Exemple d'applications parallèles

Mésocentre de calcul de Franche-Comté

Kamel Mazouzi

01/06/2010

UNIVERSITÉ DE FRANCHE-COMTÉ

mésocentre de calcul de franche-comté

Plan

Traitement des éléments d'un tableau

Multiplication matricielle

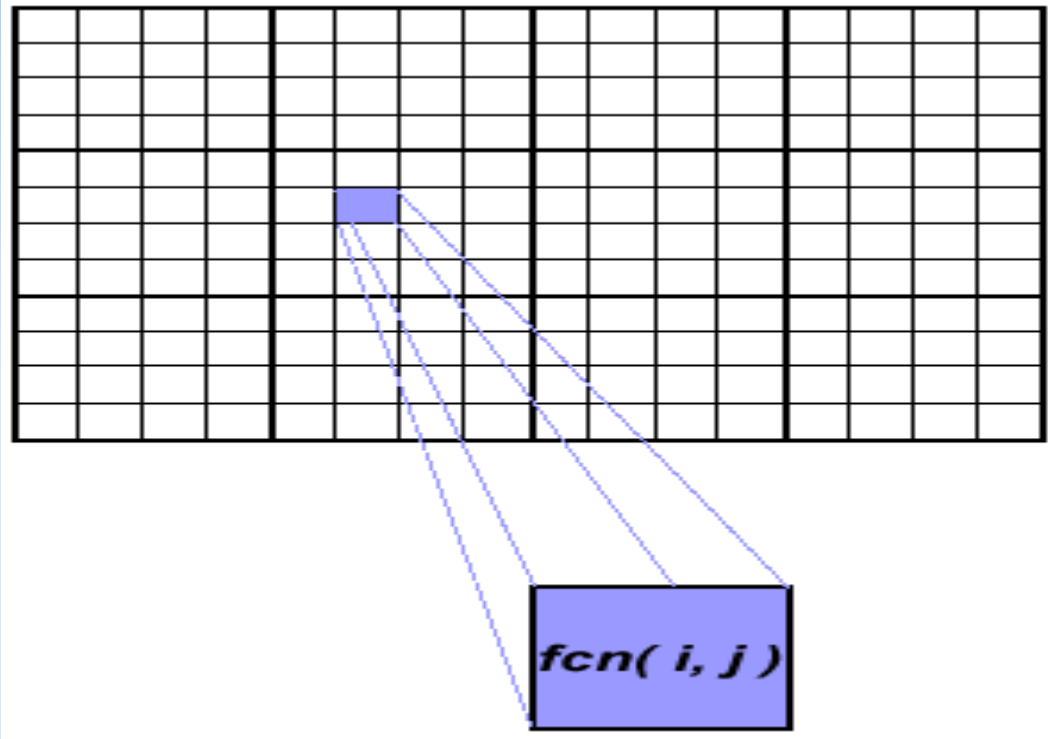
Équation de la chaleur

Traitement des éléments d'un tableau

Traitement des éléments d'un tableau

Version Séquentielle

- Cet exemple illustre le calcul sur les éléments d'un tableau à 2 dimensions
- Le calcul des éléments est indépendant les uns des autres

