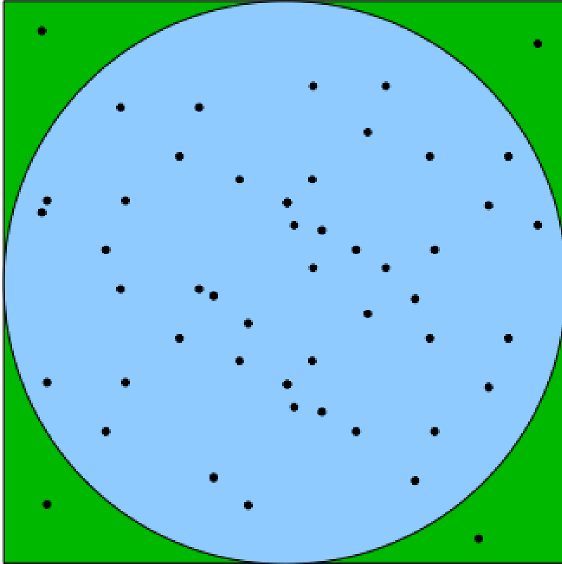
Parallélisation avec MPI :

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

Cible de Fléchettes

/



Soit un joueur de fléchettes particulièrement maladroit mais constant dans sa maladresse (il a la même probabilité d'atteindre toute zone du carré).

Si l'on étudie le rapport du nombre de lancers dans le cercle sur le nombre de lancer dans le carré en fonction du nombre total de lancers on s'aperçoit que ce rapport tend vers π/4 (très lentement !!!)

On se propose de réaliser ce calcul en utilisant la parallélisation avec MPI :

Le programme en C si dessous (pour la paternité voir AUTHOR) a été modifié pour pouvoir comparer les performances de calcul en fonction du nombre de processeurs utilisés.

Les lignes sur-lignées en jaune sont

- soit des commandes spécifiques MPI

- soit des portions de code qui seront effectuées spécifiquement sur le processus Maître (0)

```
**********************************************************************
 * FILE: mpi_pi_reduce.c
 * OTHER FILES: dboard.c
 * DESCRIPTION:
 *   MPI pi Calculation Example - C Version
 *   Collective Communication example:
 *   This program calculates pi using a "dartboard" algorithm.  See
 *   Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
 *   page 207.  All processes contribute to the calculation, with the
 *   master averaging the values for pi. This version uses mpc_reduce to
 *   collect results
 * AUTHOR: Blaise Barney. Adapted from Ros Leibensperger, Cornell Theory
 *   Center. Converted to MPI: George L. Gusciora, MHPCC (1/95)
* LAST REVISED: 06/13/13 Blaise Barney
* Traduction en français des commentaires et modification pour travailler
* à nombre total constant de lancers (IL 2017)
**********************************************************************/
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

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 */
#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 tâche MPI %d a démarré ...\n", taskid);

/* la graine du générateur pseudo-aléatoire est fixée à la valeur du numéro de tache */
srand48(taskid);
printf(" alea %f tâche %i \n",drand48(),taskid);
MPI_Barrier(MPI_COMM_WORLD);
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 *numtasks* (i + 1)),avepi);
   }
}
if (taskid == MASTER)
   printf ("\nReal value of PI: 3.1415926535897 \n");

MPI_Finalize();
return 0;
}
```

```
/*************************************************************************
* subroutine dboard
* DESCRIPTION:
*   Used in pi calculation example codes.
*   See mpi_pi_send.c and mpi_pi_reduce.c
*   Throw darts at board.  Done by generating random numbers
*   between 0 and 1 and converting them to values for x and y
*   coordinates and then testing to see if they "land" in
*   the circle."  If so, score is incremented.  After throwing the
*   specified number of darts, pi is calculated.  The computed value
*   of pi is returned as the value of this function, dboard.
*
*   Explanation of constants and variables used in this function:
*   darts       = number of throws at dartboard
*   score       = number of darts that hit circle
*   n           = index variable
*   r           = random number scaled between 0 and 1
*   x_coord     = x coordinate, between -1 and 1
*   y_coord     = y coordinate, between -1 and 1
*   pi          = computed value of pi
**************************************************************************/

double dboard(int darts)
{
#define sqr(x)   ((x)*(x))
double x_coord, y_coord, pi, r;
int score, n;

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

Compilation du Code

Dans le répertoire du code (Mpi) tapez les commandes suivantes

$make
$make install

Le Makefile est le suivant

