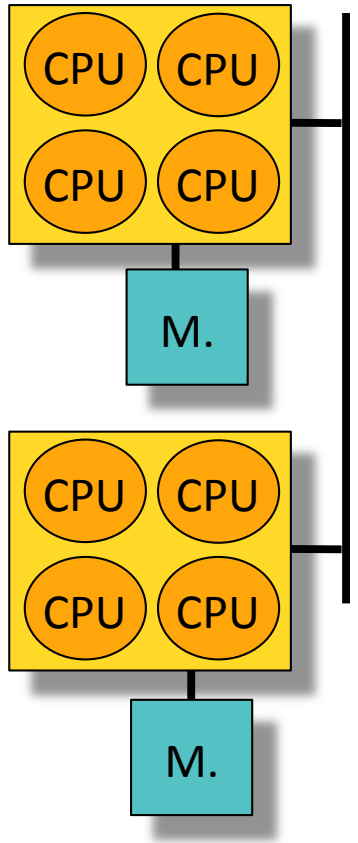
Approche mémoire partagée

Threads

- Paradigme de l'approche
- Objets exécutés par les processeurs

- threads vs processus,
  - un thread possède :
    - Ses registres, sa pile, des données propres (errno)
    - Son masque de signaux, ses signaux pendants
    - Des propriétés (mode d'ordonnancement, taille de la pile)
  - ✓ Accès aux ressources partagées
    - Mutex et Condition vs semaphore posix + mmap
  - ✓ Faible coût relatif des opérations de base (création,…)
  - − Pas de protection mémoire



API pthread

https://computing.llnl.gov/tutorials/pthreads/

- Avantages d'une interface bas niveau
  - On peut contrôler presque tout
  - On peut inventer ses propres mécanismes de synchronisation
- API POSIX
  - Création et gestion de la terminaison des threads
    - Gestion des attributs (taille de la pile, politique d'ordonnancement, placement)
  - Gestion des mutex et des rwlock
    - rwlock : plusieurs lecteurs à la fois / un seul écrivain
  - Gestion des conditions
  - Gestion des variables spécifiques à un thread
- API C11
  - Intégration des threads au langage (simplifications)
  - Gestion des données (remplace des *intrinsic/builtin functions* de gcc)
    - Variables atomiques (choix du modèle de cohérence mémoire)
    - Alignement des données

Calcul en parallèle de Y[i] = f(T,i)

```
for( int n= debut; n < fin; n++)
        Y[n] = f(T,n)
```

- Objectif : paralléliser avec la plus grande efficacité
  - Faire en sorte que les threads terminent tous en même temps
- 2 grandes approches:
  - statique : on distribue les indices au moment de la création des threads
  - dynamique : on distribue les indices au fur et à mesure

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

```
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 lorsque le travail est réparti de façon équilibré

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

Admettons que les n premiers indices concentre la moitié du travail :

=> speed-up limité à 2 si un seul thread traite ces n premiers indices

- Approche dynamique pour limiter ce risque
  - Utiliser un distributeur d'indices

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int indice = 0;

int
obtenir_indice()
{
  int k;
  pthread_mutex_lock(&mutex);
  k = indice++;
  pthread_mutex_unlock(&mutex);
  return (indice > NB_ELEM) ? -1 : indice;
}

void appliquer_f(void *i)
{
 int n;
 while( (n= obtenir_indice()) > 0)
  output[n] = f(input,n);
}
```

Calcul en parallèle de Y[i] = f(T,i)

Admettons que les n premiers indices concentre la moitié du travail :

=> speed-up limité à 2 si un seul thread traite ces n premiers indices

- Approche dynamique pour limiter ce risque
  - Utiliser un distributeur d'indices

Intuitivement le temps passé en section critique doit être négligeable par rapport au temps de calcul.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int indice = 0;

int
obtenir_indice()
{
  int k;
  pthread_mutex_lock(&mutex);
  k = indice++;
  pthread_mutex_unlock(&mutex);
  return (indice > NB_ELEM) ? -1 : indice;
}

void appliquer_f(void *i)
{
 int n;
 while( (n= obtenir_indice()) > 0)
  output[n] = f(input,n);
}
```

Coût de la synchronisation

Impact du temps passé en section critique sur le speed up

Code d'un thread

```
int k;
for (i=0; i < 100; i++)
   {
        for (j = 1; j < (100000 * (100 - pourcents)) / nb_thr; j++)
              k+=j;

        pthread_mutex_lock(&m);
         for (j = 1; j < (100000  * pourcents) /nb_thr; j++)
            k+=j;
        pthread_mutex_unlock(&m);
}
```

48 cœurs AMD

% section critique	speedup
0,0%	47,0
1,0%	45,6
2,0%	43,8
3,0%	31,4
4,0%	24,2
5,0%	19,9
6,0%	16,3

Le temps passé en section critique doit être négligeable par rapport au temps de calcul.

Dans le cadre d'un schéma de calcul itératif il est préférable que la fraction du temps passé en section critique ne dépasse pas 1 / (nombre cœurs) par itération.

Applications (Annales)

- *Il s'agit de paralléliser le plus efficacement possible la boucle suivante (en modifiant au besoin le code):*

```
for(i=0 ; i < 1000 ; i++)
        s += f(i) ;
```

  - *En supposant que le temps de calcul de f(i) ne dépends pas de la valeur de i ;*
  - *En supposant que le temps de calcul de f(i+1) est toujours (très) supérieur à celui de f(i).*

Programme film

```
#define N 128
#define N1 1024
#define N2 1280
#define N1N2 (1280 * 1024)

typedef long matrix[N][N1][N2];

long int diff[N];

matrix film;
```

```
void difference(){
  for(int n=0; n < N-1; n++)
    for(int i=0; i < 1024; i++)
      for(int j=0; j < 1280; j++)
        if (film[n][i][j] != film[n+1][i][j])
          diff[n]++;
}
```

Parallélisation

- Optimiser les accès mémoire
  - Analyse du comportement du programme pour 3 images
    - Taille d'une image: 5 Mo
    - Taille du cache L3 : 8 Mo
    - Taille du cache L2 : 256 Ko
    - Taille du cache L1 : 32 Ko
    - 2 images ne peuvent tenir en même temps dans le cache

```
void difference(){
  for(int n=0; n < N-1; n++)
    for(int i=0; i < 1024; i++)
      for(int j=0; j < 1280; j++)
        if (film[n][i][j] != film[n+1][i][j])
          diff[n]++;
}
```

Parallélisation

- Optimiser les accès mémoire
  - Analyse du comportement du programme pour 3 images
    - Taille d'une image: 5 Mo
    - Taille du cache L3 : 8 Mo
    - 2 images ne peuvent tenir en même temps dans le cache

Supposons que le cache soit de type LRU

- film[1][0][0] est chargé pour être comparé à film[1][0][0]
- film[1][0][0] est évincé du cache L3 (à film + 4Mo)
- film[1][0][0] est rechargé pour être comparé à film[2][0][0]

```
void difference(){
  for(int n=0; n < N-1; n++)
    for(int i=0; i < 1024; i++)
      for(int j=0; j < 1280; j++)
          if (film[n][i][j] != film[n+1][i][j])
            diff[n]++;
}
```

Parallélisation

- Optimiser les accès mémoire
  - Analyse du comportement du programme pour 3 images
    - Taille d'une image: 5 Mo
    - Taille du cache L3 : 8 Mo
    - 2 images ne peuvent tenir en même temps dans le cache

Supposons que le cache soit de type LRU

- film[1][0][0] est chargé pour être comparé à film[1][0][0]
- film[1][0][0] est évincé du cache L3 (à film + 4Mo)
- film[1][0][0] est rechargé pour être comparé à film[2][0][0]

```
void difference(){
  for(int n=0; n < N-1; n++)
    for(int i=0; i < 1024; i++)
      for(int j=0; j < 1280; j++)
          if (film[n][i][j] != film[n+1][i][j])
            diff[n]++;
}
```

```
void difference(){
 for(int i=0; i < 1024; i++)
    for(int n=0; n < N-1; n++)
      for(int j=0; j < 1280; j++)
          if (film[n][i][j] != film[n+1][i][j])
            diff[n]++;
}
```

Placement des threads
GNU / LINUX

- Objectif : maitriser la machine
  - Empêcher les interventions contreproductive du système d'exploitation
    - Conserver le cache
  - Technique
    - Éviter d'utiliser plus de threads qu'il y a de cœurs
    - Mieux : fixer les threads
- Exemple : placer le thread i sur le cœur i modulo 16

