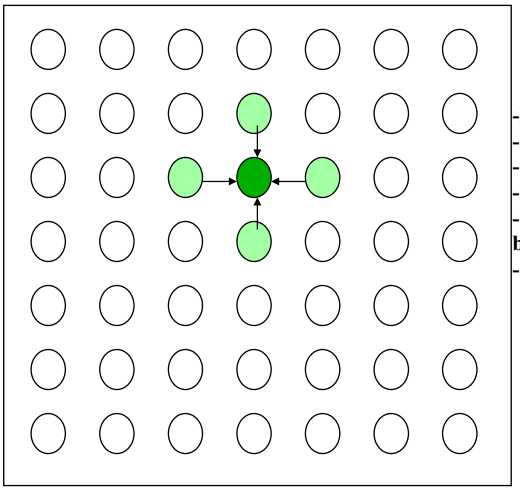
Exemple de programmation multiflot

L' algorithme de Gauss_Seidel

$$A[i,j] = 0{,}2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$$



-Résolution d'équations aux dérivées partielles
-Méthode des différences finies
-Grille régulière de (n+2)*(n+2)
-Les points de la grille n*n sont calculés
-Parcours de gauche à droite puis de haut en bas.
-L'ordre du calcul n'a pas d'importance.

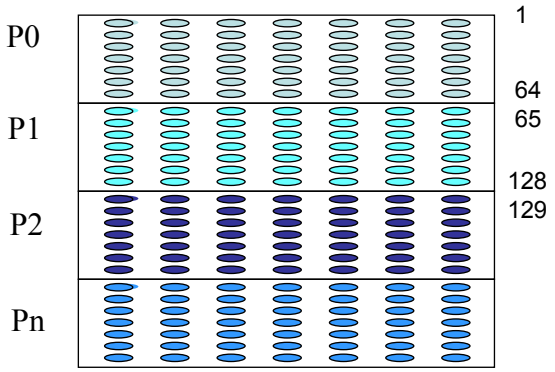
Algorithme séquentiel


```
1     int n;                                          /* matrice de n+2 par n+2*/
2     float **A, diff =0;
3     main ()
4     begin
5           read(n) ;
6           A <- malloc (tableau de n+2*n+2 de doubles) ;
7           init(A);
8           solve(A);
9     end main

10    fonction solve (float **A)
11    begin
12    int i,j, done <-0 ;
13    float diff <-0, temp <-0 ;
14    while (!done) do
15          diff <-0 ;
16          for i<-1 to n do
17                for j<-1 to n do
18                temp <- A[i,j] ;    /* sauvegarde de l'ancienne valeur */
19                 A[i,j] <-0.2 *(A[i,j] + A[i,j-1] + A[i-1,j]+A[i,j+1] + A[i+1,j] )
20                diff += abs(A[i,j]-temp);
21                end for
22          end for
23          if (diff/(n*n )< TOL) done <-1 ;
24    end while
25    end
```

Algorithme parallèle

Soit nprocs le nombre de flots à engendrer : nprocs est un diviseur de la taille de la matrice n.



Chacun des n flots traite un nombre égal de lignes contigües de la grille (64 par exemple). Les flots peuvent être exécutés par n processeurs différents.


```
1       int n , nprocs;              /* Nombre de thread, matrice de n+2 par n+2*/
2       float **A, diff ;            /* variables globales partagées : différence dans un balayage
                                        courant*/
2.a     LOCKDEC(diff_lock) ;         /* déclaration d'un verrou d'exclusion mutuelle */
2.b     BARDEC (bar1) ;              /* déclaration d'une barrière pour synchro globale*/

3       main ()
4       begin
5               read(n) ;
5.1             read(nprocs) ;                  /* nprocs thread */
6               A <- G_MALLOC(n*sizeof(float)) ;
7               initialize(A) ;
8               CREATE (nprocs-1, Solve, A) ;
9               solve(A);                       /* le principal est aussi un thread */
10              WAIT_FOR_END (nprocs-1) ;       /* synchro globale on attend la fin
                                                   tous les fils */
11      end main
```

```
fonction solve (float **A)
11  begin
12  int i,j, done <-0 ; pid <-0 ;                /* données locales à chaque thread de nom  pid */
13  float mydiff <-0, temp <-0 ;
13.1        int mymin =1 + (pid * n/nprocs) ;
13.2        int mymax = mymin +n/nprocs -1 ;/* n est un multiple de nprocs */
14  while (!done) do                             /* chaque thread procède au test de fin */
15          diff = mydiff <-0 ;

16          for i<-mymin to mymax do
17                  for  j<-1 to n do
18                  temp <- A[i,j] ;    /* sauvegarde de l'ancienne valeur */
19                   A[i,j] <-0.2 *(A[i,j] + A[i,j-1] + A[i-1,j]+A[i,j+1] + A[i+1,j] )
20                  mydiff += abs(A[i,j]-temp);
21                  end for
            end for

22.2                diff  += mydiff ;            /* accumulation de tous les mydiff locaux */

23  if (diff/(n*n )< TOL) done <-1 ; /* tous reçoivent la même réponse */

24  end while
```



```
fonction solve (float **A)
11          begin
12          int i,j, done <-0 ; pid <-0 ;            /* données locales à chaque thread de nom  pid */
13          float mydiff <-0, temp <-0 ;
13.1        int mymin =1 + (pid * n/nprocs) ;
13.2        int mymax = mymin +n/nprocs -1 ;/* n est un multiple de nprocs */
14          while (!done) do                         /* chaque thread procède au test de fin */
15                  diff = mydiff <-0 ;
15.1                BARRIER(bar1,nprocs) ; /* tous les threads sont là avant de modifier diff */
16                  for i<-mymin to mymax do
17                          for  j<-1 to n do
18                          temp <- A[i,j] ;    /* sauvegarde de l'ancienne valeur */
19                           A[i,j] <-0.2 *(A[i,j] + A[i,j-1] + A[i-1,j]+A[i,j+1] + A[i+1,j] )
20                          mydiff += abs(A[i,j]-temp);
21                          end for
22                  end for
22.1                LOCK(diff_lock) ;            /* mise à jour du diff global */
22.2                diff  += mydiff ;            /* accumulation de tous les mydiff locaux */
22.3                UNLOCK(diff_lock) ;
22.4                BARRIER(bar, nprocs) ;  /* tous sont là, avant de tester la fin */
23                  if (diff/(n*n )< TOL) done <-1 ; /* tous reçoivent la même réponse */
23.1                BARRIER (bar1, nprocs) ; /* sépare les pas sur la matrice complète */
24          end while
```

