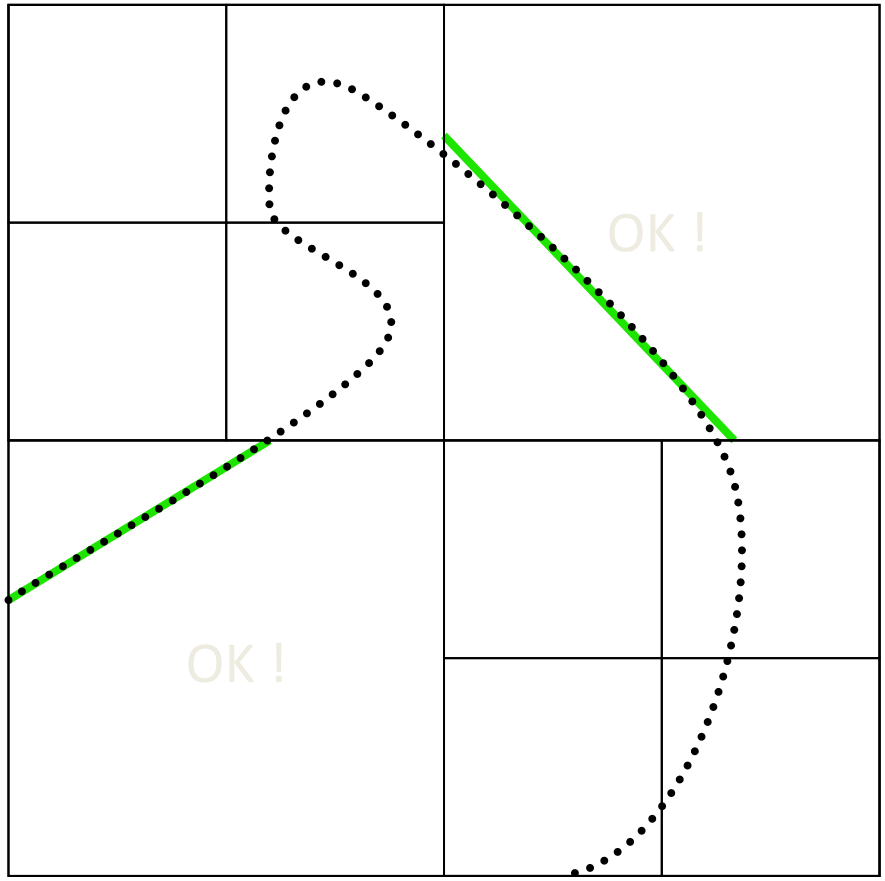
Exécution avec *Cache*

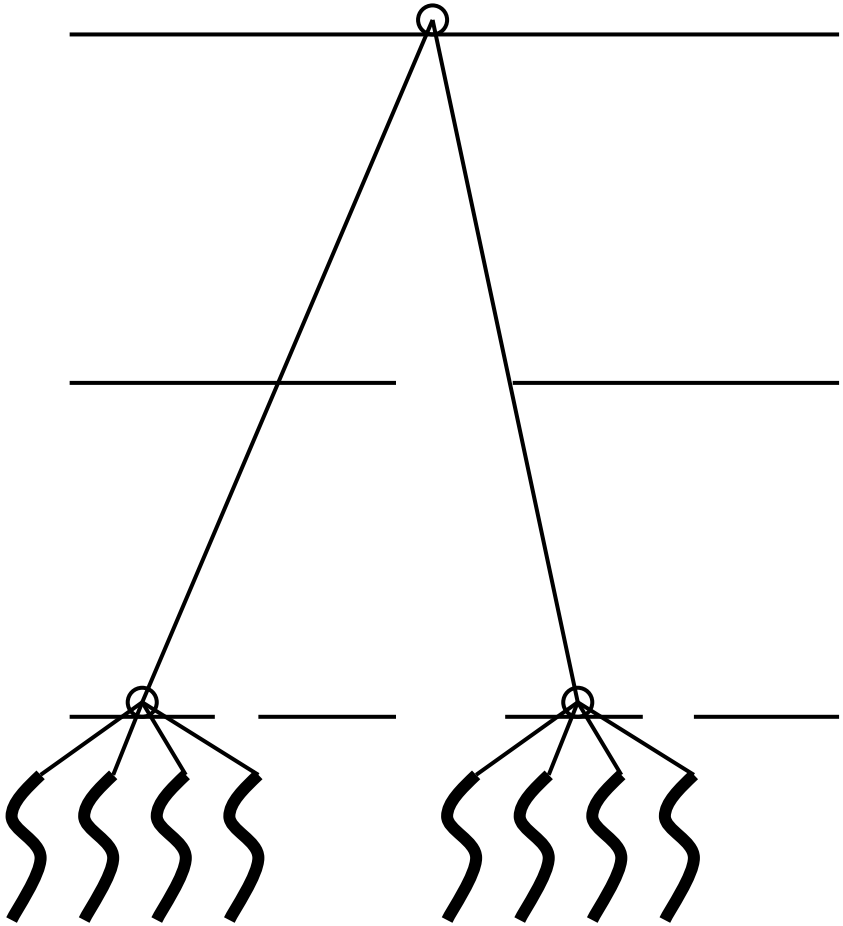




