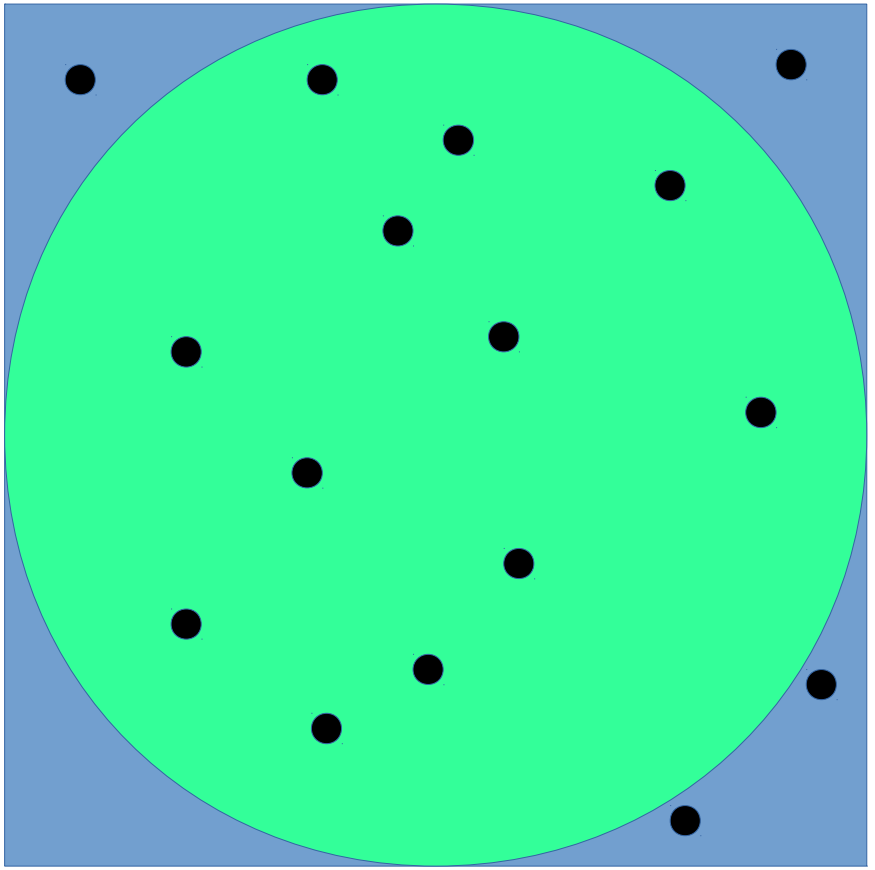
Programmation //

ou « les clones contre-attaquent »

On veut calculer la valeur approchée de π par une méthode stochastique (MC) .



Un joueur maladroit mais uniformément maladroit, joue aux fléchettes.

Si l'on fait le rapport entre le nombre de fléchettes arrivant dans le cercle inscrit et le nombre de fléchettes dans le carré, on converge (lentement) vers le rapport des deux surfaces à savoir π/4

Si le nombre de lancers est très grand nous aurons une estimation de π

Programmation //

- Nous avons plusieurs façons de faire cette simulation :
  - Soit un joueur lance les N fléchettes
  - Soit NP joueurs clonés lancent N/NP fléchettes
- Dans les deux cas le nombre total de fléchettes lancées est le même donc la valeur estimée de π est du même ordre de grandeur, mais le temps nécéssaire dans les deux cas n'est pas le même …..
- Pour que cela soit statistiquement correct il faut que les différents joueurs lancent différemment leurs fléchettes …..ils doivent avoir une petite diversité génétique...

Programmation non-// (1 joueur)

```c
#include <stdio.h>
#include <stdlib.h>

void srandom (unsigned seed);
double dboard (int darts);

#define DARTS     2000000    /* Nombre de lancer de fléchettes dans la cible par tour  */
#define ROUNDS 1000      /* Nombre de tours de lancers */

int main (int argc, char *argv[])
{
   double        homepi,    /* valeur de pi calculée à chaque tour */
                 avepi;     /* valeur moyenne de pi au ième tour */
   Int           taskid=0 ;          /*  graine du générateur pseudo-aléatoire */
   int i;
   /* la graine du générateur pseudo-aléatoire est fixée à la valeur du numéro de tache */
   srandom (taskid);

   avepi = 0;
   for (i = 0; i < ROUNDS; i++)
      {
       homepi = dboard(DARTS);
       avepi = ((avepi * i) + homepi)/(i + 1);
       printf("%8d %10.8f\n", (DARTS * (i + 1)),avepi);
      }
    printf ("\nReal value of PI: 3.1415926535897 \n");

   return 0;
}
```

