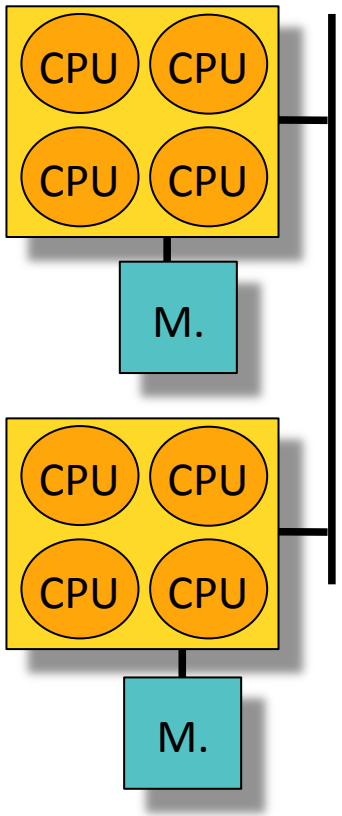
Approche mémoire partagée

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



Calcul en parallèle de Y[i] = f(T,i)

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

- approche statique : on distribue les indices au moment de la création des threads
- approche dynamique : on distribue les indices au fur et à mesure

Calcul en parallèle de Y[i] = f(T,i)

```
#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);

 ...
}
```

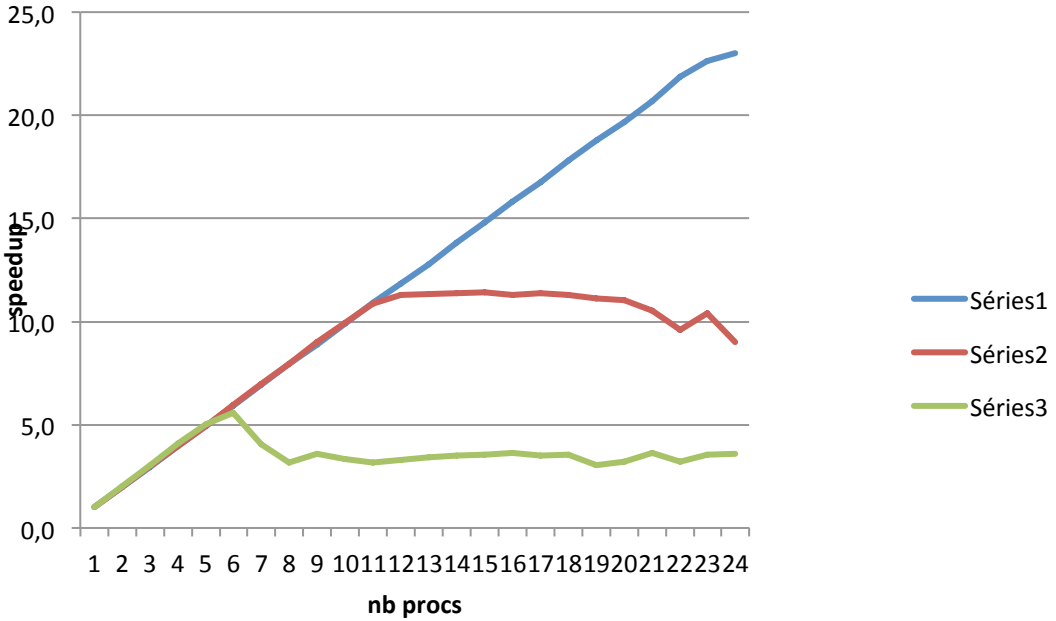
Parallélisation efficace si équilibrée

Calcul en parallèle de Y[i] = f(T,i)

- Solution statique Speedup limité à 2 si un thread a la moitié du travail
- 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);
}
```

Coût de la synchronisation



- Speed-up obtenus avec un programme trivial comprenant une section critique représentant un peu moins de 5%, 10% et 20% du temps de calcul sur une machine à 24 processeurs.

Application (Examen 2008)

- *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).*

Equilibrage de charge : un problème difficile

- Approche statique très performante si l'on sait équilibrer la charge à l'avance
  - Impossible pour les problèmes *irréguliers*
    - la complexité du traitement est plus liée à la valeur des données qu'à leur structuration
- Un recourt : approche dynamique
  - Augmente la probabilité d'équilibrer la charge
  - N'est pas à l'abri d'un manque de chance
  - Augmente la synchronisation
- Un compromis : jouer sur la granularité
  - Distribuer des tranches d'indices de tailles intermédiaires
- 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…

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

```
#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)

```
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);

…
}
```

- Surcoût moindre

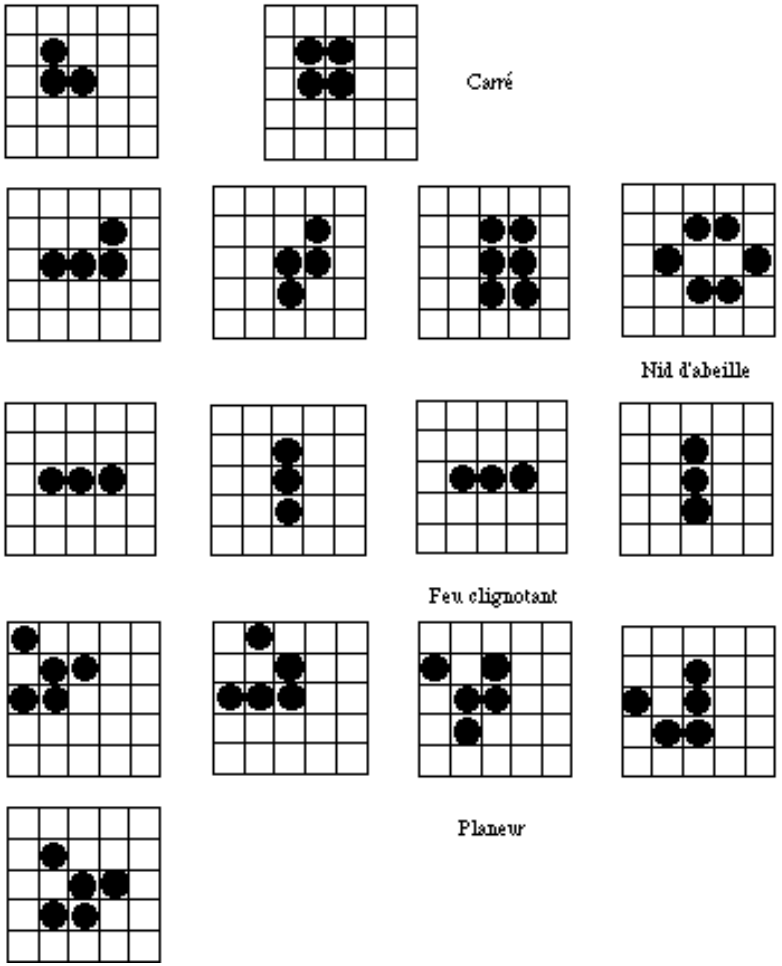
Implémentation d'une barrière

```
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;
}
```