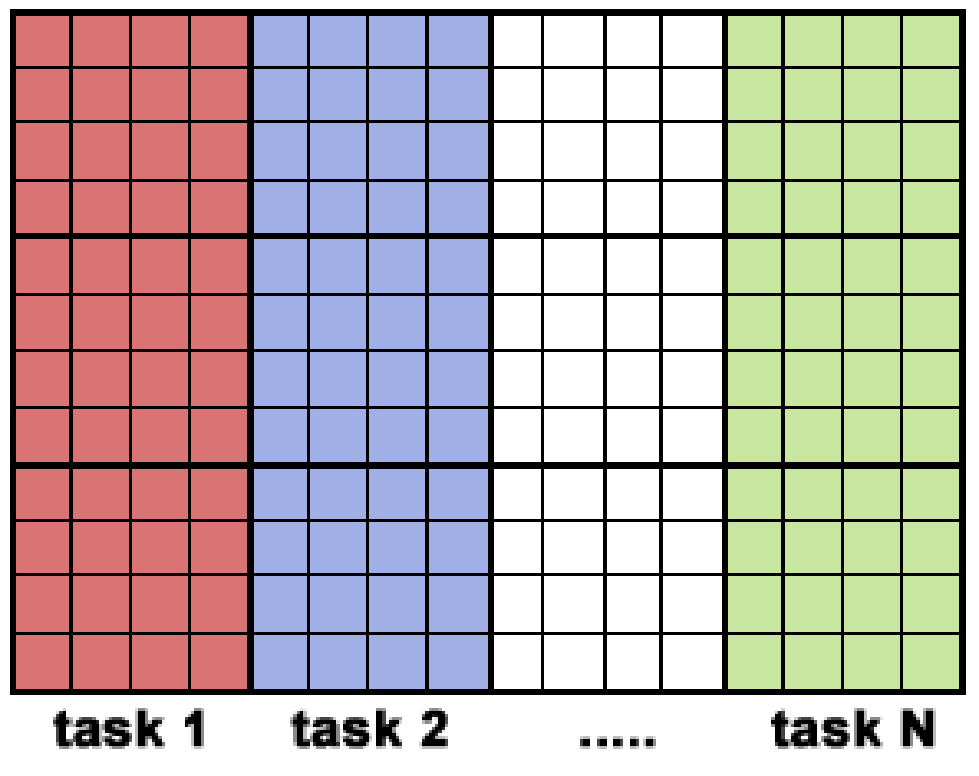



```
do j = 1,n
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```

Traitement des éléments d'un tableaux

Version parallèle

- Les éléments du tableau sont répartis, chaque processeur possède une partie de tableau (sous-tableau)
- Les tâches sont indépendantes (pas de communication entre les tâches)
- Une fois le tableau distribué, chaque tâche exécute la partie de la boucle correspondant aux données qu'elle détient




```
do j = mystart,myend
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```

find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  calculate my portion of array
  do j = my first column,my last column
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
  end do

  send MASTER results

endif

Traitement des éléments d'un tableaux

```
program array
      include 'mpif.h'

      integer   ARRAYSIZE, MASTER
      parameter (ARRAYSIZE = 16000000)
      parameter (MASTER = 0)

      integer  numtasks, taskid, ierr, dest, offset, i, tag1,
     &         tag2, source, chunksize
      real*4   mysum, sum, data(ARRAYSIZE)
      integer  status(MPI_STATUS_SIZE)
      common   /a/ data

C ***** Initializations *****
      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
      i = MOD(numtasks, 4)
      if (i .ne. 0) then
        call MPI_Abort(MPI_COMM_WORLD,ierr)
        stop
      end if
      call MPI_COMM_RANK(MPI_COMM_WORLD, taskid, ierr)
      write(*,*)'MPI task',taskid,'has started...'
      chunksize = (ARRAYSIZE / numtasks)
      tag2 = 1
      tag1 = 2
```

Traitement des éléments d'un tableaux

```
C***** Master task only ******
      if (taskid .eq. MASTER) then

C       Initialize the array
        sum = 0.0
        do i=1, ARRAYSIZE
          data(i) = i * 1.0
          sum = sum + data(i)
        end do
        write(*,20) sum

C       Send each task its portion of the array - master keeps 1st part
        offset = chunksize + 1
        do dest=1, numtasks-1
          call MPI_SEND(offset, 1, MPI_INTEGER, dest, tag1,
     &      MPI_COMM_WORLD, ierr)
          call MPI_SEND(data(offset), chunksize, MPI_REAL, dest,
     &      tag2, MPI_COMM_WORLD, ierr)
          write(*,*) 'Sent',chunksize,'elements to task',dest,
     &      'offset=',offset
          offset = offset + chunksize
        end do
```