```
int p[P];
pthread_t t[P];
pthread_attr_t attr[P];

for(i = 0; i<P; i++)
 {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(i%16,&cpuset);
  pthread_attr_init(&attr[i]);
  pthread_attr_setaffinity_np(&attr[i], sizeof cpuset, &cpuset);
  p[i]=i;
  pthread_create(&t[i],&attr[i],thread_function,&p[i]);
 }
```

Application au problème « film »

Machine Leger (salle 008)

	Séquentiel	Parallèle
original avec if	1073.8	**368.4**
Permutation n et i avec if	1023.8	150.5
Permutation sans if :	641.1	132.4
bloc de 1ko	700.3	103.0
bloc de 2ko	419.3	74.6
bloc de 4ko	300.6	81.5
bloc de 8ko	282.8	**69.0**
bloc de 16ko	**269.5**	**68.3**
bloc de 32ko	**273.9**	**66.5**
bloc de 64ko	**270.2**	**67.9**
bloc de 128ko	280.9	76.2
bloc de 256ko	287.5	77.1
bloc de 512ko	285.9	87.8
bloc de 1024ko	285.7	106.9

Application au problème « film »

Leger

original avec if	1073.8	**368.4**	
Permutation n et i avec if	1023.8	150.5	
Permutation sans if :	641.1	132.4	
bloc de 1ko	700.3	103.0	mauvaise utilisation TLB
bloc de 2ko	419.3	74.6	
bloc de 4ko	300.6	81.5	
bloc de 8ko	282.8	**69.0**	
bloc de 16ko	**269.5**	**68.3**	L1 saturé
bloc de 32ko	**273.9**	**66.5**	
bloc de 64ko	**270.2**	**67.9**	
bloc de 128ko	280.9	76.2	L2 saturé
bloc de 256ko	287.5	77.1	
bloc de 512ko	285.9	87.8	L3 complet
bloc de 1024ko	285.7	106.9	

Application au problème « film »

Machine Leger (salle 008)

	Séquentiel	statique	dynamique
original avec if	1073.8	**368.4**	**211**
Permutation n et i avec if	1023.8	150.5	159
Permutation sans if :	641.1	132.4	158
bloc de 1ko	700.3	103.0	148
bloc de 2ko	419.3	74.6	112
bloc de 4ko	300.6	81.5	90
bloc de 8ko	282.8	**69.0**	85
bloc de 16ko	**269.5**	**68.3**	**81**
bloc de 32ko	**273.9**	**66.5**	**78.4**
bloc de 64ko	**270.2**	**67.9**	**79**
bloc de 128ko	280.9	76.2	**83**
bloc de 256ko	287.5	77.1	90
bloc de 512ko	285.9	87.8	87
bloc de 1024ko	285.7	106.9	123

Application au problème « film »

Boursouf – AMD 48 cœurs

	Seq	48 c.	16 c.
original avec if	3506.9	905.3	846
Permutation n et i avec if	2178.9	468.3	440
Permutation sans if :	1825.0	465.5	437
bloc de 1ko	1055.8	280.2	270
bloc de 2ko	944.4	251.9	244
bloc de 4ko	885.8	239.8	230
bloc de 8ko	858.8	**237.0**	223
bloc de 16ko	840.2	239.7	**220**
bloc de 32ko	**834.4**	236.8	222
bloc de 64ko	1005.0	252.9	225
bloc de 128ko	1004.4	267.1	250
bloc de 256ko	1008.5	276.8	284
bloc de 512ko	809.9	270.1	268
bloc de 1024ko	621.5	350.6	341

Point sur l'équilibrage de charge

- Répartir algorithmiquement équitablement le travail entre les threads
  - Nécessaire mais pas suffisant
- Le débit mémoire peut limiter l'accélération
  - Nécessaire optimisation des accès mémoire
    - Un défaut en version séquentielle peut provoquer un embouteillage lors d'une exécution parallèle
    - Ce peut être une limitation intrinsèque de l'application
- Utiliser tous les cœurs peut être contreproductif
  - Augmentation de la contention
    - mémoire – bus
    - sections critiques (variables partagées)
- *L'objectif est d'utiliser de façon équilibrée l'utilisation de la machine (cœurs, mémoires, bus, cartes réseaux…)*

Point sur l'équilibrage de charge

- Approche statique très performante si l'on sait équilibrer la charge à l'avance
  - Approche la plus simple à comprendre et à optimiser
    - Plus satisfaisante intellectuellement
  - Non adaptée aux problèmes *irréguliers*
    - la complexité du traitement est plus liée à la valeur des données qu'à leur structuration
- Un recours : l'approche dynamique
  - Augmente la probabilité d'équilibrer la charge
    - N'est pas à l'abri d'un manque de chance
  - Augmente la synchronisation (contention sur un mutex)
  - Un compromis : jouer sur la granularité
    - Distribuer des tranches d'indices de tailles intermédiaires
- Alternatives :
  - Voler du travail
    - Un processeur inoccupé prend des indices à un autre processeur
  - Augmenter le nombre de threads
    - Et laisser faire le système d'exploitation…

Maîtriser les synchronisations

- Mise œuvre d'une barrière de synchronisation
  - Simple
  - Version deux temps
- Montrer que l'on peut remplacer des synchronisations par du calcul redondant

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

```
#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)

```
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);
  pthread_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);

 …
}
```

=> Surcoût moindre

Paradigme de programmation *Single Program Multiple Data*

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

```
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)

```
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);
  pthread_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);

 …
}
```

=> Surcoût moindre

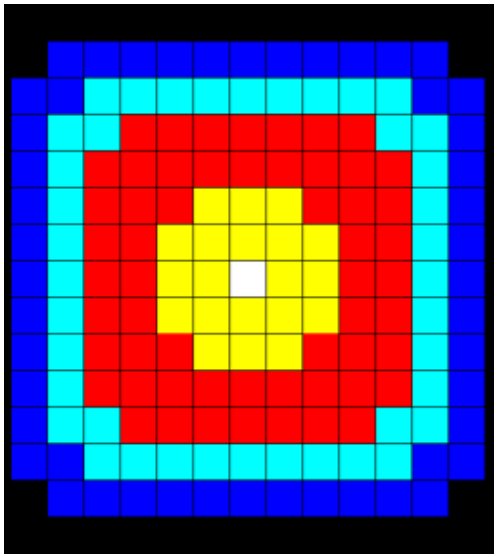
Implémentation d'une barrière

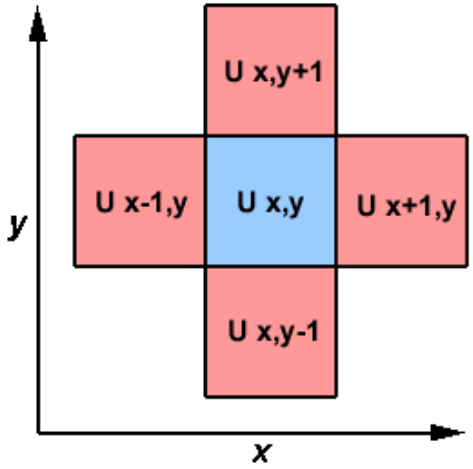
```
typedef struct {
  pthread_cond_t condition;
  pthread_mutex_t mutex;
  int attendus;
  int arrives;
} barrier_t ;

int
barrier_wait(barrier *b)
{
int val = 0;
pthread_mutex_lock(&b->mutex);
b->arrives++;
if ( b->arrives != b->attendus)
  pthread_cond_wait(&b->condition,
                                &b->mutex) ;
else {
   val=1;
   b->arrives = 0;
   pthread_cond_broadcast(b->condition);
 }
pthread_mutex_unlock(&b->mutex);
return val;
}
```

Application au *stencil* (pochoir)

- Méthode itérative de calcul de la valeur des éléments d'un tableau
- La valeur suivante est fonction des cellules voisines.
- Les cellules utiles au calcul d'une cellule forment un motif : le pochoir
- Applications :
  - Résolution EDP
  - Résolution système linéaire
  - simulation

$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$

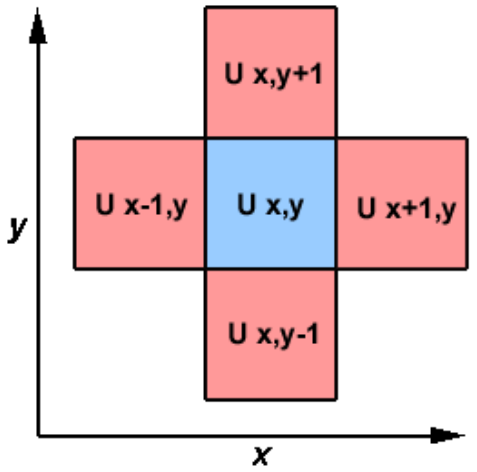




Stencil - séquentiel

On travaille sur deux tableaux