Exemple d'implémentation d'une barrière

```
public class Barrier {

        private int m_numThreads = 0 ;
        private int m_currentCount = 0 ;

        public Barrier (int numThreads) {
                m_numThreads = numThreads ;
        }

        public void waitForGo()         {
                synchronized (this)  { /* un seul à la fois peut exécuter la méthode*/
                        m_currentCount++ ;
                        if (m_currentCount < m_numThreads)  {
                          try {
                               wait () ;
                           } catch (interruptedException exp  {
                                   System.out.println (‘’ Exception sur barriere ’’);
                          }
                        else notifyAll() ;
                }
        }
}
```


Section critique avec n-threads : Algorithme du ticket

- Lorsqu'un thread souhaite entrer en section critique il demande un ticket numéroté.
- Les threads sont autorisés à entrer en SC dans l'ordre croissant du numéro de leur ticket.
- L'algorithme utilise deux variables locales *next* et *permit* et des opérations atomiques reposant sur des instructions machines dédiées.

Algorithme

Lorsqu'un thread demande un ticket, il lui est donné le ticket avec le numéro contenu dans *next*, puis *next* est incrémenté de 1. La variable *permit* est initialisée à 1. Un thread peut entrer en section critique lorsqu'il possède un ticket dont le n° est égal à la valeur de *permit*.
permit est incrémenté chaque fois qu'un thread entre en section critique.

```
volatile int number[n] ; // tableau de tickets ; number[i] est le ticket du thread i
volatile long next = 1 ; // prochain n° de ticket donné a un thread
volatile long permit = 1 ; // le n° de ticket qui permet d'entrer en SC
```

Chaque thread Ti exécute le code suivant 0 <= i < n :

```
while (true) {
        number[i] = InterlockedExchangeAdd(&next,1) ;
        while (number[i] != permit) {;}
        ….
        section critique
        ….
        ++permit ;
        section non critique
}
```

La fonction InterlockedExchangeAdd() engendre une instruction machine « barrière de synchronisation » : toutes les opérations anticipées sont terminées avant d'exécuter cette instruction (tous les pipe-line sont vidés et les instructions précédentes sont achevées lorsque cette fonction est exécutée).


```
number[i] = InterlockedExchangeAdd(&next,1) ;
```

Équivalent à :

```
number[i] = next ;
next = next +1 ;
```

Sur les processeurs Intel cette fonction est implémentée par l'instruction atomique XADD (Exchange and Add) qui est aussi une barrière de synchronisation pour garantir la consistance mémoire des accès aux variables communes partagées.

```
XADD (dest,source)
        temp=dest ;
        dest = dest + source ;
        source = temp ;
```

est une instruction atomique

Le problème de l'attente active :

```
while (number[i] != permit) {;}
```

Le test continu de l'égalité consomme des ressources :
solution 1 : l'attente active si le nombre de cycles perdus est faible;
solution 2 : forcer une commutation de threads

```
while (number[i] != permit) {sleep(time);}
```


Sémaphores et verrous

- Les sémaphores sont utilisés pour garantir l'exclusion mutuelle et fournir des conditions de synchronisation
- Les verrous assurent l'exclusion mutuelle mais pas de condition de synchronisation

```
class countingSemaphore {
        public countingSemaphore(int initialPermits) {permits = initialPermits ;}
        public void P() {…} ;
        public void V() {…} ;
        private int permits;
}

countingSemaphore s(1) ;  // déclaration et initialisation d'un sémaphore
```

Utilisation des sémaphores pour la gestion des ressources

- Problème : trois threads se partagent l'utilisation de deux ressources ; si aucune ressource n'est disponible, le thread demandeur doit attendre une libération d'un des deux autres.

countingSemaphore s(2) ; // Deux ressources disponibles initialement

Thread1	Thread2	Thread3
s.P() ;	s.P() ;	s.P() ;
/* utilisation de la ressource */	/* utilisation de la ressource */	/* utilisation ..*/
s.V() ;	s.V() ;	s.V() ;

- Un thread bloqué sur l'instruction s.P() pourra être débloqué par un autre thread ayant exécuté un s.V() ;
- Les opérations P() et V() réalisent en interne les opérations de réservation, de test de disponibilités et de synchronisation.