Jeu de la vie 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



Jeu de la vie - séquentiel

```
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);
}
```

Le programme affiche le nombre de cellules vivantes à chaque étapes.

Jeu de la vie - parallèle

```
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++)
     …
  if (T[out][i][j]) // cellule vivante
 {
  pthread_mutex_lock(&mutex_cell);
  nb_cellules ++;
   pthread_mutex_unlock(&mutex_cell)
 }
}/* for i */
pthread_barrier_wait(&bar);

 if (mon_ordre == 0)
  {
    printf(...);
    nb_cellules = 0;
  }
 pthread_barrier_wait(&bar);
}
```

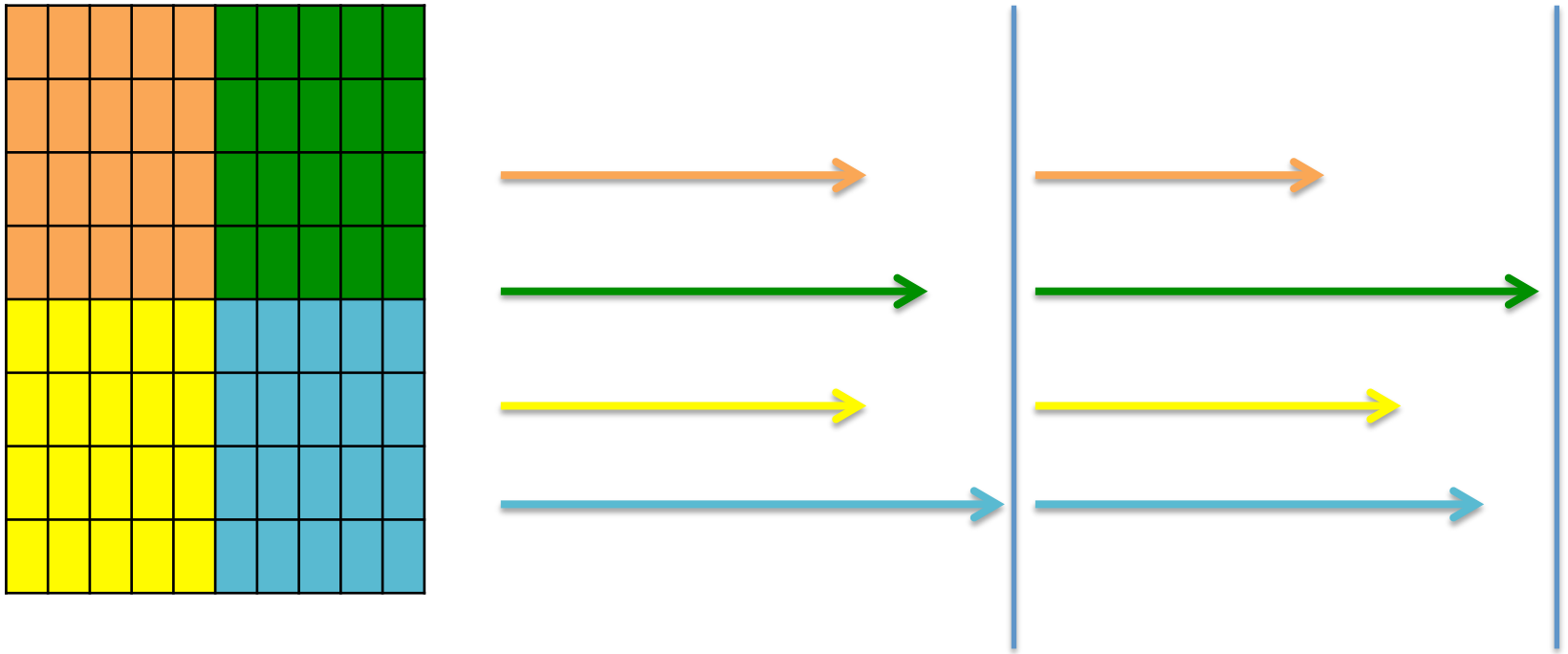
Jeu de la vie - parallèle

```
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 ...)
 {
 int mes_cellules = 0;
  for(i = debut ; i <  fin ; i++)
    { …
     mes_cellules++ ;
     …
     }
```

```
pthread_mutex_lock(&mutex_cell);
nb_cellules += mes_cellules;
pthread_mutex_unlock(&mutex_cell)
pthread_barrier_wait(&bar);

if (mon_ordre == 0)
 {
   printf(...);
   nb_cellules = 0;
 }
pthread_barrier_wait(&bar);
}
```