```
EXEC=pi_calc
MPICC=/opt/mpi/bullxmpi/1.2.4.1/bin/mpicc

all: ${EXEC}

pi_calc : pi_calc.c
        ${MPICC} -o pi_calc pi_calc.c

clean:
            rm -f ${EXEC}
install :
      ln -s ${PWD}/${EXEC} ${HOME}/bin/${EXEC}
```

Lancement du Code

Ici sur deux coeurs de calcul :

$ sbatch -n 2 batchplus
batchplus :

```
#!/bin/bash

#SBATCH --job-name pi_calc
#SBATCH --nodes 1
##SBATCH --ntasks 2
#SBATCH --cpus-per-task  1
#SBATCH --partition=r424

# Set OMP_NUM_THREADS to the same value as -c
# with a fallback in case it isn't set.
if [ -n "$SLURM_CPUS_PER_TASK" ] ; then
    export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
else
    export OMP_NUM_THREADS=1
fi

EXEC=~/bin/pi_calc
MPIRUN=/opt/mpi/bullxmpi/1.2.4.1/bin/mpirun

echo ${SLURM_JOB_NUM_NODES} noeuds
echo ${SLURM_NTASKS} tâches

time ${MPIRUN} -n ${SLURM_NTASKS} ${EXEC}
```

Manipulation :

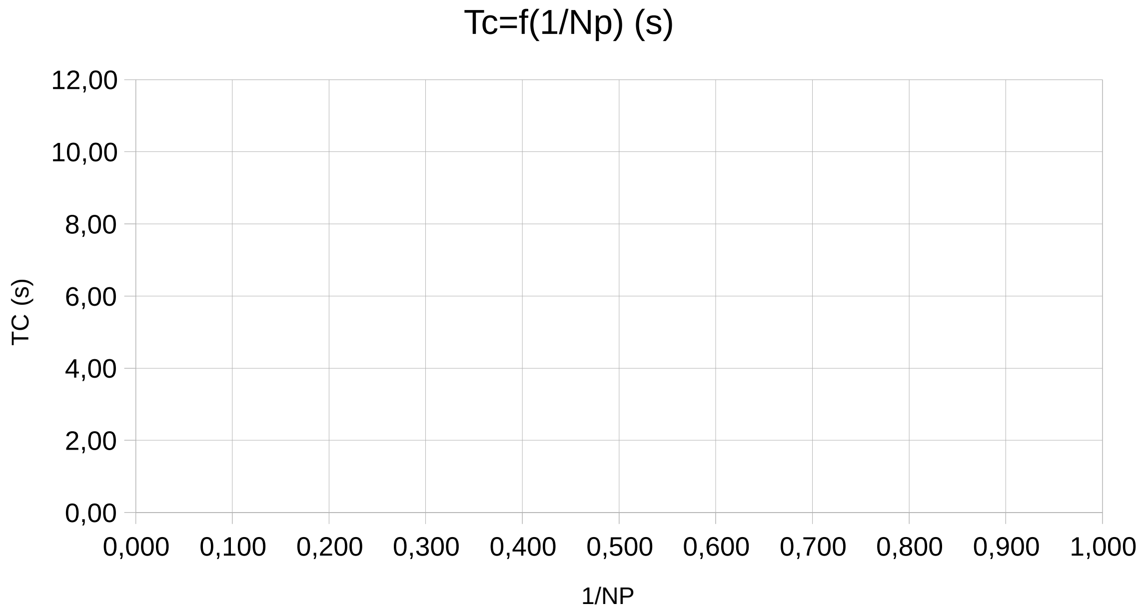
1) Lancez le code avec la commande ***sbatch -n NP batchplus***

2) Notez les temps de calcul pour chaque nombre de processeurs listés dans le tableau ci-dessous

résultats des temps de calcul :

NP	1/NP	TC (s)	TC recalc (s)	Acc	Acc théo