Traitement des éléments d'un tableaux

```
C       Master does its part of the work
        offset = 1
        call update(offset, chunksize, taskid, mysum)

C       Wait to receive results from each task
        do i=1, numtasks-1
          source = i
          call MPI_RECV(offset, 1, MPI_INTEGER, source, tag1,
     &      MPI_COMM_WORLD, status, ierr)
          call MPI_RECV(data(offset), chunksize, MPI_REAL,
     &      source, tag2, MPI_COMM_WORLD, status, ierr)
        end do

C       Get final sum and print sample results
        call MPI_Reduce(mysum, sum, 1, MPI_REAL, MPI_SUM, MASTER,
     &    MPI_COMM_WORLD, ierr)
        print *, 'Sample results:'
        offset = 1
        do i=1, numtasks
          write (*,30) data(offset:offset+4)
          offset = offset + chunksize
        end do
        write(*,40) sum

       end if
```

Traitement des éléments d'un tableaux

```
C       Master does its part of the work
        offset = 1
        call update(offset, chunksize, taskid, mysum)

C       Wait to receive results from each task
        do i=1, numtasks-1
          source = i
          call MPI_RECV(offset, 1, MPI_INTEGER, source, tag1,
     &      MPI_COMM_WORLD, status, ierr)
          call MPI_RECV(data(offset), chunksize, MPI_REAL,
     &      source, tag2, MPI_COMM_WORLD, status, ierr)
        end do

C       Get final sum and print sample results
        call MPI_Reduce(mysum, sum, 1, MPI_REAL, MPI_SUM, MASTER,
     &    MPI_COMM_WORLD, ierr)
        print *, 'Sample results:'
        offset = 1
        do i=1, numtasks
          write (*,30) data(offset:offset+4)
          offset = offset + chunksize
        end do
        write(*,40) sum

       end if
```

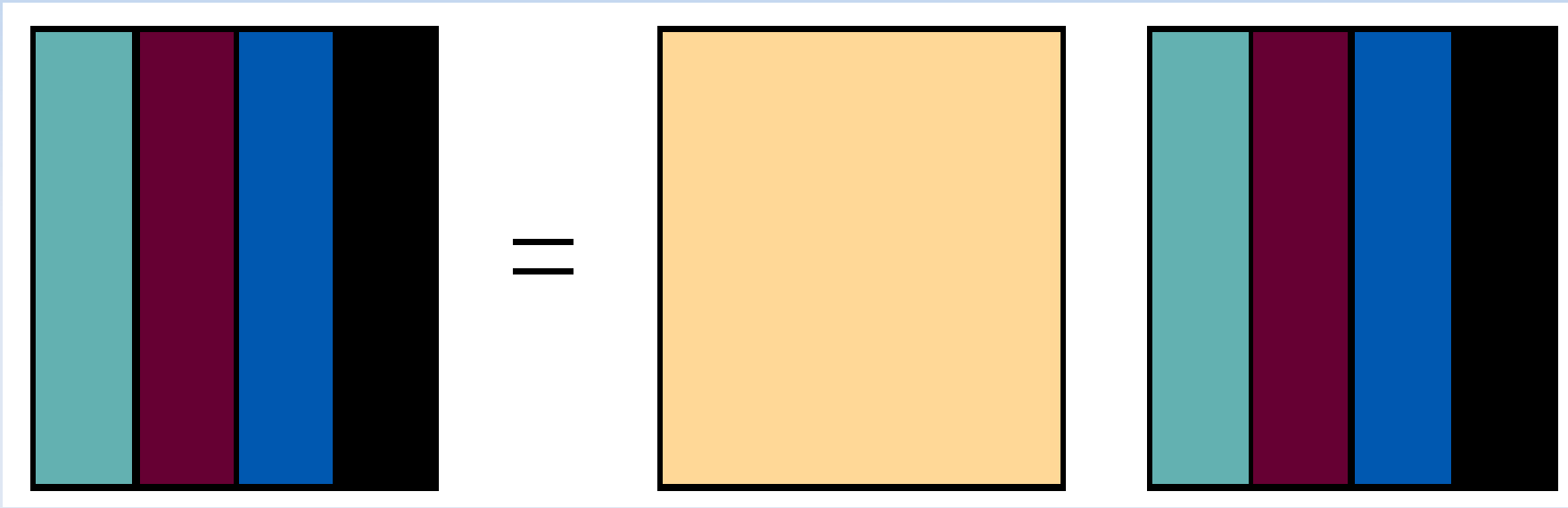
Traitement des éléments d'un tableaux

```
C***** Non-master tasks only *****

      if (taskid .gt. MASTER) then

C       Receive my portion of array from the master task */
        call MPI_RECV(offset, 1, MPI_INTEGER, MASTER, tag1,
     &    MPI_COMM_WORLD, status, ierr)
        call MPI_RECV(data(offset), chunksize, MPI_REAL, MASTER,
     &    tag2, MPI_COMM_WORLD, status, ierr)

        call update(offset, chunksize, taskid, mysum)

C       Send my results back to the master
        call MPI_SEND(offset, 1, MPI_INTEGER, MASTER, tag1,
     &    MPI_COMM_WORLD, ierr)
        call MPI_SEND(data(offset), chunksize, MPI_REAL, MASTER,
     &    tag2, MPI_COMM_WORLD, ierr)

        call MPI_Reduce(mysum, sum, 1, MPI_REAL, MPI_SUM, MASTER,
     &    MPI_COMM_WORLD, ierr)

      endif
      call MPI_FINALIZE(ierr)

  20  format('Initialized array sum = ',E12.6)
  30  format(5E14.6)
  40  format('*** Final sum= ',E12.6,' ***')

      end
```