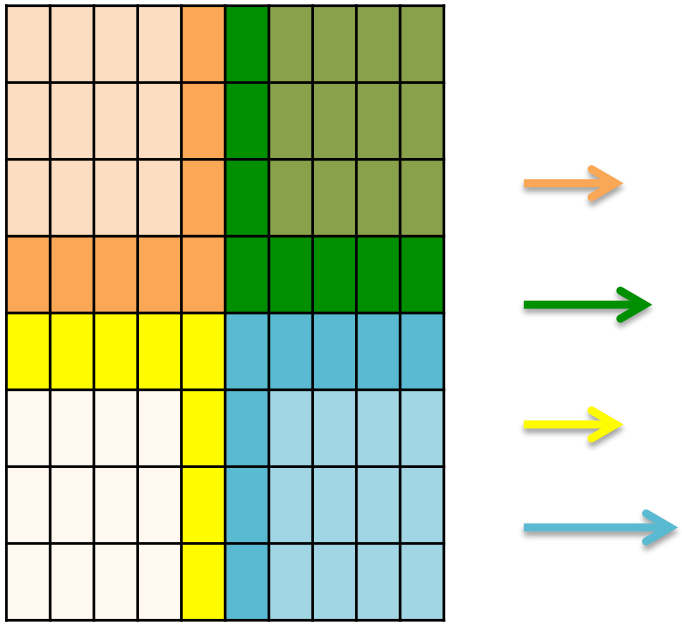
Jeu de la vie

- Les threads sont très synchronisés



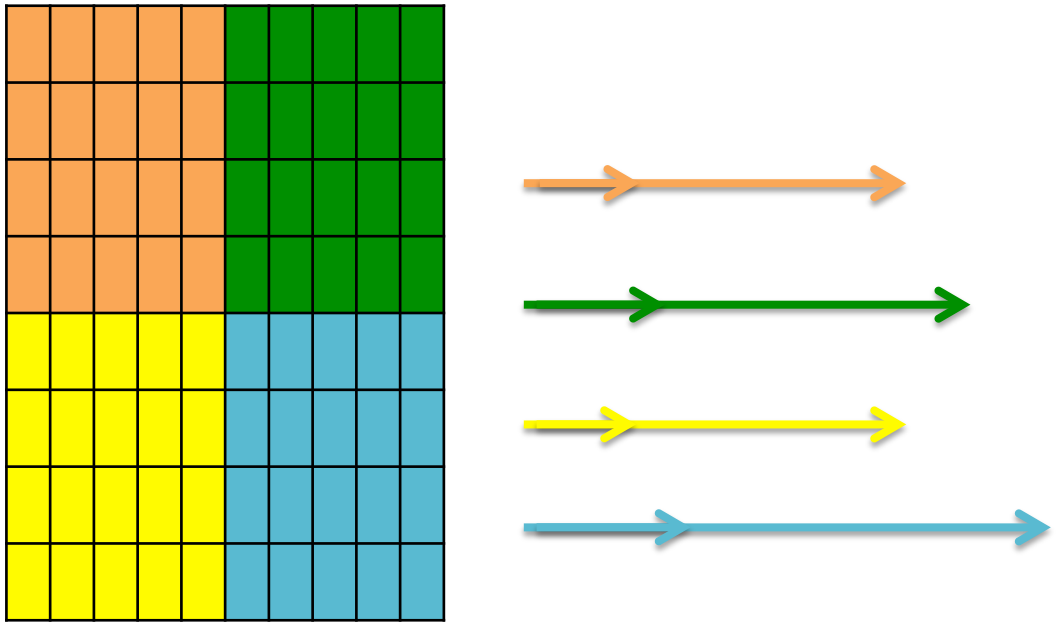
Jeu de la vie

- Désynchroniser en calculant d'abord les bords



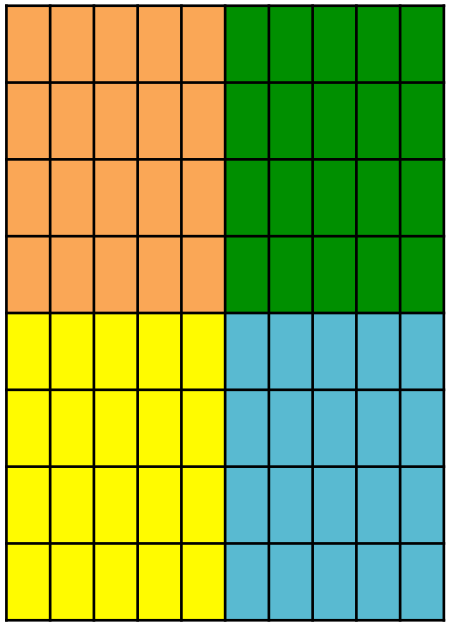
Jeu de la vie

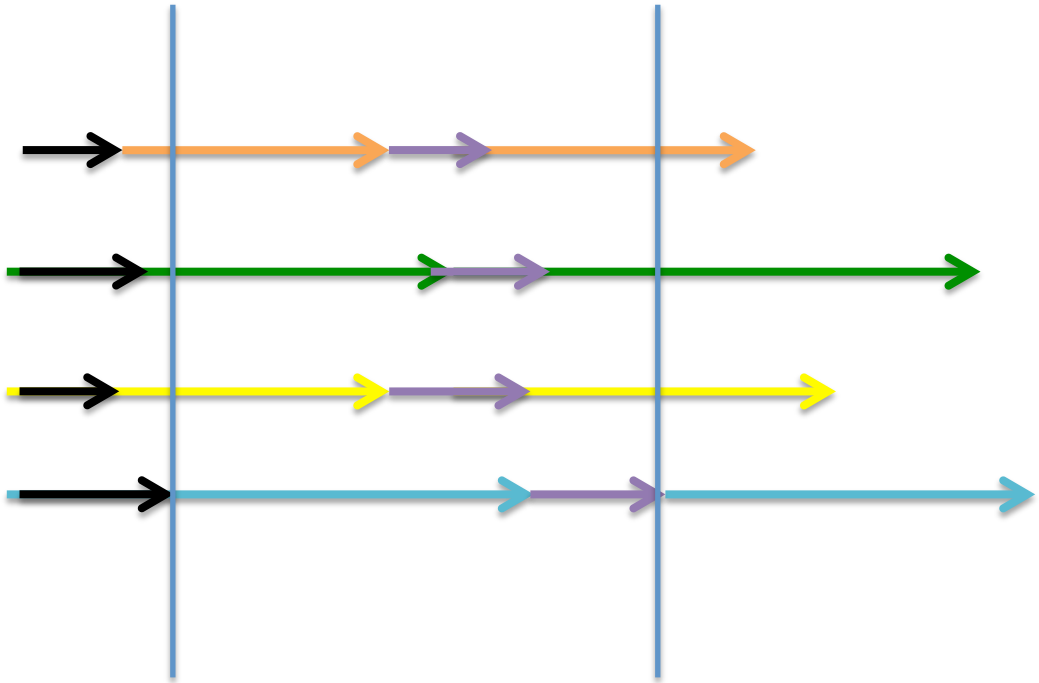
- Puis l'intérieur des régions



Jeu de la vie

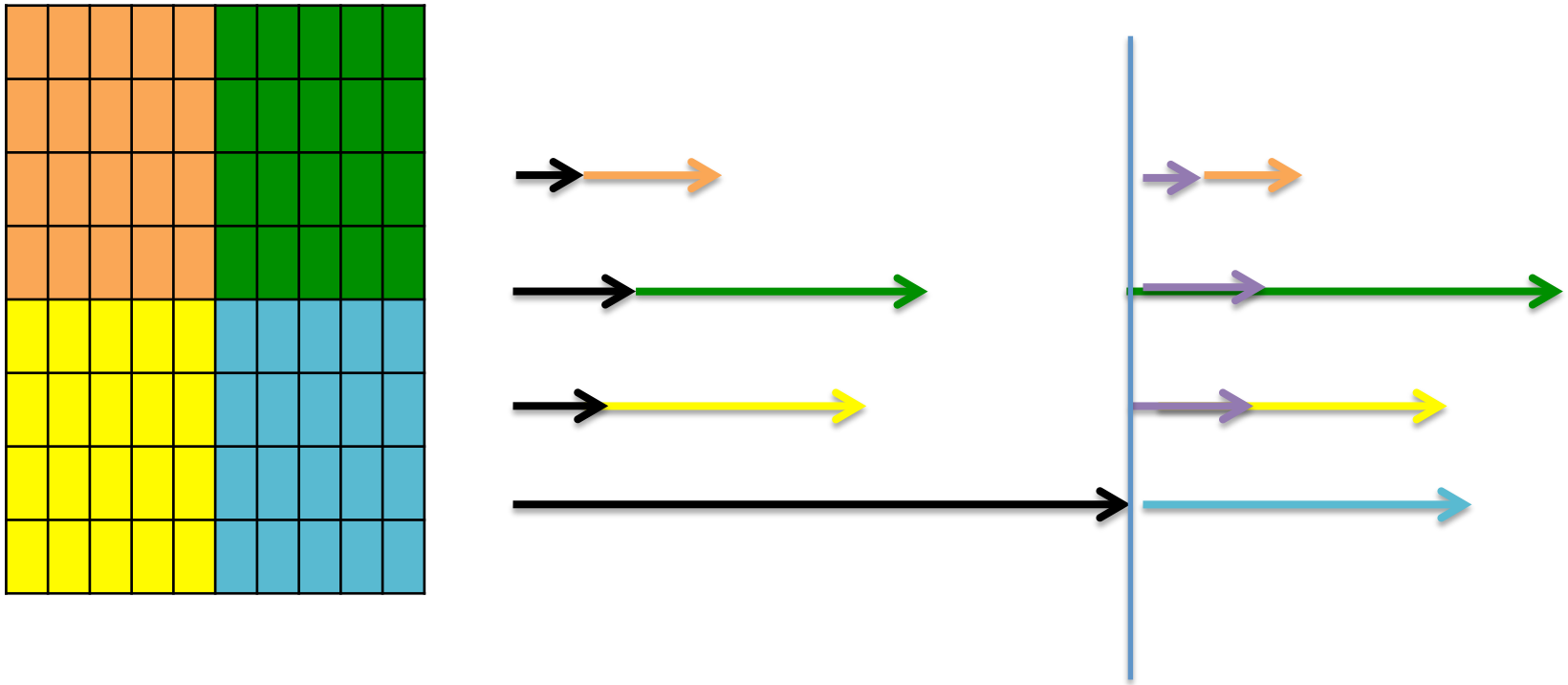
- Les threads s'attendent moins





Jeu de la vie

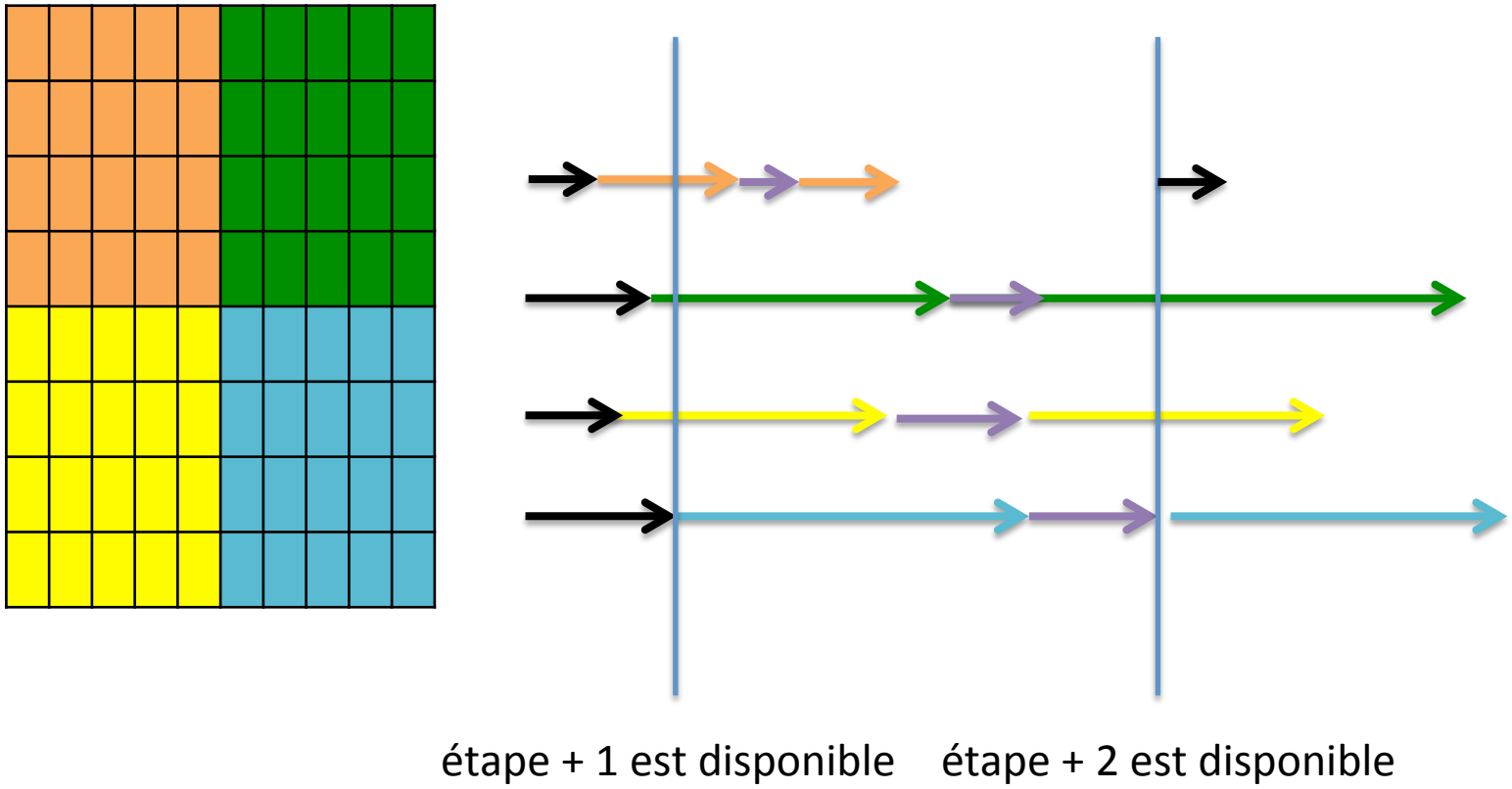
- Les threads s'attendent un peu moins…



étape + 1 est disponible

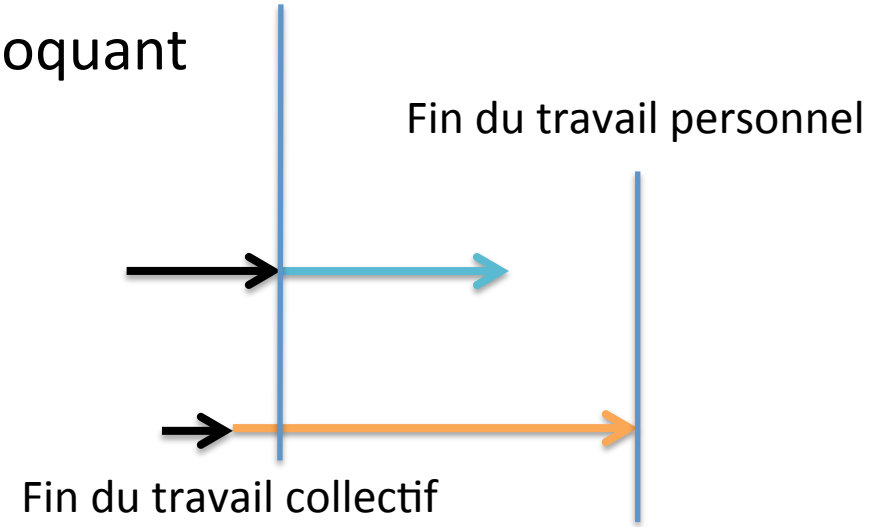
Jeu de la vie

- Les threads s'attendent un peu moins…



Barrière en 2 temps

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