Exécution avec *Cache*

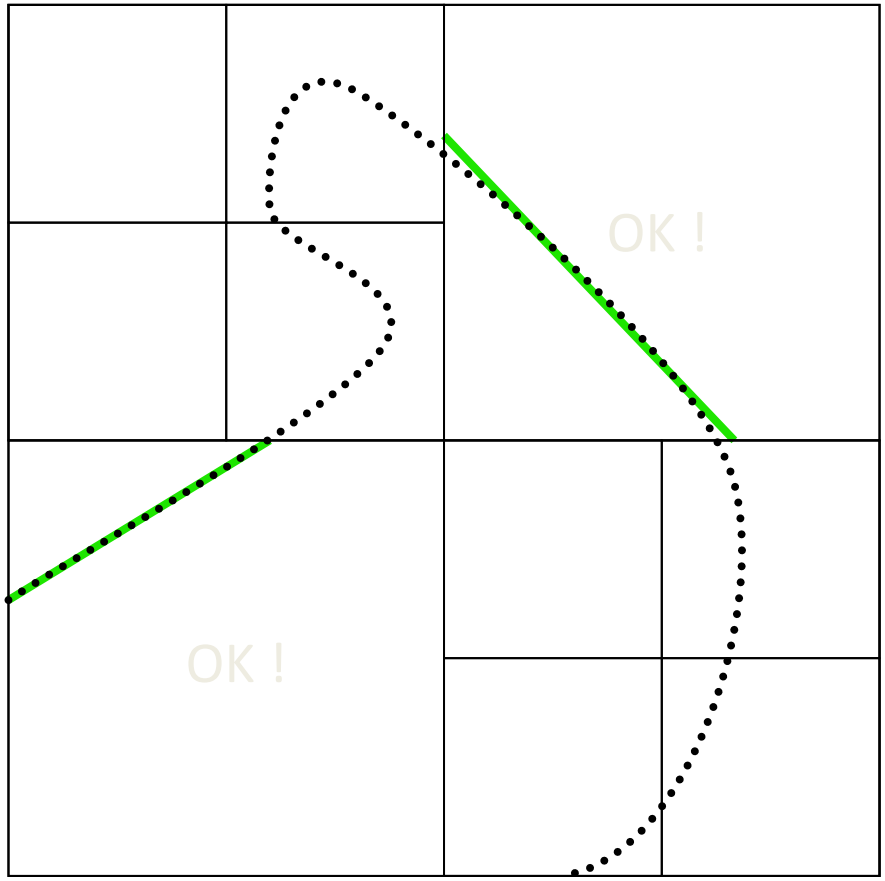
- Vol de travail de proche en proche pour rééquilibrer la charge