```
int T[2][DIM][DIM];
for(etape = 0; etape < ETAPE; etape++)
{
       in = 1-in;
       out = 1 - out;
        for(i=1; i < DIM-1; i++)
            for(j=1; j < DIM-1; i++)
               {
                 T[out][i][j] =f(Voisinage(T,i,j,in))
                }
}
```

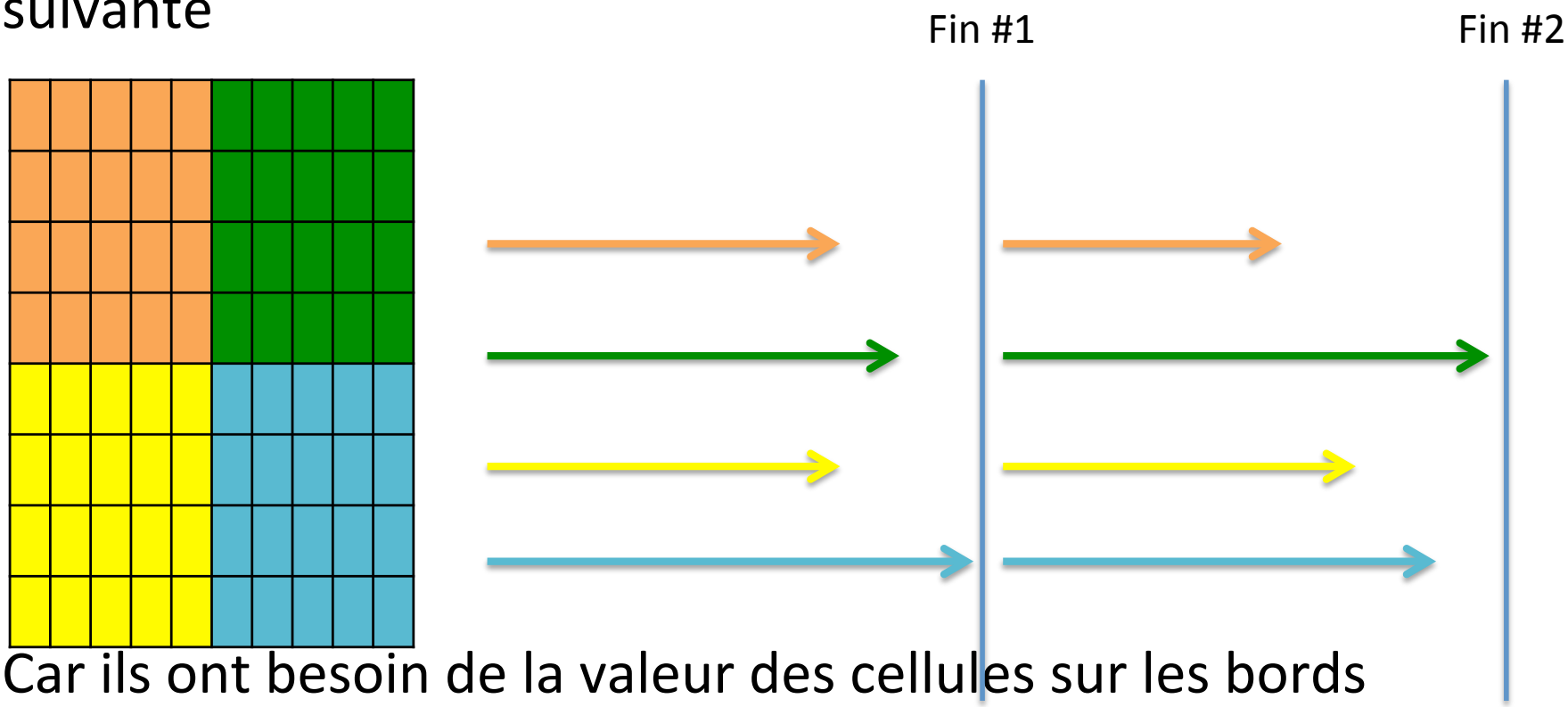


Stencil
Code d'un thread version SPMD