Multiplication matricielle

Multiplication matricielle

A = B*C



Décomposition des matrices, A = B * C, en Fortran 90

Multiplication matricielle

- **Algorithme :**
  - Distribuer les colonnes de **C** entre les processeurs
  - Diffuser de la matrice B sur tout les processeurs
  - Effectuer la multiplication de **B** avec les colonnes de **C** sur chaque processeur
  - Rassembler les colonnes de A sur un seul processeur

Matrix-vector Multiplication (Columnwise)

```
//MASTER
/***************************** master task ************************************/
   if (taskid == MASTER)
   {
      for (i=0; i<NRA; i++)
         for (j=0; j<NCA; j++)
            a[i][j]= i+j;

       for (i=0; i<NCA; i++)
         for (j=0; j<NCB; j++)
            b[i][j]= i*j;

      /* Send matrix data to the worker tasks */
      averow = NRA/numworkers;
      extra = NRA%numworkers;
      offset = 0;
      mtype = FROM_MASTER;
      for (dest=1; dest<=numworkers; dest++)
      {
         rows = (dest <= extra) ? averow+1 : averow;
         printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
         MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
         MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
                  MPI_COMM_WORLD);
         MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
         offset = offset + rows;
      }
```

Matrix-vector Multiplication (Columnwise)

```
//MASTER
/* Receive results from worker tasks */
      mtype = FROM_WORKER;
      for (i=1; i<=numworkers; i++)
      {
         source = i;
         MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
         MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
         MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
                  MPI_COMM_WORLD, &status);
         printf("Received results from task %d\n",source);
      }

      /* Print results */
      printf("******************************************************\n");
      printf("Result Matrix:\n");
      for (i=0; i<NRA; i++)
      {
         printf("\n");
         for (j=0; j<NCB; j++)
            printf("%6.2f   ", c[i][j]);
      }
      printf("\n******************************************************\n");
      printf ("Done.\n");
```

Matrix-vector Multiplication (Columnwise)