– int pthread_barrier_wait_end(barrier_t *bar);
  - Bloquant

Jeu de la vie (barrière en 2 temps)

```
calculer_bordure(mon_ordre, in, out);
pthread_barrier_wait_begin(&bar);
calculer_centre(mon_ordre, in, out);
pthread_mutex_lock(&mutex_cell);
nb_cellules += mes_cellules;
pthread_mutex_unlock(&mutex_cell)

if( pthread_barrier_wait_end(&bar) == 0) // dernier thread a avoir franchi la
barrière
{
        pthread_mutex_lock(&mutex_cell);
        printf(nb_cellules);
        pthread_mutex_unlock(&mutex_cell) ;
}
```

Jeu de la vie (barrière en 2 temps)

```
calculer_bordure(mon_ordre, in, out);
pthread_barrier_wait_begin(&bar);
calculer_centre(mon_ordre, in, out);
pthread_mutex_lock(&mutex_cell);
nb_cellules[etape%2] += mes_cellules;
pthread_mutex_unlock(&mutex_cell)

if( pthread_barrier_wait_end(&bar) == 0) // dernier thread a avoir franchi la
barrière
{
        printf(nb_cellules[etape%2]);
        nb_cellules[etape%2] = 0;
 }
```

Jeu de la vie (barrière en 2 temps)

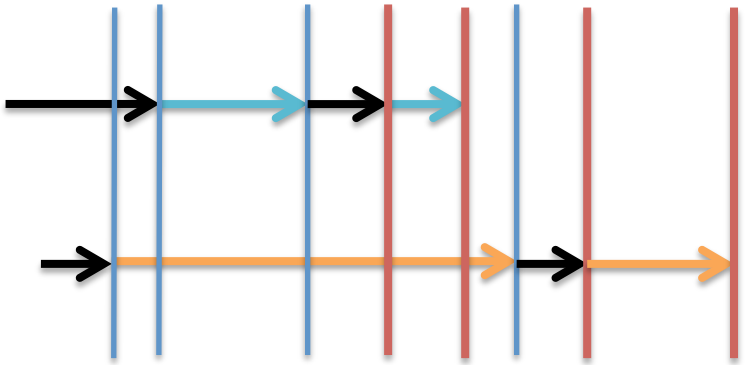
```
calculer_bordure(mon_ordre, in, out);
pthread_barrier_wait_begin(&bar);
calculer_centre(mon_ordre, in, out);
pthread_mutex_lock(&mutex_cell);
nb_cellules[etape%2] += mes_cellules;
pthread_mutex_unlock(&mutex_cell)

if( pthread_barrier_wait_end(&bar) == 0) // dernier thread a avoir franchi la
barrière
{
        printf(nb_cellules[etape%2]);
        nb_cellules[etape%2] = 0;
}
```

Barrière en 2 temps

Difficulté

Deux générations de barrières doivent coexister.