Généralisation

```
// variables partagées
int count = 2 ;                          // compteur de ressources disponibles
int waiting = 0 ;                        // nb de threads en attente
// sémaphore d'exclusion mutuelle pour accès aux variables count et waiting
countingSemaphore mutex = new countingSemaphore(1) ;
// sémaphore de gestion des flots bloqués
countingSemaphore resourceAvailable = new contingSemaphore(0) ;
Thread i { // chaque thread exécute le code suivant
mutex.P() ;                              //entrée en SC pour gérer la variable count
 if (count >0) {                         // existe-t-il une ressource disponible?
        count-- ;                        // oui, une de moins est alors dispo
        mutex.V() ;                      // sortie de SC
 }
 else {
        waiting++ ;                      //un autre thread en attente
        mutex.V() ;                      // sortie de SC
        resourceAvailable.P() ;          // attente d'une ressource
}
/* utilisation de la ressource */
```

(Suite)

```
mutex.P() ;                          //entrée en SC pour gérer la variable waiting
 if (waiting >0) {                   // existe-t-il des threads en attente ?
          -- waiting ;               // oui, un de moins en attente
          ressourceAvailable.V() ;   //réveille un flot bloqué
}
else count++ ;                       //rajoute une ressource dans le pool
 mutex.V() ;
}
```


Pattern de conception

- Section Critique : mutex
  - P() et V() assurent un et seul thread dans une SC à la fois.
- Entrée-&-Test
  - Soit un test de condition qui concerne des variables partagées soit une autre opération P() qui bloque le thread tant qu'il n'a pas reçu une opération V() qui matérialise la disponibilité de la ressource.
- SortieSC-avant-Attente
  - Avant de se bloquer sur une opération P() en attente de la ressource, il doit y avoir sortie de SC pour éviter un interblocage.
- Gestion de files sur condition
  - Le sémaphore implémente une file de threads en attente qu'une condition devienne vrai : une opération P() bloquera un thread appelant et une opération V() débloquera un thread en attente.

Les verrous

```
Thread1                          Thread2
mutex.P() ;                      mutex.P() ;
/* SC */                         /*SC*/
mutex.V() ;                      mutex.V() ;
```

```
mutexLock mutex ;

Thread1                          Thread2
mutex.lock() ;                   mutex.lock();
/* SC */                         /* SC */
mutex.unlock();                  mutex.unlock();
```


- Exemple 3 revu

```
pthread_mutex_ t mutex  ;
pthread_mutex_init(&mutex,&init);
compte_client = 0 ;
        /* le code des tâches */
```

```
void f(void) {
    int x = 0;        /*  variable locale */
    int j;
    for (j=0 ; j < 16 ; j++) {
    pthread_mutex_lock(&mutex) ;
    //P(&S)
          x = compte_client;          /*P1
          x = x +1 ;                  /*P2
          compte_client = x ;         /*P3
    pthread_mutex_unlock(&mutex) ;
    //V(&S)
    }
}
```

```
void g(void) {
    int y = 0;        /*  variable locale */
    int j;
    for (j=0 ; j < 16 ; j++) {
    pthread_mutex_lock(&mutex) ;
    //P(&S)          y = compte_client;
          /*Q1
          y = y +1 ;                  /*Q2
          compte_client = y ;         /*Q3
    pthread_mutex_unlock(&mutex) ;
    //V(&S)
          }
}
```

P1 Q1 Q2 P2 Q3 P3 Impossible

Q1 Q2 Q3 P1 P2 P3 Résultat = ?

P1 P2 P3 Q1 Q2 Q3 Résultat = ?

Comparaison

- Un thread devient propriétaire du verrou par l'appel de la fonction *mutex.lock*
- Un thread devient propriétaire si aucun autre thread n'est propriétaire
- Un thread libère le verrou par l'opération *unlock*
- Un thread qui possède un verrou et appelle à nouveau la fonction n'est pas bloqué : verrou récursif utile en programmation objet. (Un sémaphore engendrerait un interblocage).
- Le propriétaire pour des appels successifs de lock et unlock doit être le même thread ; des appels successifs de P() et V() peuvent être réalisés par des threads différents.


```
class lockableObject {
        public void F() {
            mutex.lock();
            …. ;
            mutex.unlock() ;
        }
        public void G() {
            mutex.lock() ;
            …. ;
             F() ;          // la méthode G appelle la méthode F
            … ;             // G n'est pas bloqué
            mutex.unlock() ;
        }
        private mutexLock mutex;
}
```