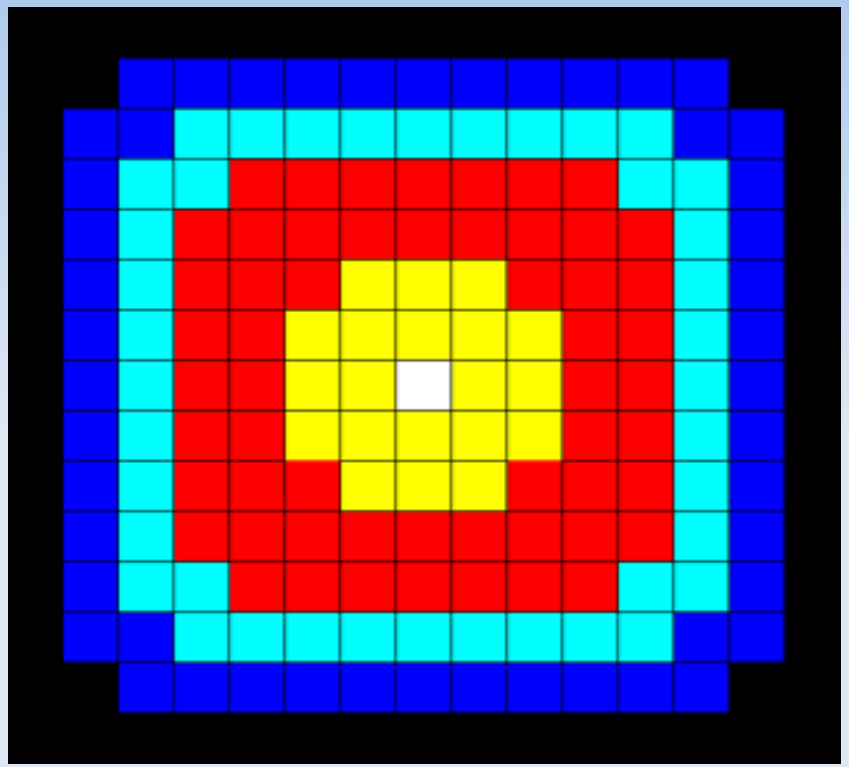
```
//WORKER
/**************************** worker task ************************************/
   if (taskid > MASTER)
   {
      mtype = FROM_MASTER;
      MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

      for (k=0; k<NCB; k++)
         for (i=0; i<rows; i++)
         {
            c[i][k] = 0.0;
            for (j=0; j<NCA; j++)
               c[i][k] = c[i][k] + a[i][j] * b[j][k];
         }
      mtype = FROM_WORKER;
      MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
      MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
   }
   MPI_Finalize();
```

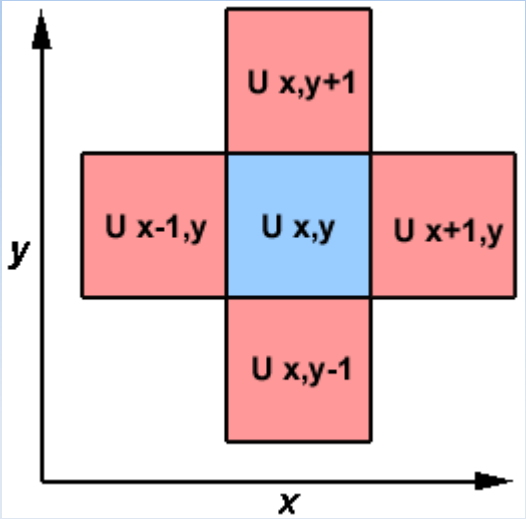
Équation de la chaleur

Équation de la chaleur

- La plupart des problèmes de calcul parallèle nécessitent la communication entre les tâches (généralement avec les voisins)
- L'équation de la chaleur décrit le changement de température en fonction du temps, en prenant en compte la distribution de la température initiale et les conditions aux limites
- La méthode des différences finies est utilisée pour résoudre l'équation de la chaleur sur une région carrée
- La température initiale est nulle sur les limites et élevée au milieu
- La température aux limites est maintenue à zéro
- Les éléments du tableau représentent la température des points sur la surface
- Le calcul d'un élément dépend des valeurs des éléments voisins