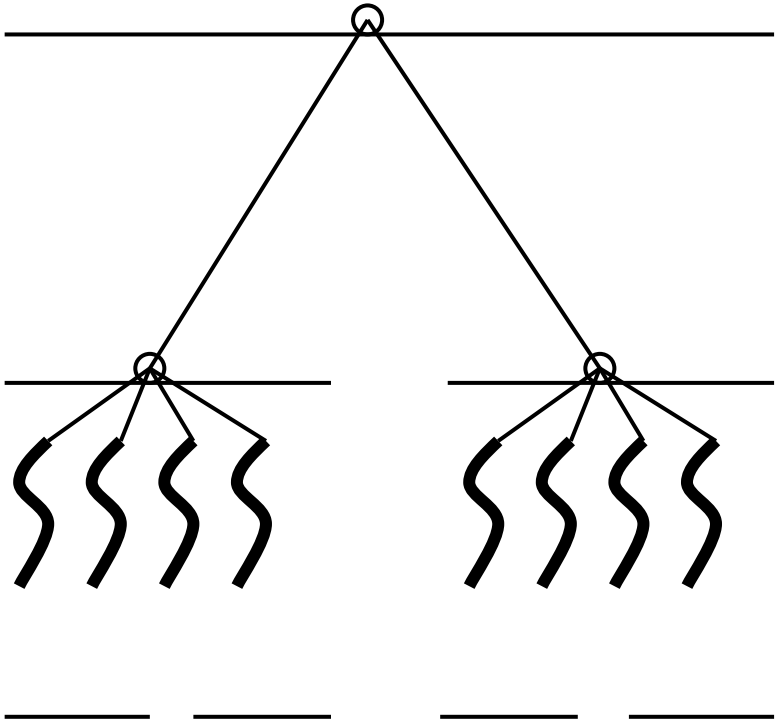




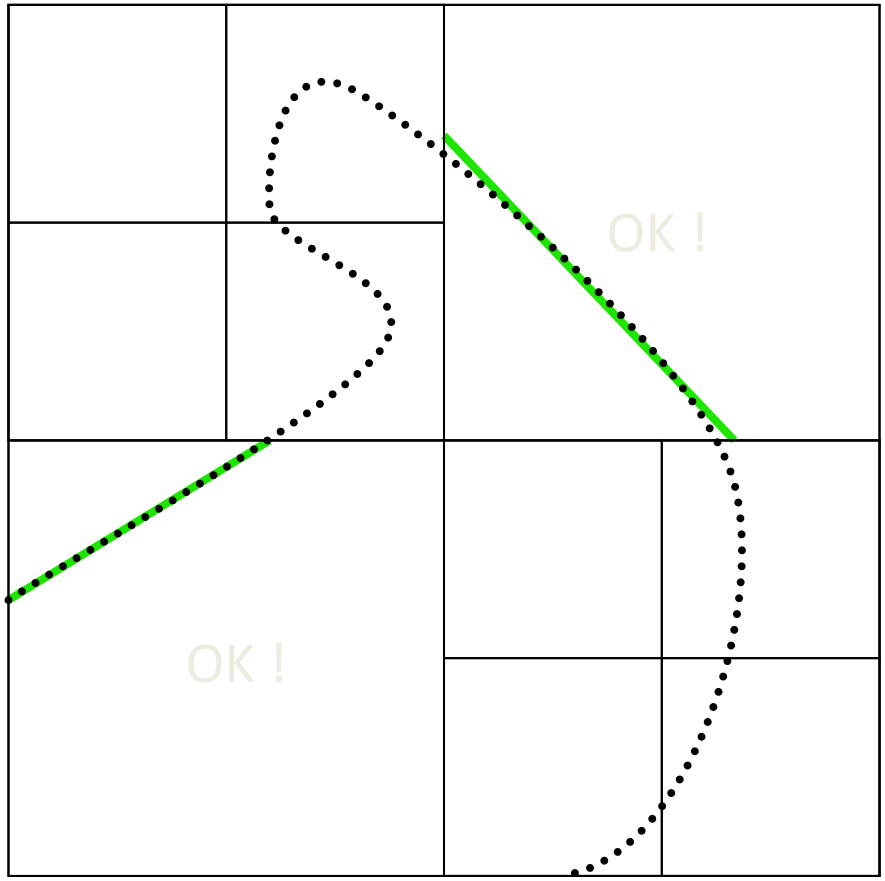
Exécution avec *Cache*

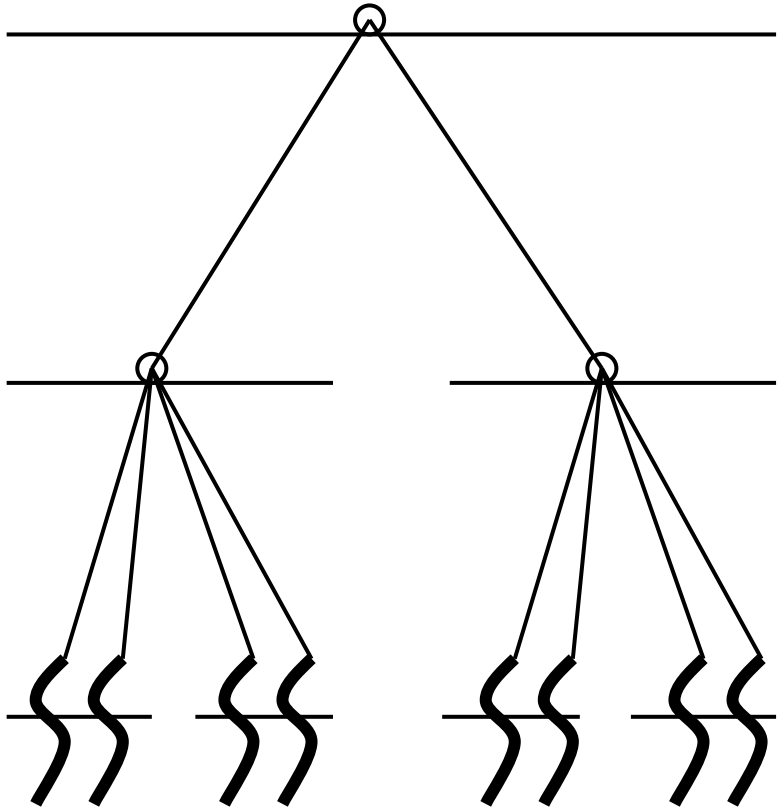
- Vol de travail de proche en proche pour rééquilibrer la charge





Exécution avec *Cache*





MPU : Accélération sur 16 cœurs

- Exécution sur un nuage de 437644 points
- 101185 créations de threads
- Profondeur maximale de récursion : 15