Programmation non-MPI (1 joueur)

```
double dboard(int darts)
{
#define sqr(x) ((x)*(x))
    long random(void);
    double x_coord, y_coord, pi, r;
    int score, n;
    unsigned int cconst;  /* must be 4-bytes in size */

    /* 2 bit shifted to MAX_RAND later used to scale random number between 0 and 1 */
    cconst = 2 << (31 - 1);
    score = 0;

    /* "throw darts at board" */
    for (n = 1; n <= darts; n++)  {
        /* generate random numbers for x and y coordinates */
        r = (double)random()/cconst;
        x_coord = (2.0 * r) - 1.0;
        r = (double)random()/cconst;
        y_coord = (2.0 * r) - 1.0;

        /* if dart lands in circle, increment score */
        if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
            score++;
    }
    /* calculate pi */
    pi = 4.0 * (double)score/(double)darts;
    return(pi);
}
```

Programmation MPI

Imaginons utiliser 3 Joueurs clonés

Donc le même programme doit être lancé (cloné) 3 fois…

Comme ces « clones » utilisent des zones mémoire non partagées et que l'on veut échanger des informations entre ces trois « clones » il va falloir établir entre eux des communications explicites (**dire qui envoie quoi et à qui d'autre**). Il faudra donc les distinguer entre eux **(leur donner un numéro et peut-être désigner un porte-parole ou un chef ?**)

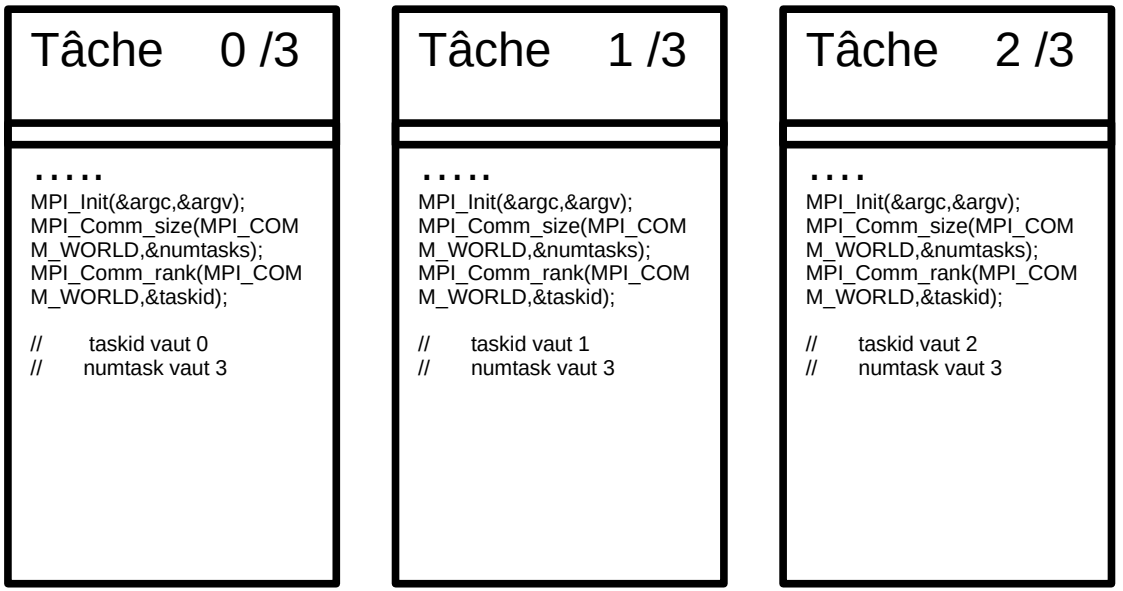
C'est à ça que sert MPI (Message Passing Interface). Cette interface de programmation nous permet de gérer les communications entre tâches (clones) appartenant à un même groupe (on parle de communicateur).