Équation de la chaleur





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

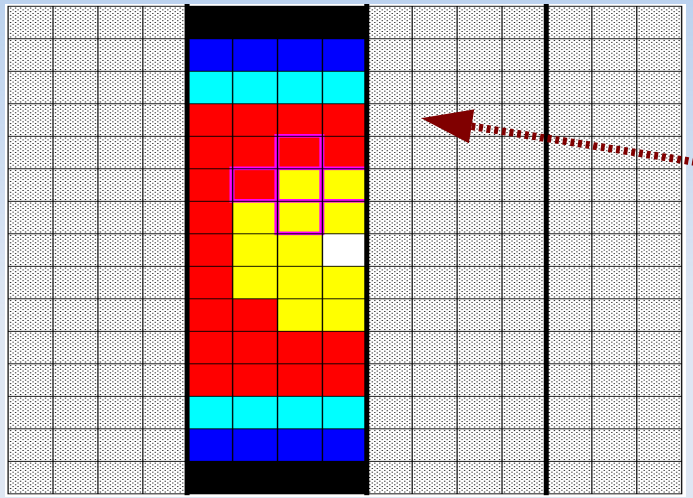
```
do iy = 2, ny - 1
do ix = 2, nx - 1
  u2(ix, iy) =
    u1(ix, iy)  +
      cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy)) +
      cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy))
end do
end do
```

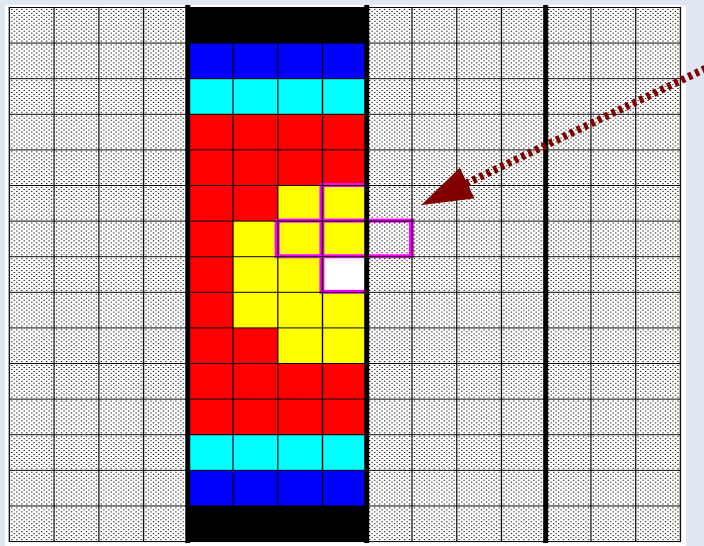
Équation de la chaleur : version parallèle

- Mettre en œuvre un modèle **SPMD**
- Le tableau est découpé et distribué sur toutes les tâches
- Déterminer la dépendance les données :
  - les éléments intérieurs sont indépendants des autres tâches
  - Les éléments de bordure (frontière) dépendent des données d'une tâche voisine (nécessite une communication)
- Le processus maître envoie les informations aux workers, détecte la convergence et collecte les résultats
- Les workers calculent la solution et communiquent au besoin avec les processus voisins