– Utiliser deux barrières

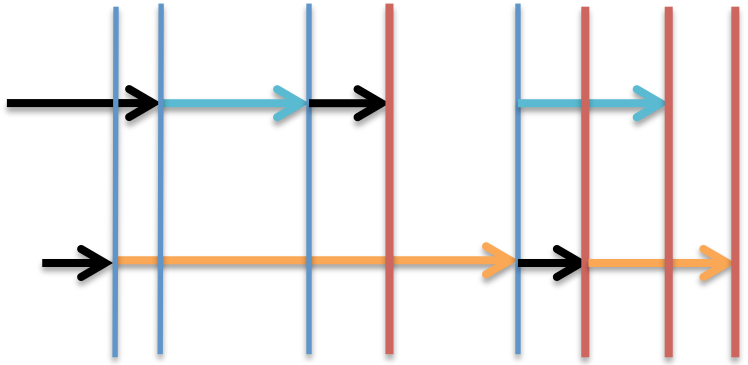


Barrière en 2 temps

Difficulté

Deux générations de barrières doivent coexister.

- Utiliser deux barrières



- Limiter la concurrence
  - Les threads entre begin et end doivent être de la même génération

Barrière en 2 temps

```
typedef struct
{
  pthread_cond_t conditionB;
  pthread_cond_t conditionE;
  pthread_mutex_t mutex;
  int attendus;
  int leftB;
  int leftE;
} barrier ;
```

```
void pthread_barrier_init(barrier_t *b,
                          int attendus)
{
...
 b->leftB = attendus;
 b->leftE = 0;
}
```

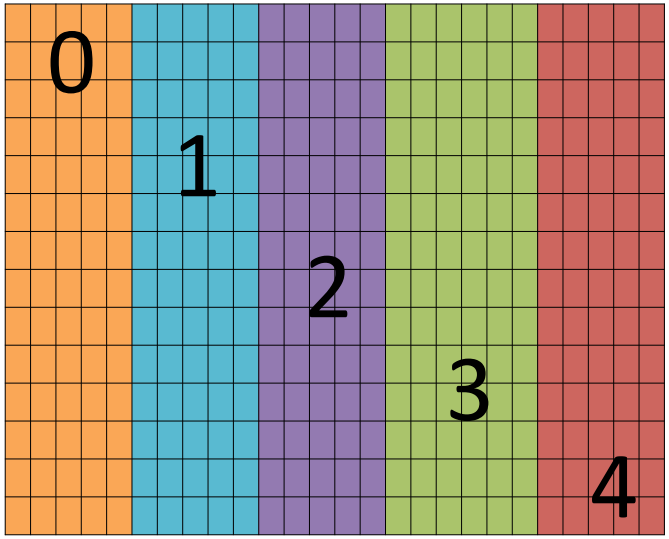
Barrière en 2 temps

```
int pthread_barrier_wait_begin(barrier_t *b)
{
 int ret = 0;
 pthread_mutex_lock(&b->mutex);
if (b->leftE)
  pthread_cond_wait(&b->conditionB, &b->mutex) ;
 ret = --b->leftB;
 if (ret == 0)
 {
   b->leftE = b->attendu;
  pthread_cond_broadcast(b->conditionE);
 }
pthread_mutex_unlock(&b->mutex);
return ret;
}
```

```
int pthread_barrier_wait_end(barrier_t *b)
{
int ret;
pthread_mutex_lock(&b->mutex);
if (b->leftB)
  pthread_cond_wait(&b->conditionE, &b->mutex) ;
ret = --b->leftE;
if(b->letfE == 0)
 {
  b->leftB = = b->attendu;
  pthread_cond_broadcast(b->conditionB);
 }
pthread_mutex_unlock(&b->mutex);
return ret;
}
```

Jeu de la vie
Vers plus de désynchronisation

Une barrière par frontière



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

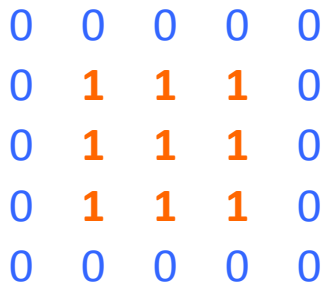
Jeu de la vie
Encore plus de désynchronisation

- 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 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

Jeu de la vie
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.