```
void calculer(void *id)
{
   int mon_ordre = (int) id;
   int etape, in = 0, out = 1 ;
   int debut = id * …
   int fin = (id +1) * …
  for (etape=0 ...)
  {
    for(i = debut ; i <  fin ; i++)
       …
    pthread_barrier_wait(&bar);
}
```

Stencil
Observation des synchronisations

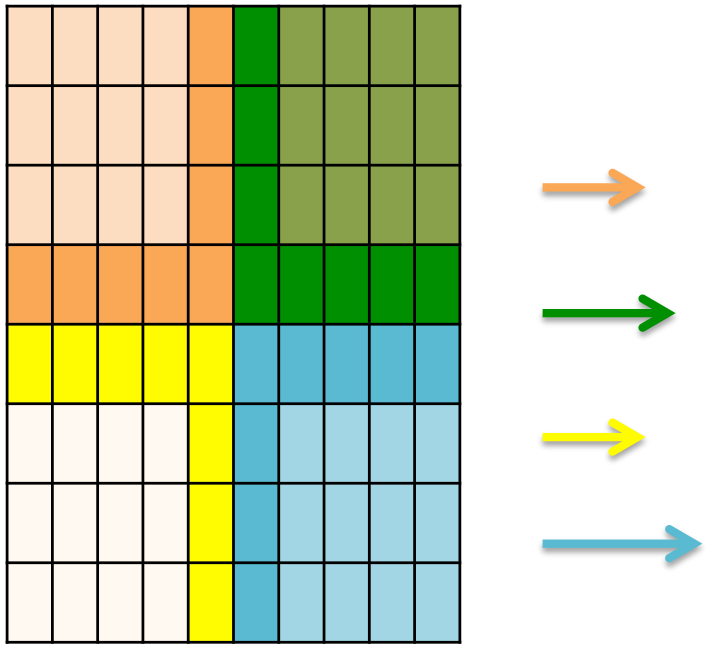
- Les threads s'attendent en fin d'étape pour passer à la suivante



- Car ils ont besoin de la valeur des cellules sur les bords calculés par les threads voisins

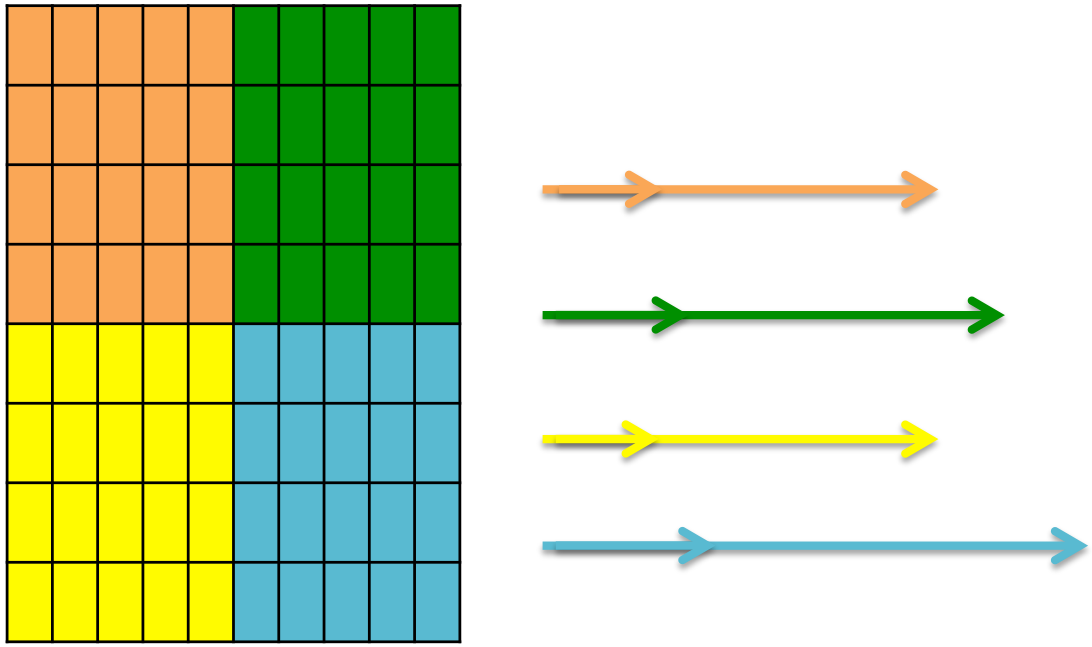
Stencil

- Calculons les bords en premier



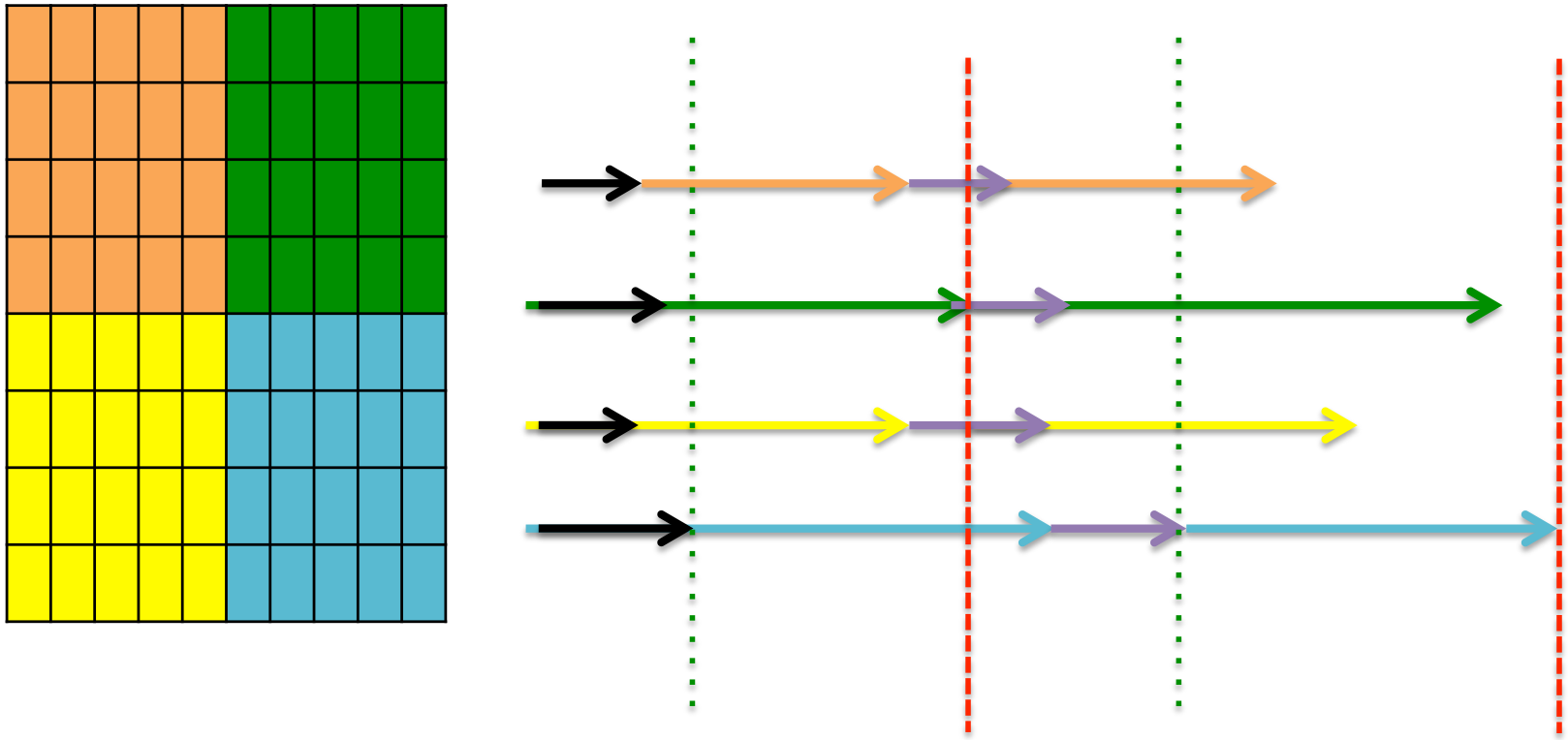
Stencil

- Puis l’intérieur des régions



Stencil