Programmation MPI (3 joueurs)

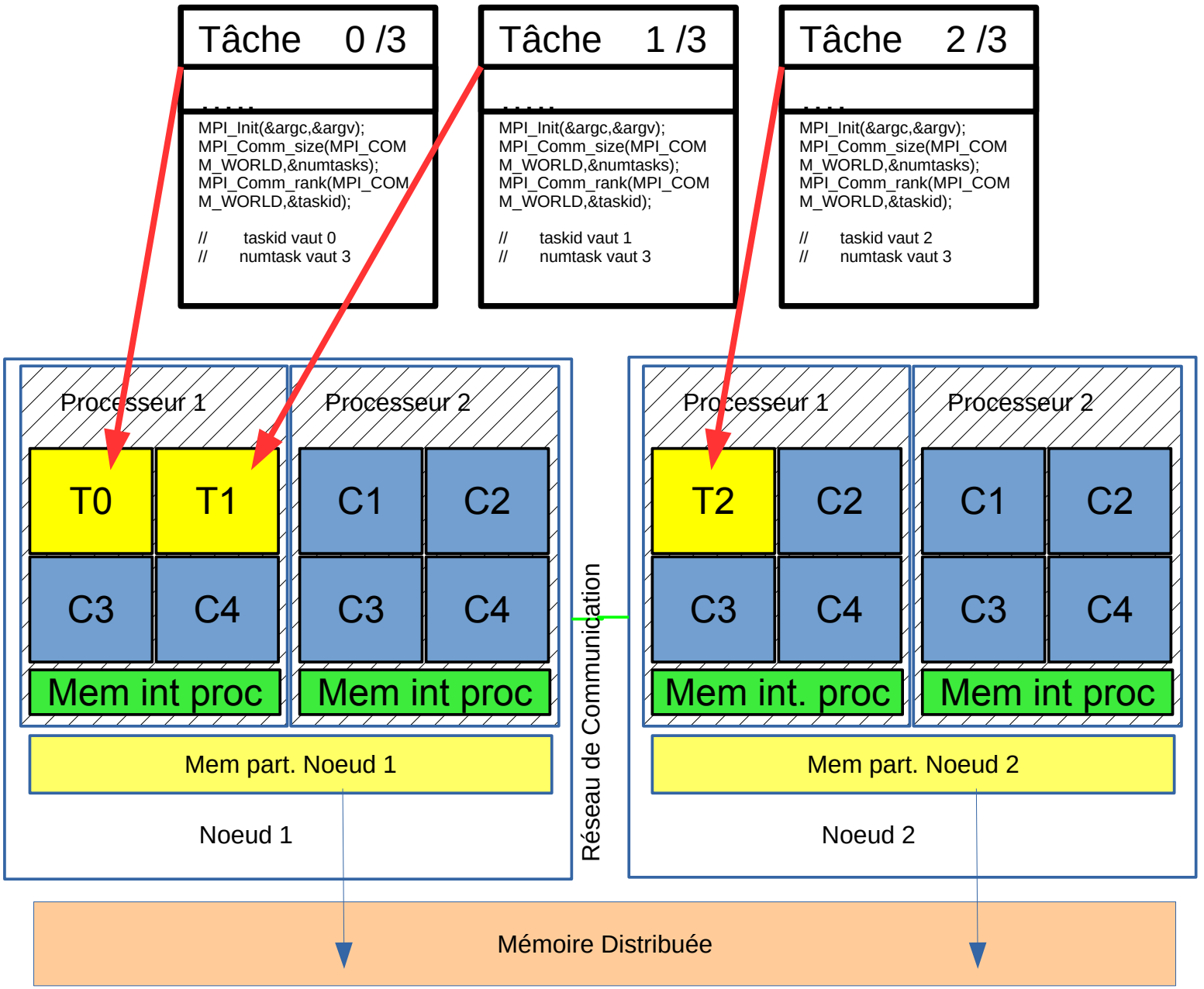
Déroulement du programme :

Au lancement on indique combien de tâches on veut exécuter en parallèle (ici 3 tâches):

le programme est lancé 3 fois (3 tâches) et chaque tâche se verra attribuer un numéro spécifique (**taskid**)



Programmation MPI (3 joueurs)



Programmation MPI (3 joueurs)

Code du programme MPI (début):

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

void srandom (unsigned seed);
double dboard (int darts);