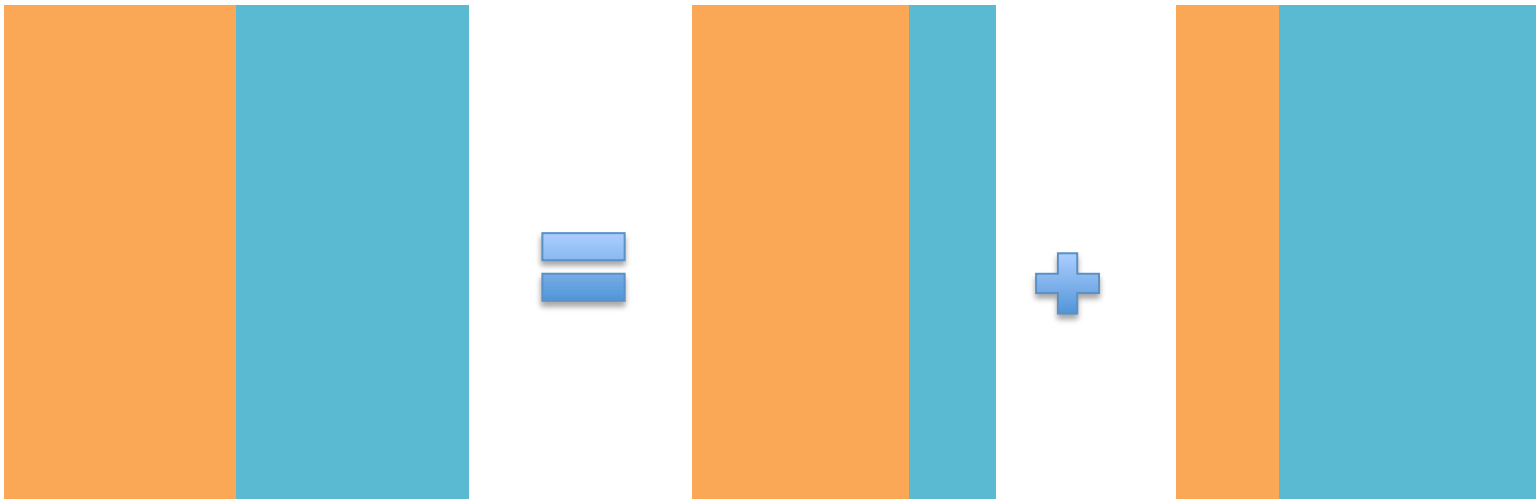
Jeu de la vie
Encore plus de désynchronisation

- 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.

  - Idée : remplacer des synchronisations par du calcul redondant

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

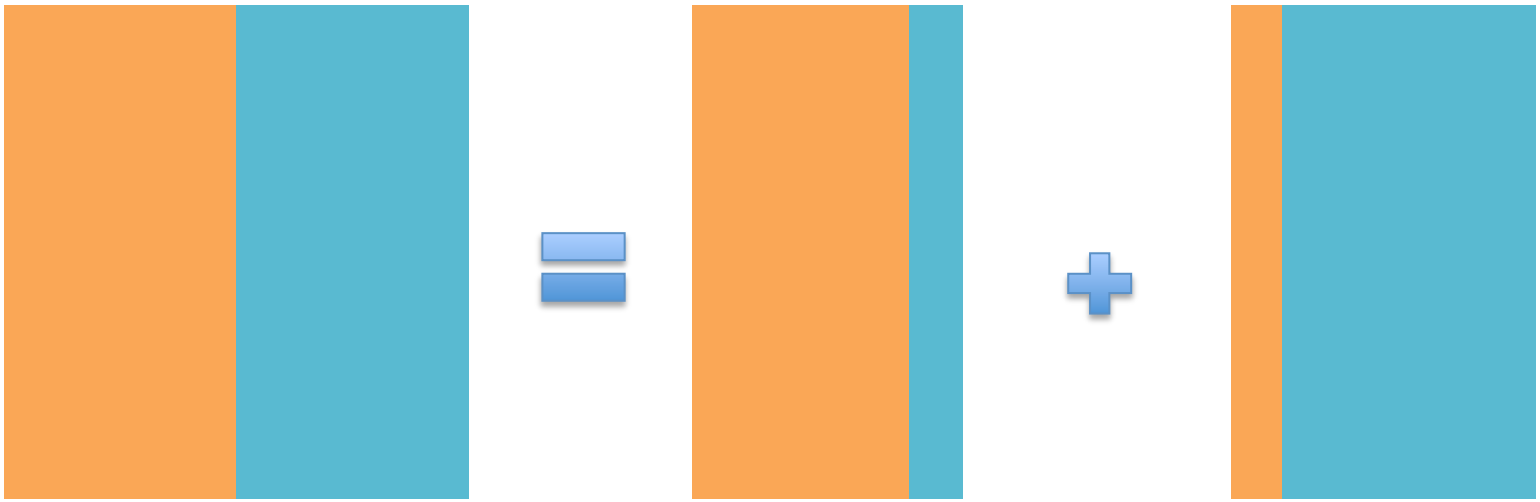
- Dupliquer la zone frontière voisine
  - épaisseur k permet de calculer k étapes sans synchronisation



- Étape 1

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

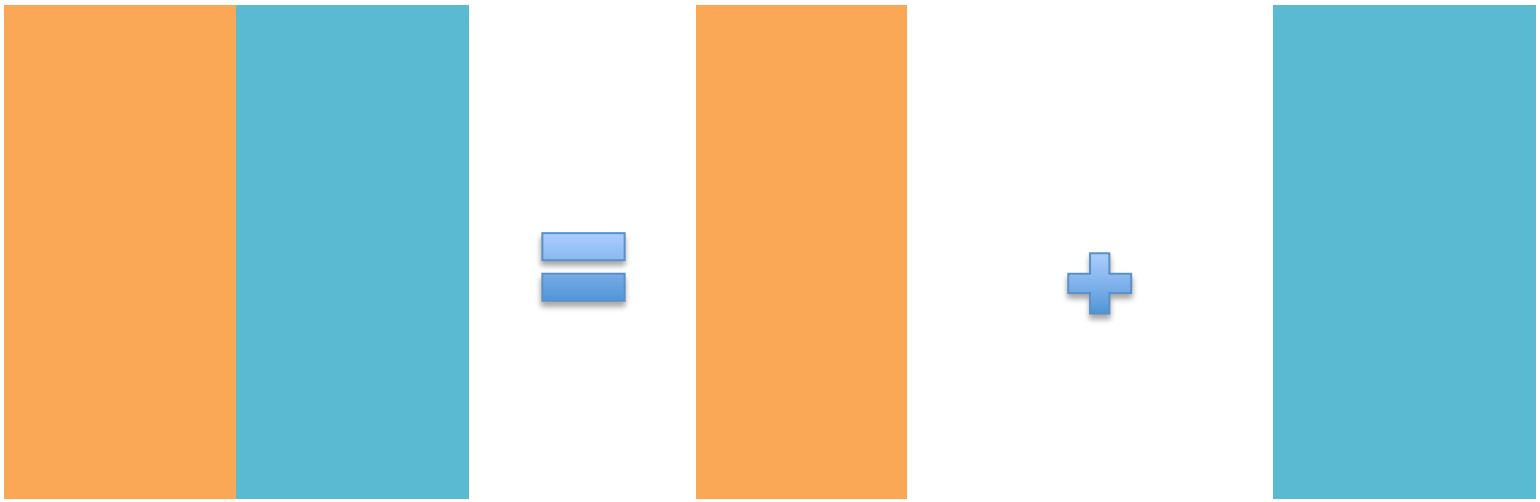
- Dupliquer la zone frontière voisine
  - épaisseur k permet de calculer k étapes sans synchronisation