- Ici les threads peuvent enchainer deux étapes sans s'attendre

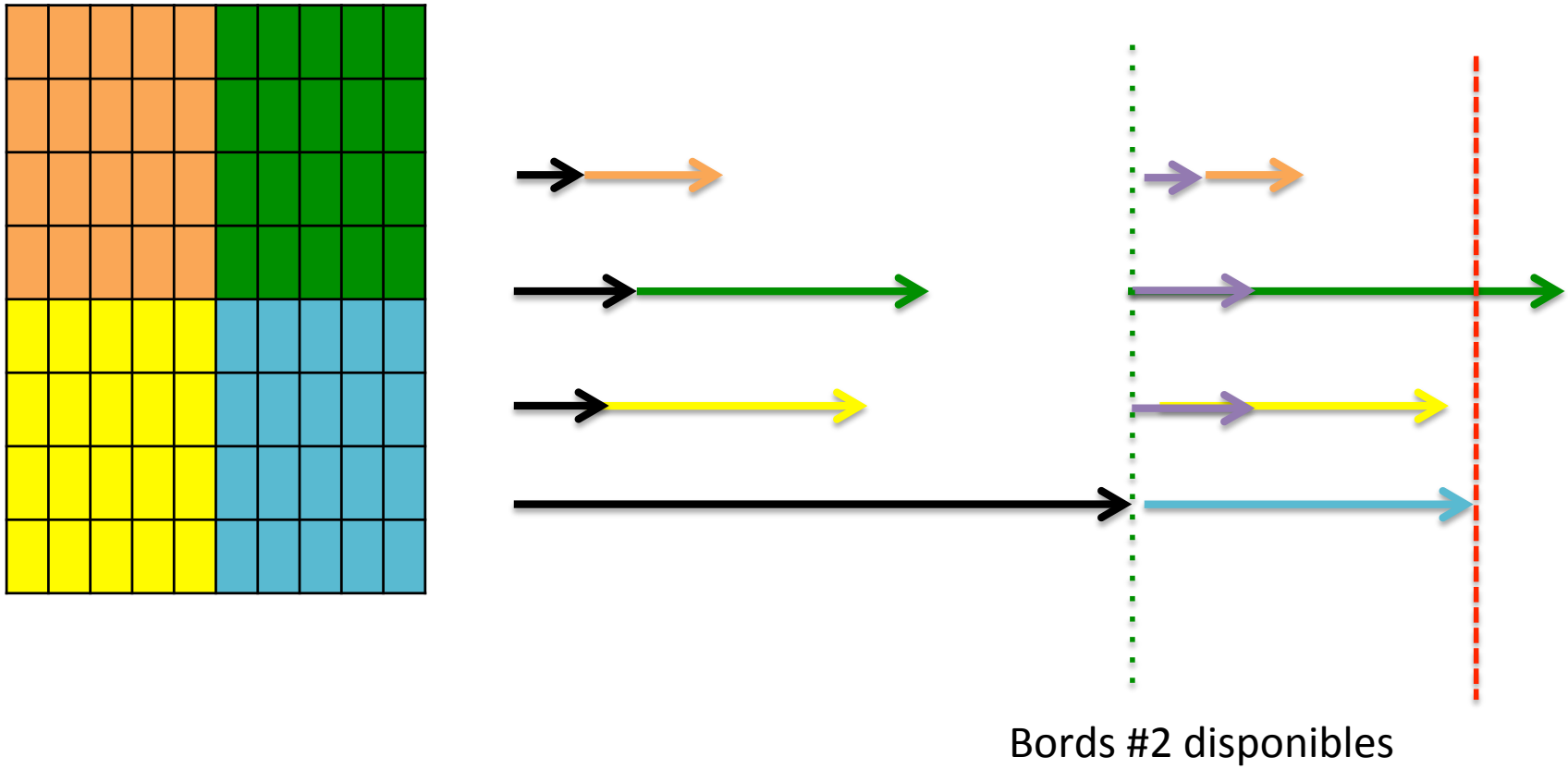


Bords pour l'étape #2 sont disponibles

Bords pour l'étape #3 sont disponibles

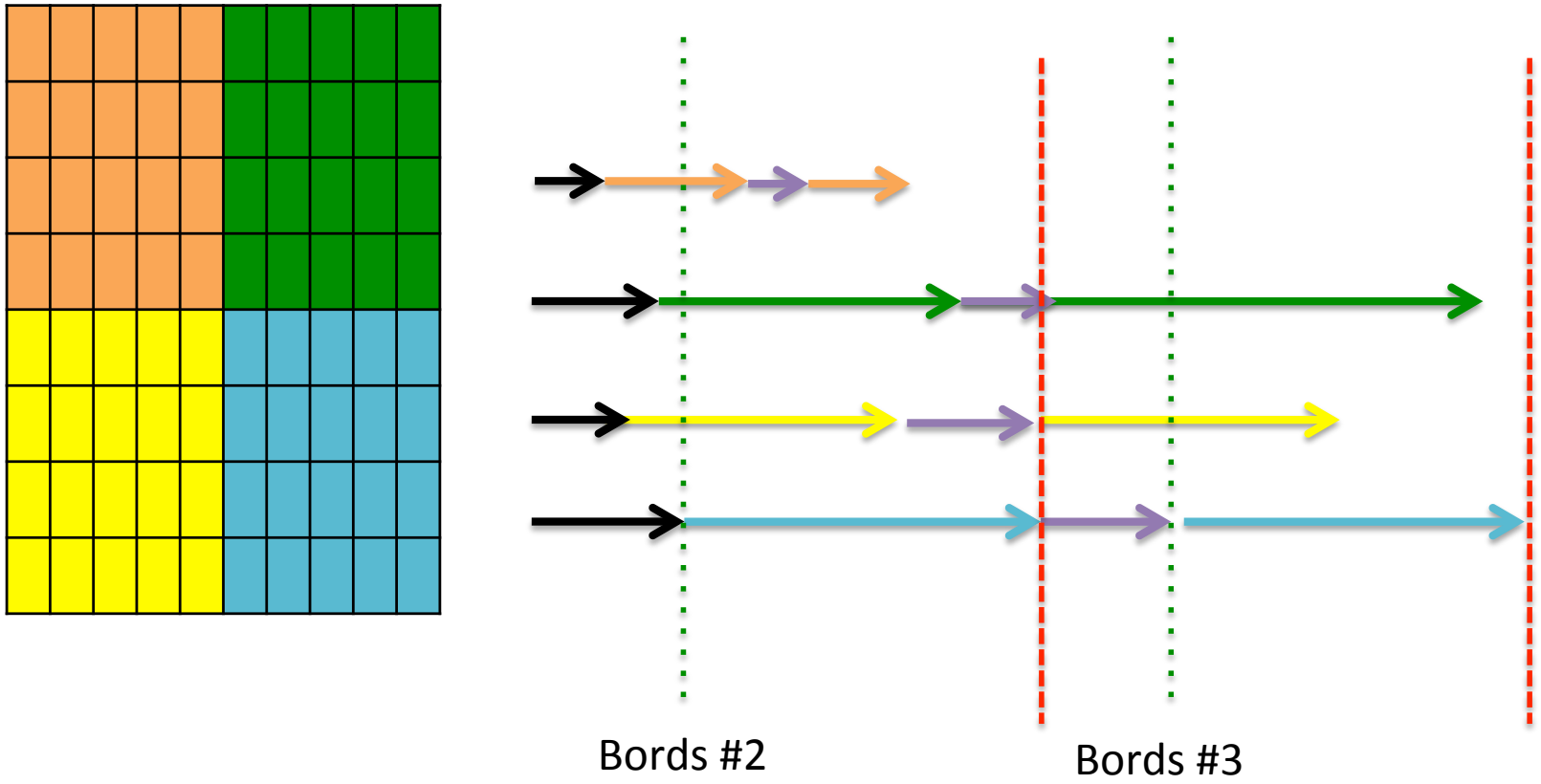
Stencil

- Mais on ne gagne pas toujours :
  - Cas d'un bord trop long à calculer



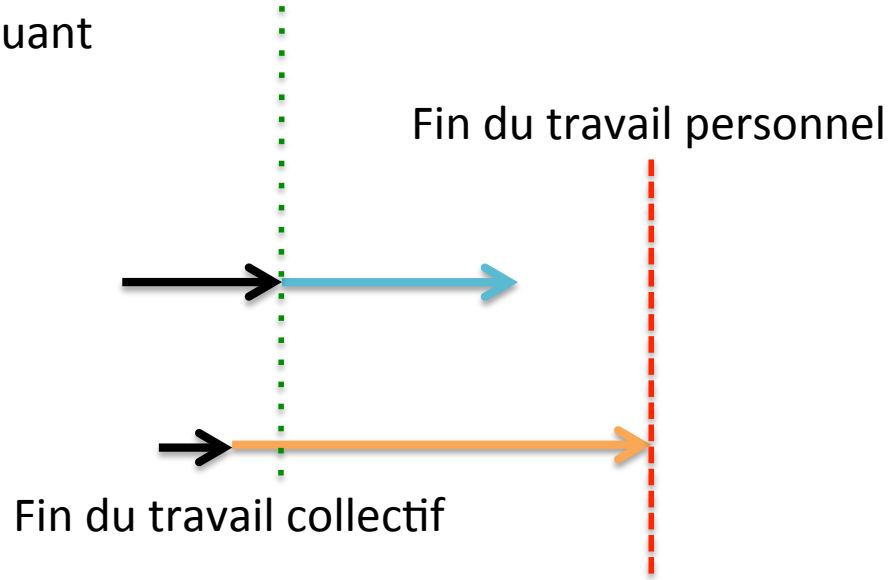
Stencil

- Pour gagner le calcul doit être bien équilibrer



Barrière en 2 temps

- int pthread_barrier_wait_begin(barrier_t *bar);
  - Non bloquant



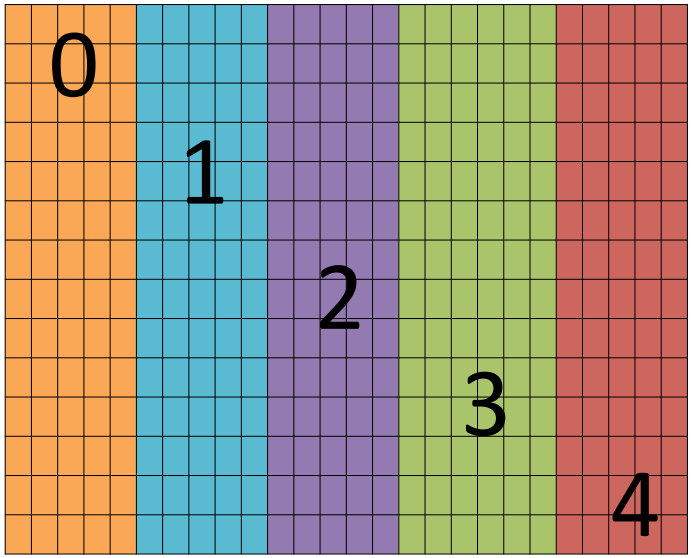
- int pthread_barrier_wait_end(barrier_t *bar);
  - Bloquant si tous les threads n'ont pas passé le wait_begin

Stencil
code barrière en 2 temps