#define DARTS  504000    /* Nombre de lancer de fléchettes dans la cible par tour  */
#define ROUNDS 1000      /* Nombre de tours de lancers */
#define MASTER 0         /* Numéro du « Maître » */



int main (int argc, char *argv[])
{
double homepi,     /* valeur de pi calculée par la tache courante */
       pisum,      /* somme des valeurs de pi de chaque tache  */
       pi,         /* moyenne de pi après chaque tour */
       avepi;      /* average pi value for all iterations */
int    taskid,     /* numéro de la tache  - et aussi graine du générateur aléatoire */
       numtasks,   /* nombre total de taches lancées */
       rc,         /* code de retour */
       i;
MPI_Status status;

/* on initialise les comm's et on récupère le nombre de taches et le numéro de notre tache */

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
printf ("la tache MPI %d à démarré ...\n", taskid);
```

Programmation MPI (3 joueurs)

Code du programme MPI (suite et fin):

```
  /* la graine du générateur pseudo-aléatoire est fixée à la valeur du numéro de tache */
   srandom (taskid);

   avepi = 0;
   for (i = 0; i < ROUNDS; i++) {
     /* toutes les tâches calculent pi par MC  */
     /* chaque tâche prend sa part de travail  DARTS/numtasks lancers /*
     homepi = dboard(DARTS/numtasks);

     /* On utilise MPI_Reduce pour faire la somme des valeurs de homepi calculées par chaque tâche
      * la tâche Maître stockera la valeur de la somme (pisum)
      * - homepi est le buffer d'envoi
      * - pisum est le buffer de réception (utilisé seulement par la tâche qui reçoit)
      * - la taille du message est 1 réel double
      * - MASTER est le numéro de la tâche qui recevra le résultat de l'opération de réduction
      * - MPI_SUM est un drapeau prédéfini qui effectue la somme vectorielle.
      * - MPI_COMM_WORLD est le groupe de communication défini au démarrage des tâches */
    rc = MPI_Reduce(&homepi, &pisum, 1, MPI_DOUBLE, MPI_SUM, MASTER, MPI_COMM_WORLD);
     if (rc != MPI_SUCCESS)
       printf("%d: failure on mpc_reduce\n", taskid);