- Étape 2

Introduction d'une zone de recouvrement (shadow zone)

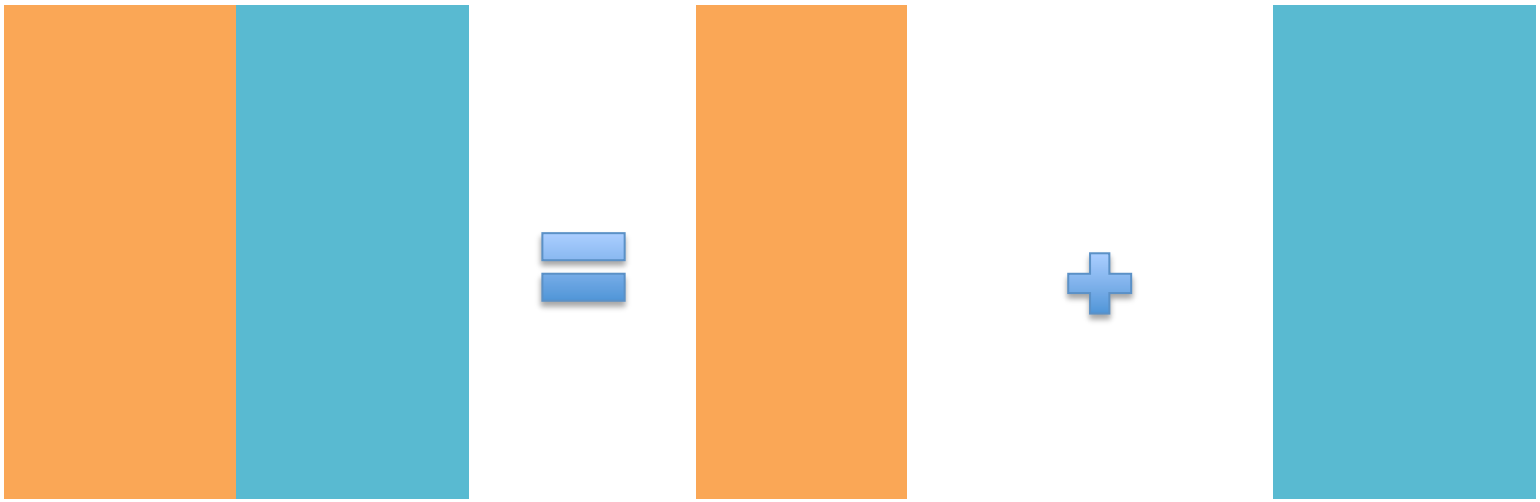
- Dupliquer la zone frontière voisine
  - épaisseur k permet de calculer k étapes sans synchronisation



- Étape 3

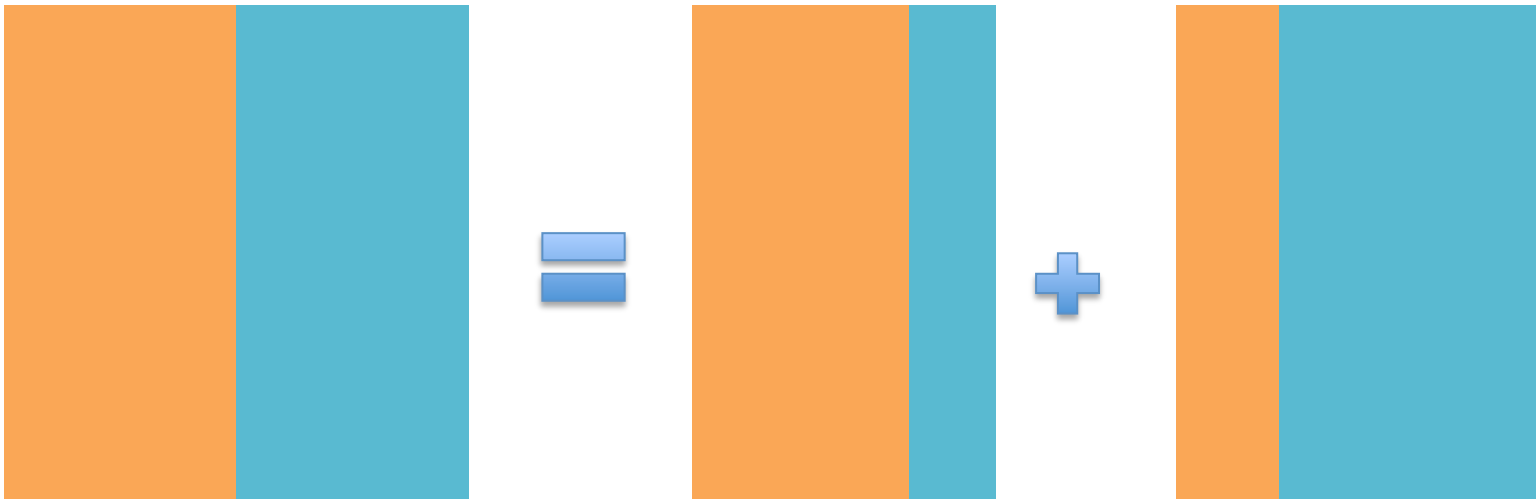
Introduction d'une zone de recouvrement (shadow zone)

- Dupliquer la zone frontière voisine
  - épaisseur k 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



Conclusion sur le chapitre

- Programmer avec les threads
  - C'est bien car
    - On contrôle 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
- Synchronisation / calcul redondant / Mémoire
  - Un compromis subtil