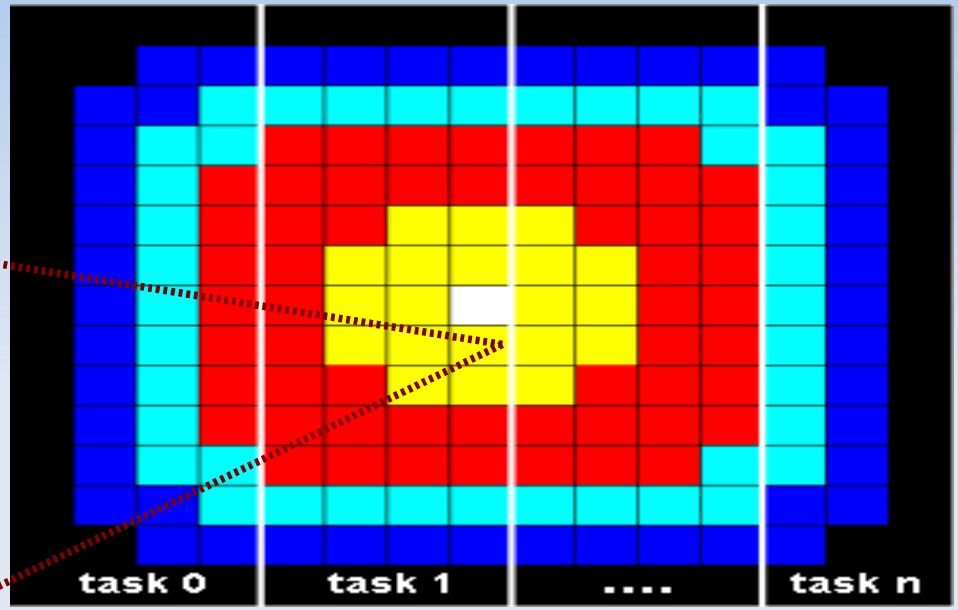
Équation de la chaleur

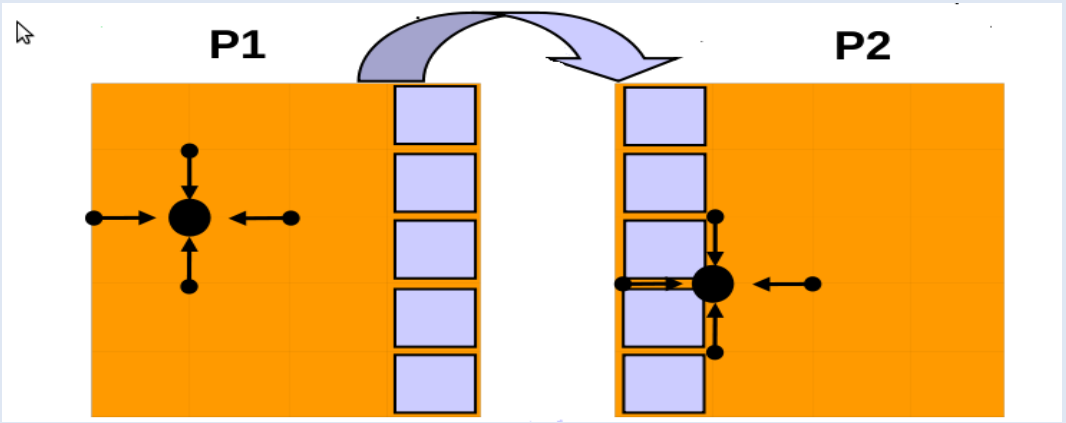
Éléments intérieurs





Éléments frontières





Équation de la chaleur : algorithme SPMD

```
if I am MASTER
  initialize array
  send each WORKER starting info and subarray
 do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do
  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged

    send neighbors my border info
    receive from neighbors their border info
    update my portion of solution array

    determine if my solution has converged
    send MASTER convergence data
    receive from MASTER convergence signal
  end do

  send MASTER results
endif
```

Équation de la chaleur

```
/// MASTER
/* Distribute work to workers.  Must first figure out how many rows to */
      /* send and what to do with extra rows.  */
      averow = NXPROB/numworkers;
      extra = NXPROB%numworkers;
      offset = 0;
      for (i=1; i<=numworkers; i++)
      {
         rows = (i <= extra) ? averow+1 : averow;
         /* Tell each worker who its neighbors are, since they must exchange */
         /* data with each other. */
         if (i == 1)
            left = NONE;
         else
            left = i - 1;
         if (i == numworkers)
            right = NONE;
         else
            right = i + 1;
         /*  Now send startup information to each worker  */
         dest = i;
         MPI_Send(&offset, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
         MPI_Send(&left, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
         MPI_Send(&right, 1, MPI_INT, dest, BEGIN, MPI_COMM_WORLD);
         MPI_Send(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, dest, BEGIN,
                  MPI_COMM_WORLD);

         offset = offset + rows;
      }
```