     /* la tâche maître calcule la moyenne pour cette itération pour les itérations totales*/
     if (taskid == MASTER) {
       pi = pisum/numtasks;
       avepi = ((avepi * i) + pi)/(i + 1);
       printf("   After %8d throws, average value of pi = %10.8f\n", (DARTS * (i + 1)),avepi);
     }
   }
   if (taskid == MASTER)
     printf ("\nReal value of PI: 3.1415926535897 \n");
   MPI_Finalize();
   return 0;
   }
```

Programmation //

OpenMP
+Mémoire partagée

Programmation openMP

En openMP il n'y a au départ qu'un Joueur

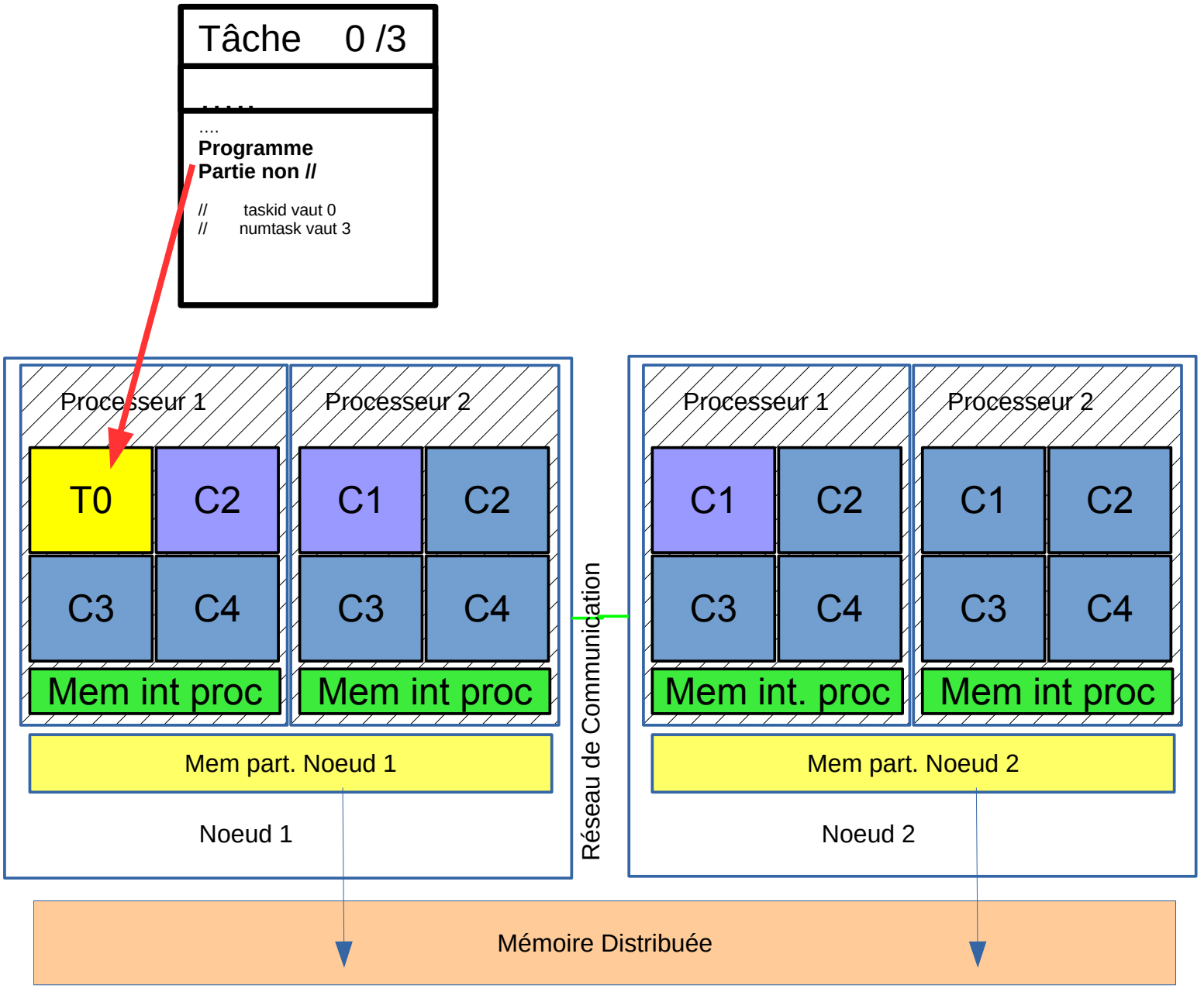
Donc le programme doit être lancé 1 fois…

Au cours du programme (dans la zone //) une partie du code sera dupliquée en NP threads (processus légers) qui effectueront chacun une tache différenciée. Ils **partageront certaines zones mémoire (shared)** et auront **des zones mémoire propres (private).**

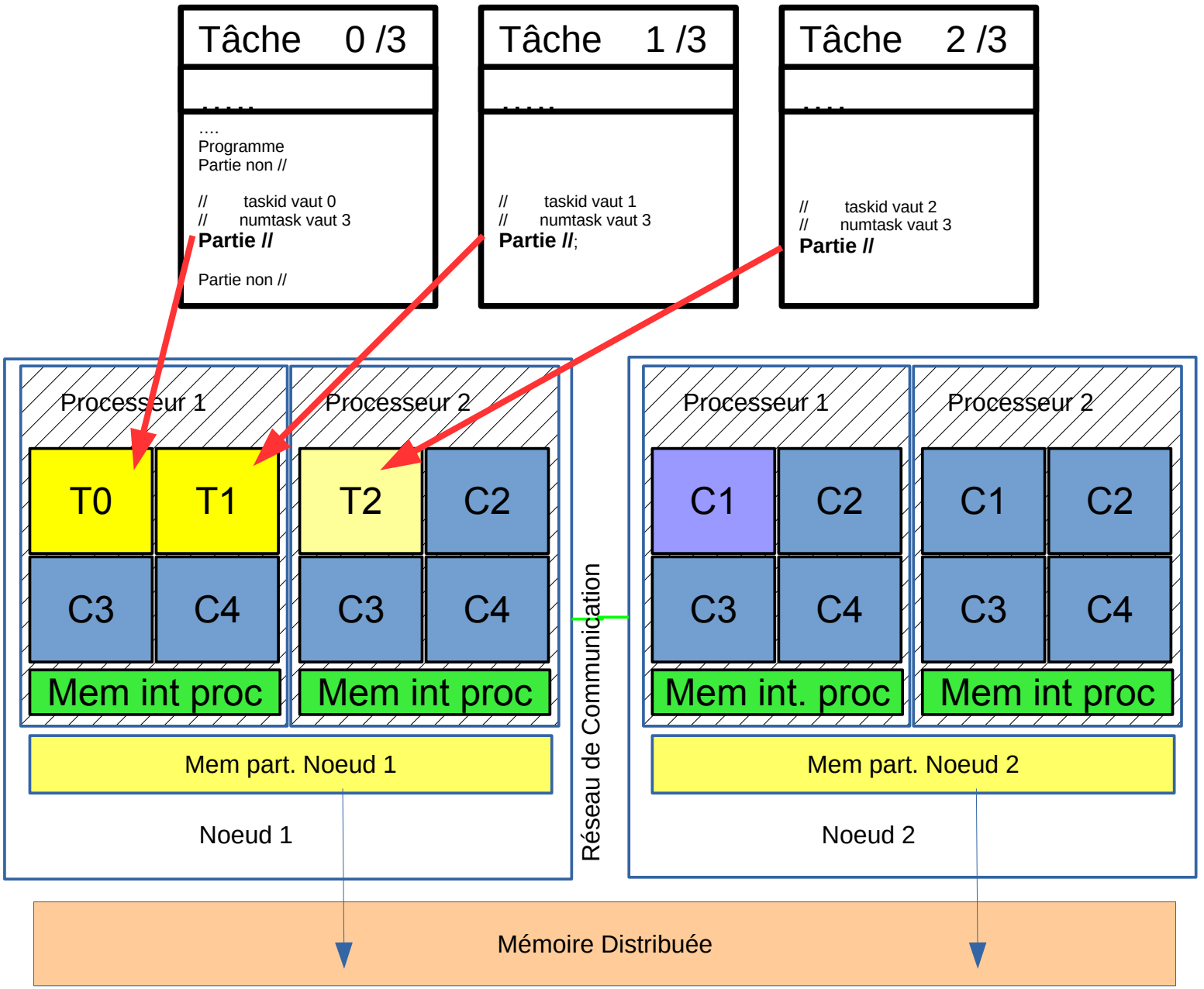
On pourra aussi les distinguer entre eux **(leur donner un numéro et désigner un porte-parole ou un chef?**)

Comme openMP travaille en mémoire partagée il n'y a **pas** besoin de communication **explicite** entre threads mais quelque fois d'une collecte d'information

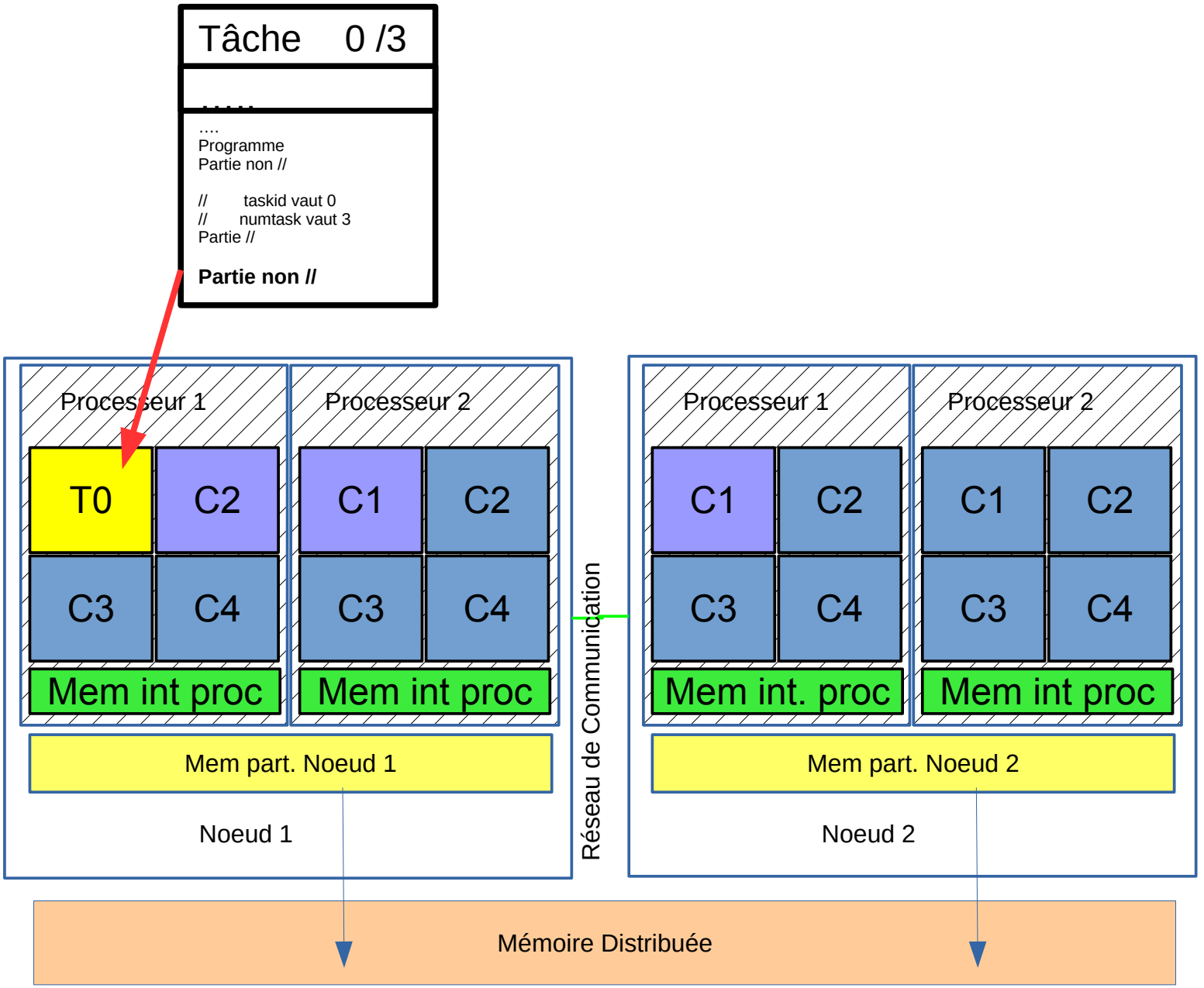
Programmation OpenMP