```
calculer_bordure();
pthread_barrier_wait_begin(&bar);
calculer_centre();

if( pthread_barrier_wait_end(&bar) == 0) // dernier thread a avoir franchi la barrière
{
}
```

Stencil
Vers plus de désynchronisation

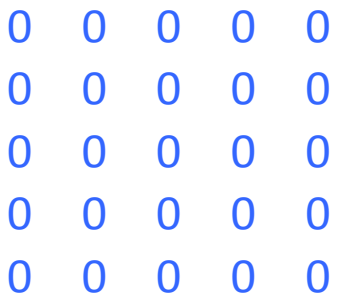
Une barrière par frontière



– Faire cohabiter plusieurs générations en même temps
  - Nécessite un compteur de cellules par génération

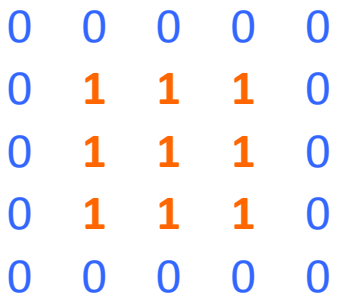
Stencil
Encore plus de désynchronisation

- Réduire le nombre de synchronisations
  - On peut calculer l'état d'une cellule sur k étapes si on connaît l'état des cellules à distance k.



Stencil
Encore plus de désynchronisation

- Réduire le nombre de synchronisations
  - On peut calculer l'état d'une cellule sur k étapes si on connaît l'état des cellules à distance k.



Stencil
Encore plus de désynchronisation

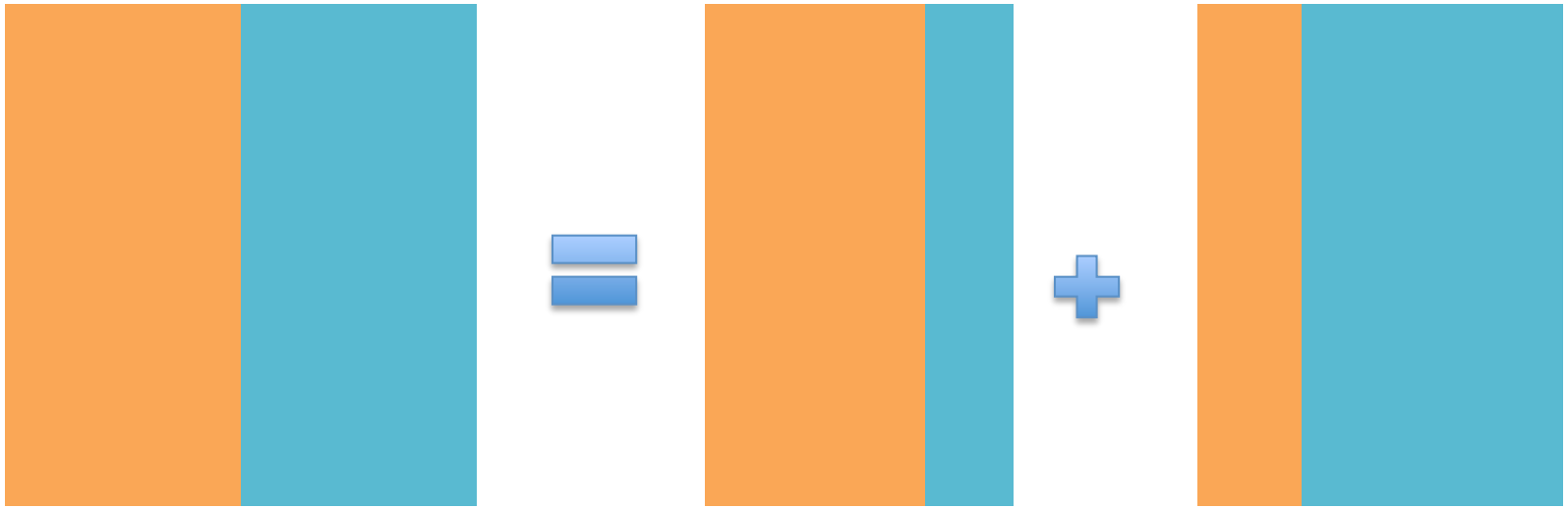
- Réduire le nombre de synchronisations
  - On peut calculer l'état d'une cellule sur k étapes si on connaît l'état des cellules à distance k.

0 0 0 0 0
0 1 1 1 0
0 1 **2** 1 0
0 1 1 1 0
0 0 0 0 0

  - Idée : remplacer des synchronisations par du calcul redondant

Introduction d'une zone de recouvrement (shadow-zone)

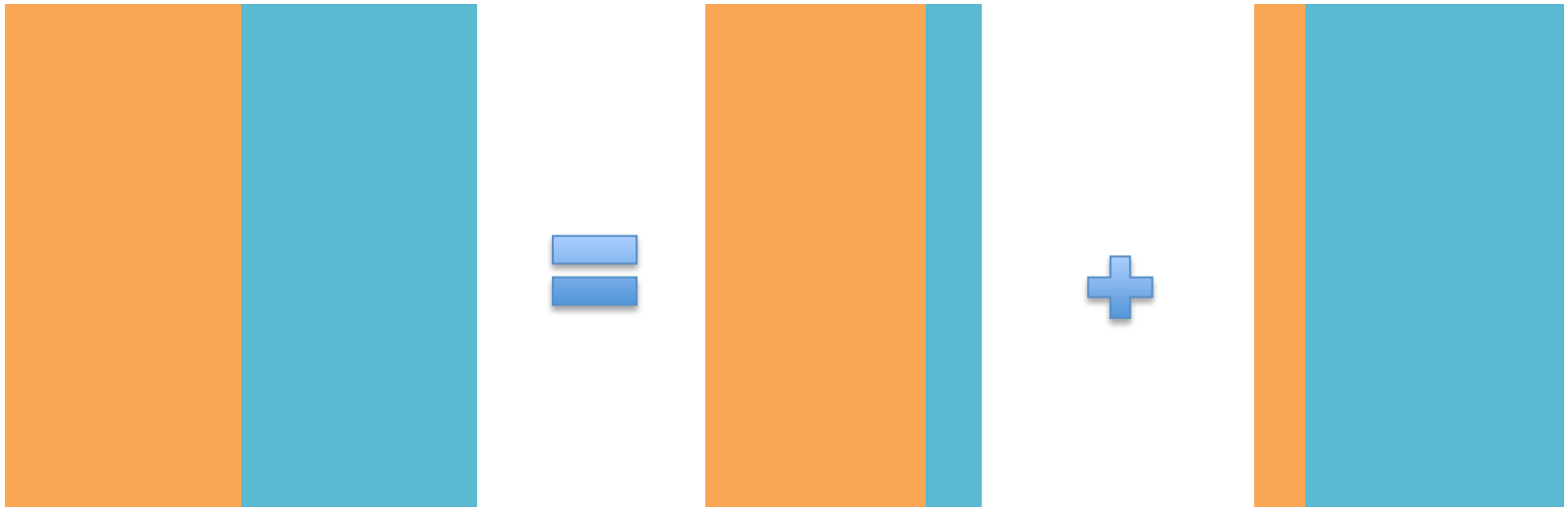
- Dupliquer la zone frontière voisine
  - épaisseur de k cellules permet de calculer k étapes sans synchronisation



- Étape 1

Introduction d'une zone de recouvrement (shadow-zone)

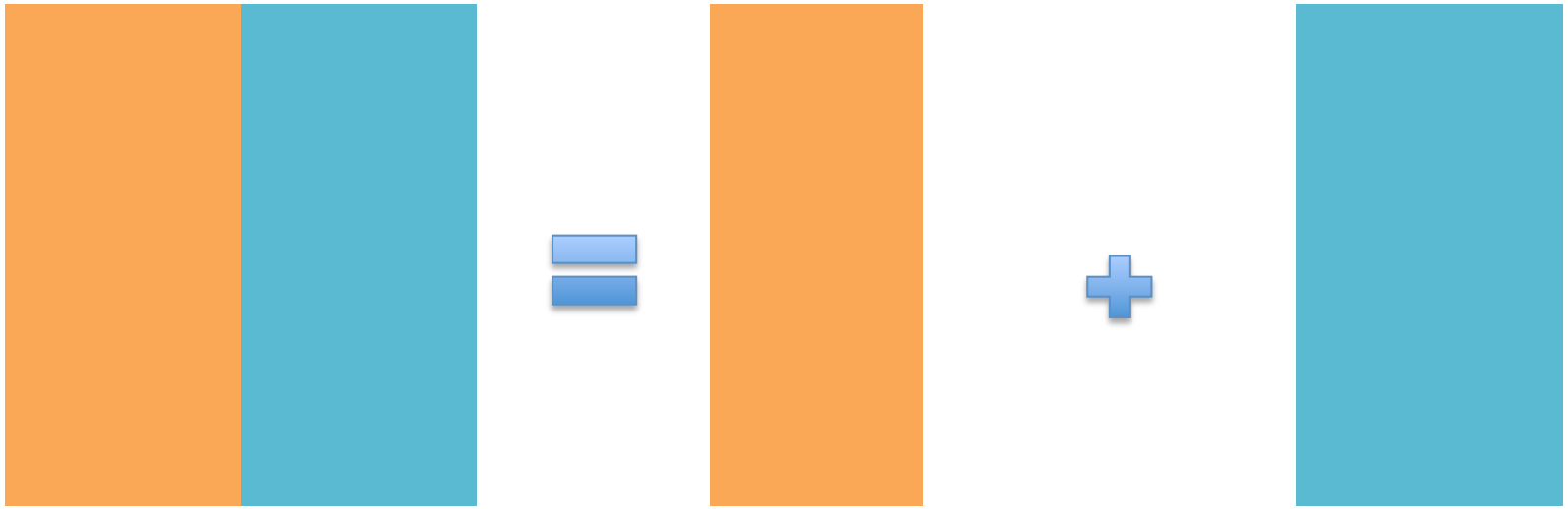
- Dupliquer la zone frontière voisine
  - épaisseur de k cellules permet de calculer k étapes sans synchronisation



- Étape 2

Introduction d'une zone de recouvrement (shadow-zone)

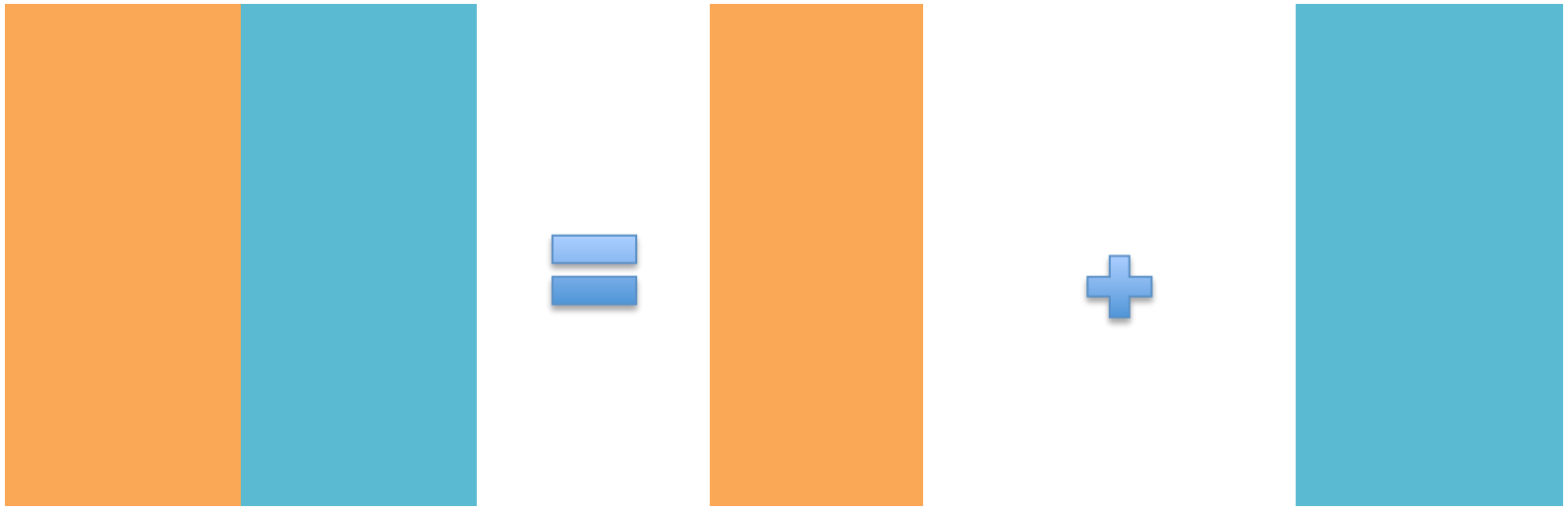
- Dupliquer la zone frontière voisine
  - épaisseur de k cellules permet de calculer k étapes sans synchronisation



- Étape 3

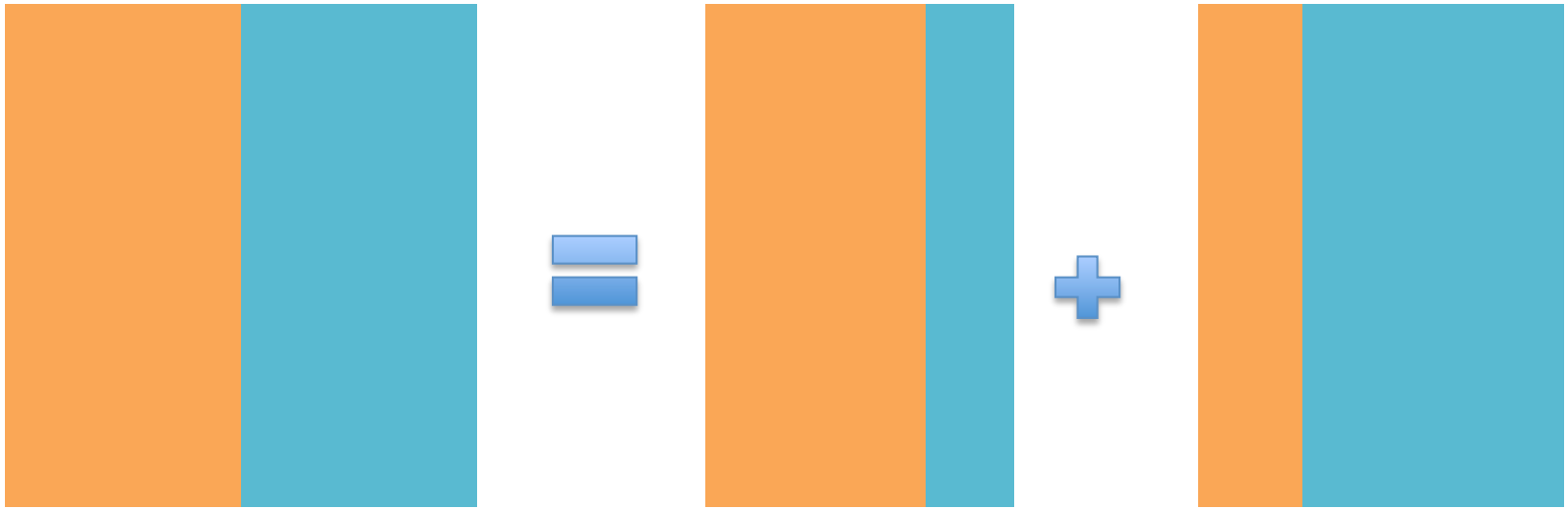
Introduction d'une zone de recouvrement (shadow-zone)

- Dupliquer la zone frontière voisine
  - épaisseur de k cellules permet de calculer k étapes sans synchronisation
  - Nécessite une barrière puis une recopie



Introduction d'une zone de recouvrement (shadow zone)

- Dupliquer la zone frontière
  - épaisseur k permet de calculer k étapes sans synchronisation
  - Nécessite une barrière puis une recopie



Technique utilisée lorsque les synchronisation sont coûteuses (réseaux – machines complexes)

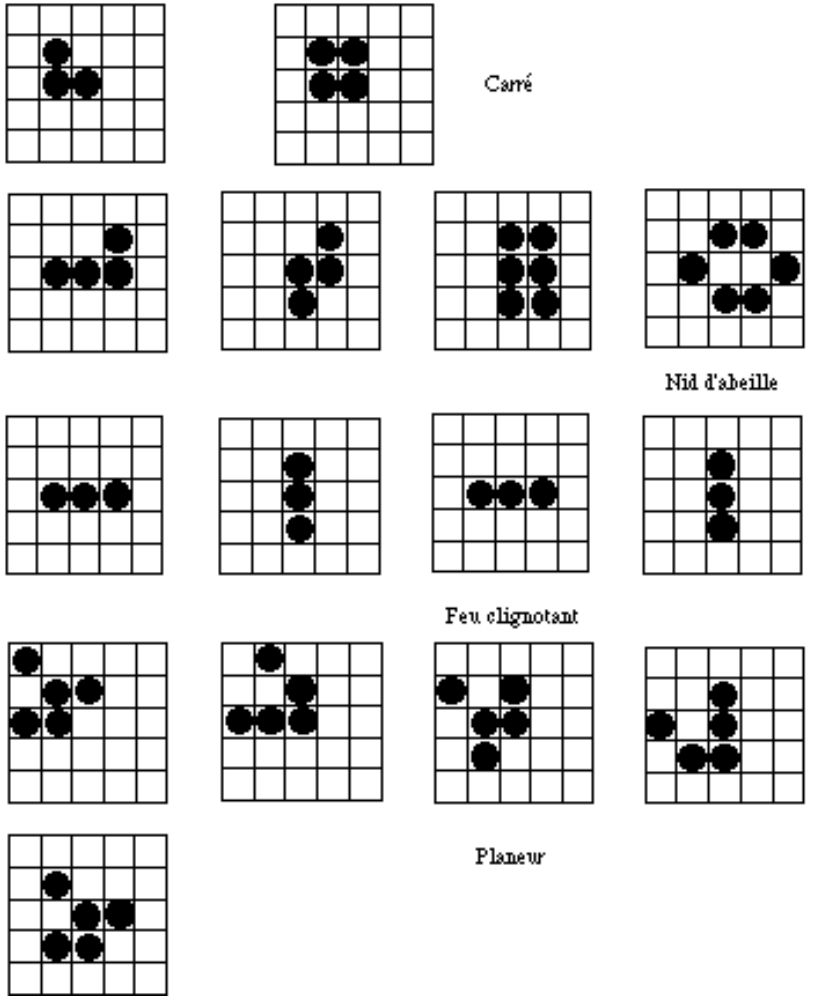
Conclusion sur le chapitre

- Tirer les meilleures performances d'une machine pour un problème donné demande
  - Une étude algorithmique
    - Faire apparaître des traitements indépendants
    - Synchroniser modérément
  - Une mise au point sur machine
    - Expérimenter pour observer
    - Optimiser pour comprendre
    - Utilisation d'outils spécialisés
- Programmer avec les threads
  - C'est bien car
    - On contrôle presque tout
    - On peut inventer ses propres mécanismes de synchronisation
  - Mais c'est un peu pénible…
    - Surtout pour un non informaticien
    - Souvent les mêmes schémas
    - Modification lourde du code

Stencil par Conway

- À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :
  - Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
  - Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

- Intérêts pour le calcul parallèle :