Équation de la chaleur

```
/// MASTER
/* Now wait for results from all worker tasks */
      for (i=1; i<=numworkers; i++)
      {
         source = i;
         msgtype = DONE;
         MPI_Recv(&offset, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD,
                  &status);
         MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
         MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source,
                  msgtype, MPI_COMM_WORLD, &status);
      }
  MPI_Finalize();
```

Équation de la chaleur

```
/// WORKER
 /************************** workers code **********************************/
   if (taskid != MASTER)
   {
      /* Initialize everything - including the borders - to zero */
      for (iz=0; iz<2; iz++)
         for (ix=0; ix<NXPROB; ix++)
            for (iy=0; iy<NYPROB; iy++)
               u[iz][ix][iy] = 0.0;

      /* Receive my offset, rows, neighbors and grid partition from master */
      source = MASTER;
      msgtype = BEGIN;
      MPI_Recv(&offset, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&left, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&right, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source, msgtype,
               MPI_COMM_WORLD, &status);
```

Équation de la chaleur

```
/// WORKER
 /**************************** workers code ************************************/
   if (taskid != MASTER)
   {
      /* Initialize everything - including the borders - to zero */
      for (iz=0; iz<2; iz++)
         for (ix=0; ix<NXPROB; ix++)
            for (iy=0; iy<NYPROB; iy++)
               u[iz][ix][iy] = 0.0;

      /* Receive my offset, rows, neighbors and grid partition from master */
      source = MASTER;
      msgtype = BEGIN;
      MPI_Recv(&offset, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&left, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&right, 1, MPI_INT, source, msgtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&u[0][offset][0], rows*NYPROB, MPI_FLOAT, source, msgtype,
               MPI_COMM_WORLD, &status);
```

Équation de la chaleur

```
/// WORKER
      /* Determine border elements.  Need to consider first and last columns. */
      /* Obviously, row 0 can't exchange with row 0-1.  Likewise, the last */
      /* row can't exchange with last+1.  */
      start=offset;
      end=offset+rows-1;
      if (offset==0)
         start=1;
      if ((offset+rows)==NXPROB)
         end--;
      iz = 0;
      for (it = 1; it <= STEPS; it++)
      {
         if (left != NONE)
         {
            MPI_Send(&u[iz][offset][0], NYPROB, MPI_FLOAT, left,
                     RTAG, MPI_COMM_WORLD);
            source = left;
            msgtype = LTAG;
            MPI_Recv(&u[iz][offset-1][0], NYPROB, MPI_FLOAT, source,
                      msgtype, MPI_COMM_WORLD, &status);
         }
```

Équation de la chaleur

```
/// WORKER

 if (right != NONE)
         {
            MPI_Send(&u[iz][offset+rows-1][0], NYPROB, MPI_FLOAT, right,
                     LTAG, MPI_COMM_WORLD);
            source = right;
            msgtype = RTAG;
            MPI_Recv(&u[iz][offset+rows][0], NYPROB, MPI_FLOAT, source, msgtype,
                     MPI_COMM_WORLD, &status);
         }
         /* Now call update to update the value of grid points */
         update(start,end,NYPROB,&u[iz][0][0],&u[1-iz][0][0]);
         iz = 1 - iz;
      }

/* Finally, send my portion of final results back to master */

      MPI_Send(&offset, 1, MPI_INT, MASTER, DONE, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, MASTER, DONE, MPI_COMM_WORLD);
      MPI_Send(&u[iz][offset][0], rows*NYPROB, MPI_FLOAT, MASTER, DONE,
               MPI_COMM_WORLD);
      MPI_Finalize();
```