(1 joueur au départ)



Programmation OpenMP
(3 joueurs pour la partie // du code)



Programmation OpenMP
(1 joueur aprés la partie non // (série))



Programmation openMP

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
double dboard (int darts,struct drand48_data *buf);
#define DARTS  2000000    /* Nombre de lancer de fléchettes dans la cible par tour  */
#define ROUNDS 1000       /* Nombre de tours de lancers */
#define MASTER 0          /* Numéro du « Maître » */

int main (int argc, char *argv[])
{ double homepi,         /* valeur de pi calculée par la tache courante */
          sumpi=0,
          pi,
          avepi;          /* valeur moyenne de pi  */
   int Nb_taches=omp_get_num_threads();
   struct drand48_data drand_buf;
#pragma omp parallel private(drand_buf)
   {
   // zone paralelle
   /* la graine du générateur pseudo-aléatoire est fixée à la valeur du numéro de tâche */
   int taskid = omp_get_thread_num();
   double r;
   //srand48(taskid);
   int seed =1202107158+ taskid *1999;
   srand48_r(seed,&drand_buf);
#pragma omp barrier
   drand48_r(&drand_buf,&r);
   printf(" alea %f tâche %i \n",r,taskid);
   Avepi = 0;

#pragma omp for private(homepi)
   for (int i = 0; i < ROUNDS; i++)
      {