  - *Stencil* : classe importante d'application
  - Turing puissant
    - Problème Régulier ou Irrégulier



Résultats expérimentaux nehalem salle 008

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution statique des lignes par blocs

Nb threads	base	speedup
1	5533,15	
2	2819,41	1,92
4	1450,32	3,73
8	709,62	7,62
16	494,69	10,93
32	453,92	11,91
64	457,33	11,82

Performances non optimales pour une machine de 8 cœurs « hyperthreadés »: on devrait atteindre les meilleures performances avec 8 ou 16 cœurs

Résultats expérimentaux nehalem salle 008

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution statique des lignes par blocs

Nb threads	base	speedup
1	5533,15	
2	2819,41	1,92
4	1450,32	3,73
8	709,62	7,62
16	494,69	10,93
32	453,92	11,91
64	457,33	11,82

Performances non optimales pour une machine de 8 cœurs « hyperthreadés »: on devrait atteindre les meilleures performances avec 8 ou 16 cœurs

Pistes : loi d'amdahl,
mauvais équilibrage de charge,
contention (mutex, mémoire),
placement / ordonnancement des threads

Résultats expérimentaux nehalem salle 008

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution statique des lignes par blocs

Nb threads	base	speedup
1	5533,15	
2	2819,41	1,92
4	1450,32	3,73
8	709,62	7,62
16	494,69	10,93
32	453,92	11,91
64	457,33	11,82

Mesure du nombre de cycles entre juste avant la création et la terminaison de chaque thread

11 364.549
4 364.963
5 365.235
7 456.930
10 457.681
6 457.883
12 457.920
3 458.381
8 458.477
1 459.642
9 586.832
2 589.490
0 590.815
13 598.460
15 600.970
14 603.831
604.094

Pistes : loi d'amdahl,
mauvais équilibrage de charge,
contention (mutex, mémoire),
placement / ordonnancement des threads

Résultats expérimentaux nehalem salle 008

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution statique des lignes par blocs

Nb threads	base		Optim. placement		Optim. mémoire		Placement + mémoire	
1	5533,15	0,98	5534,75	0,98	5428,23	1,00	5404,59	1,00
2	2819,41	1,92	2811,09	1,92	2766,00	1,95	2747,78	1,97
4	1450,32	3,73	1419,52	3,81	1400,10	3,86	1373,94	3,93
8	709,62	7,62	724,34	7,46	697,68	7,75	685,75	7,88
16	494,69	10,93	460,49	11,74	**433,69**	**12,48**	**435,69**	**12,40**
32	453,92	11,91	454,95	11,88	**435,89**	**12,40**	444,82	12,15
64	457,33	11,82	466,93	11,57	438,73	12,32	443,83	12,18

Optimisation placement : on fixe les threads sur les cœurs
Optimisation mémoire : on réparti le tableau sur les 2 *sockets*

Résultats expérimentaux nehalem salle 008

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution statique des lignes par blocs

Nb threads	base		Optim. placement		Optim. mémoire		Placement + mémoire	
1	5533,15	0,98	5534,75	0,98	5428,23	1,00	5404,59	1,00
2	2819,41	1,92	2811,09	1,92	2766,00	1,95	2747,78	1,97
4	1450,32	3,73	1419,52	3,81	1400,10	3,86	1373,94	3,93
8	709,62	7,62	724,34	7,46	697,68	7,75	685,75	7,88
16	494,69	10,93	460,49	11,74	**433,69**	**12,48**	**435,69**	**12,40**
32	453,92	11,91	454,95	11,88	**435,89**	**12,40**	444,82	12,15
64	457,33	11,82	466,93	11,57	438,73	12,32	443,83	12,18

Delta pour 16 threads ~200 ~20 ~2 ~10

Optimisation placement : on fixe les threads sur les cœurs

Optimisation mémoire : on réparti le tableau sur les 2 *sockets*

Optimisation mémoire

Initialisation de la matrice par les threads sur le cœur où il seront exécutés