      if(sumpi!=0) printf("la tâche %i a un sumpi diff de 0\n",taskid);
      homepi = dboard(DARTS,&drand_buf);
      sumpi +=homepi;
      if (taskid==MASTER)
        {avepi = ((avepi * i) + sumpi/Nb_taches)/(i + 1);
         printf("%8d %10.8f %10.8f \n",(DARTS * Nb_taches*(i + 1)),avepi,sumpi/Nb_taches);
        }
      sumpi=0;
      }
   }
   // partie  non parallèle (série)
   printf ("\nReal value of PI: 3.1415926535897 \n");
   return 0;}
```

Programmation openMP

```
double dboard(int darts,struct drand48_data *buf)
{
#define sqr(x) ((x)*(x))
//        long random();

        double x_coord, y_coord, pi, r;
        int score, n;
             score = 0;

        /* "throw darts at board" */
        for (n = 1; n <= darts; n++)  {
            /* generate random numbers for x and y coordinates */
            //r=drand48();
            drand48_r(buf ,&r);
            x_coord = (2.0 * r) - 1.0;
            // r=drand48();
            drand48_r(buf ,&r);
            y_coord = (2.0 * r) - 1.0;

            /* if dart lands in circle, increment score */
            if ((sqr(x_coord) + sqr(y_coord)) <= 1.0)
                score++;
        }

    /* calculate pi */
    pi = 4.0 * (double)score/(double)darts;
    return(pi);
}
```