```
void *init_par(void *p){
  int numero =*(int*)p;
  int i,j;

  int debut = ((N/P)*numero);
  int fin= ((N/P)*(numero+1));

for (i=debut;i<fin;i++)
     for (j=0;j<N;j++)
      C[i][j]=B[i][j]=A[i][j] = random();
}
```

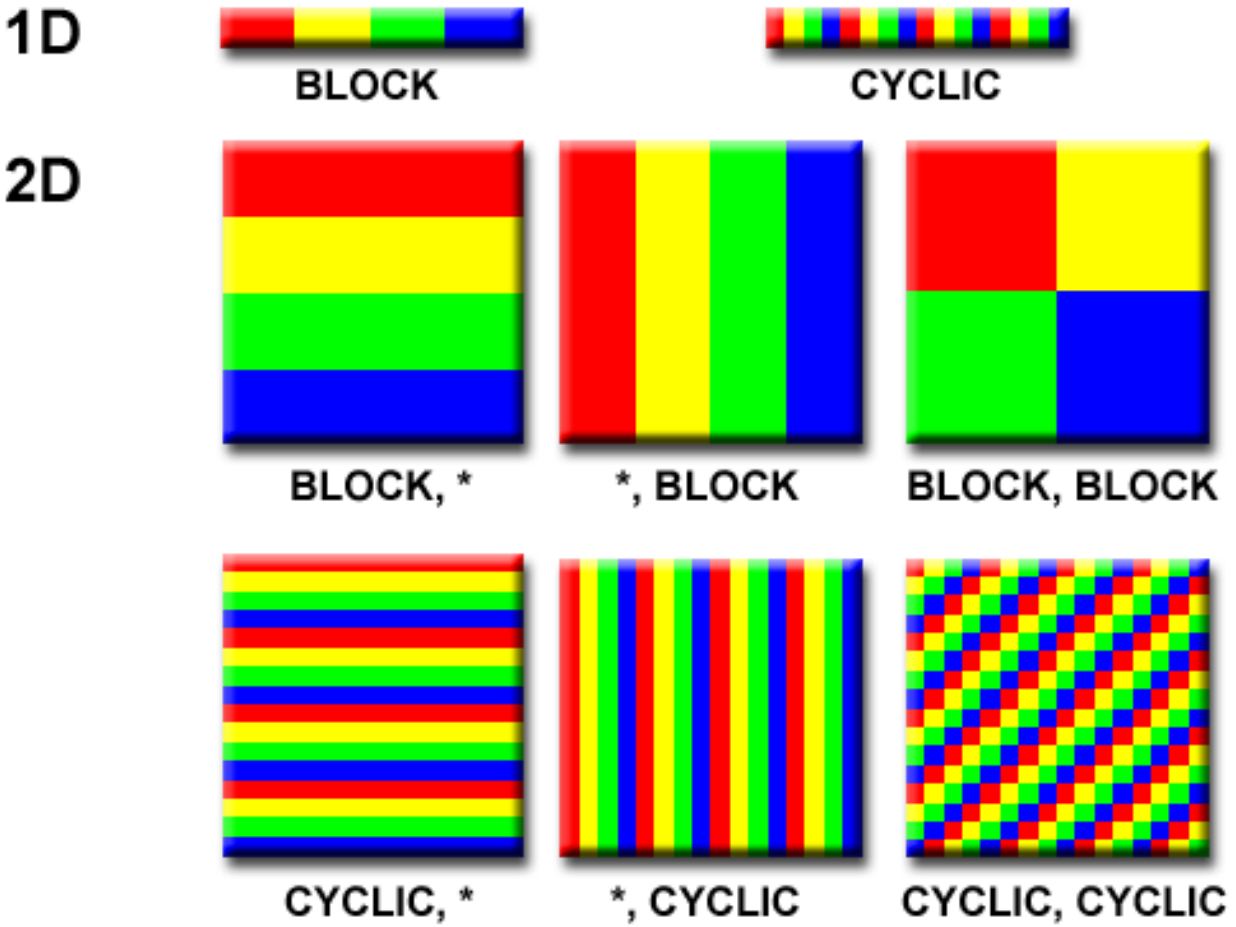
Résultats expérimentaux

C[i][j] = cos(A[i][j]) + sin(B[i][j]);

Distribution dynamique des lignes par blocs

	base	placement	memoire	les 2	dynamique	Dyn + mem	Dyn 1
1	5533,15	5534,75	5428,23	5404,59	5507.039	5390.514	5989.049
2	2819,41	2811,09	2766,00	2747,78	2802.935	2719.030	5851.930
4	1450,32	1419,52	1400,10	1373,94	1407.822	1372.205	11507.095
8	709,62	724,34	697,68	685,75	722.501	694.075	17271.893
16	494,69	460,49	433,69	435,69	452.924	441.039	17255.047
32	453,92	454,95	435,89	444,82	508.485	444.968	17364.186
64	457,33	466,93	438,73	443,83	458.390	442.785	17498.092

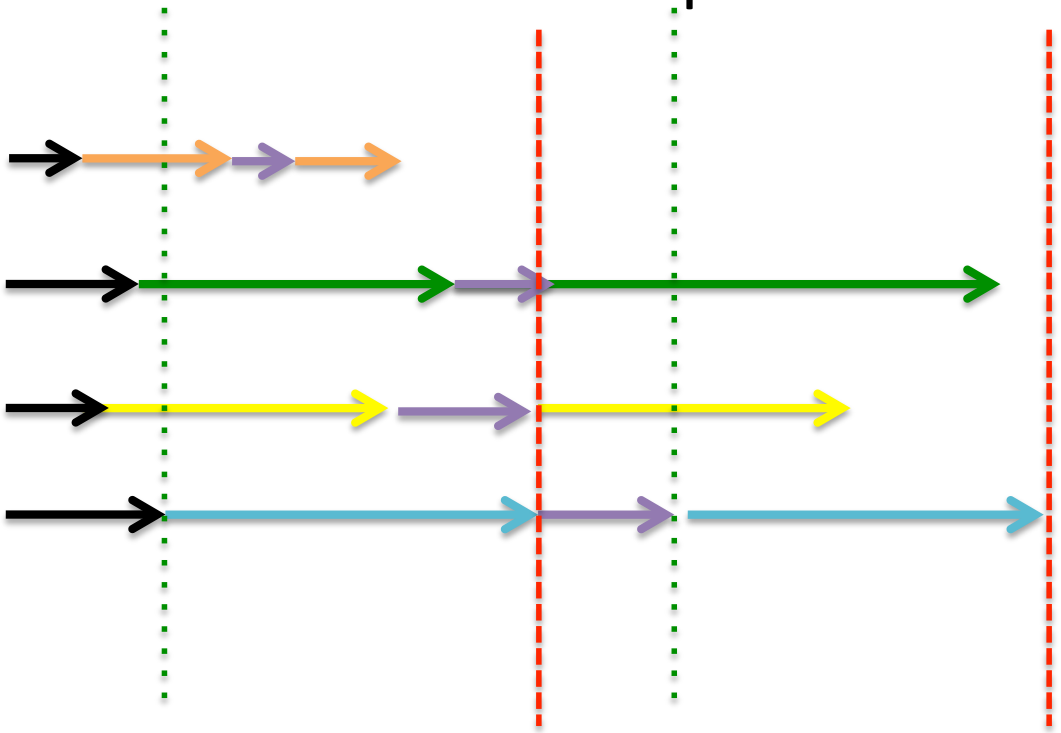
Répartition des données



Barrière en 2 temps

Difficulté de réalisation :

Deux générations de barrières doivent pouvoir coexister.



étape + 2 est disponible