1	1,000				1
2	0,500				2
4	0,250				4
6	0,167				6
7	0,143				7
8	0,125				8
9	0,111				9
10	0,100				10
11	0,091				11
12	0,083				12
18	0,056				18
20	0,050				20

- tracez le graphe Temps de calcul = f(1/ Nprocesseurs)

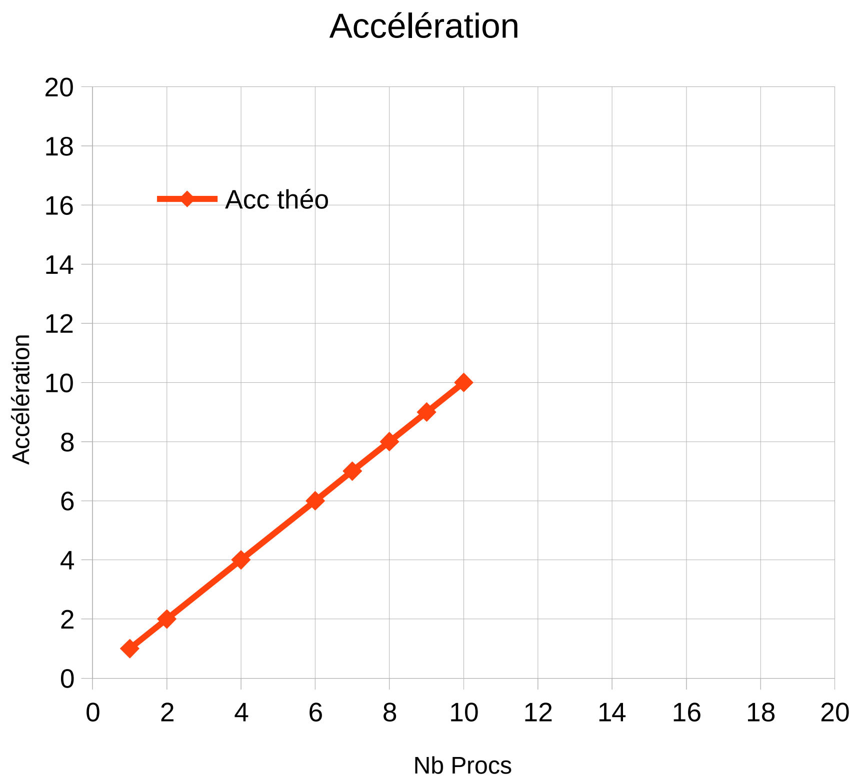


On appelle accélération du code le rapport :

$$A_{(NP)} = \frac{TC(1P)}{TC(NP)}$$

- Remplissez le tableau précédent en calculant ce rapport et faites le graphe A = f(NP).

- Comparez à la valeur idéale de l'accélération théorique (A = NP)



Parallélisation avec OpenMP:

On veut réaliser le même calcul mais en utilisant OpenMP. Le code suivant est la version OpenMP du code précédent :

```
/*********************************************************************
 * FILE: pi_omp.cpp
 * DESCRIPTION:
 *   pi Calculation Example - C Version
 *   This program calculates pi using a "dartboard" algorithm.  See
 *   Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
 *   page 207.  All processes contribute to the calculation, with the
 *   master averaging the values for pi.
 * AUTHOR: Blaise Barney. Adapted from Ros Leibensperger, Cornell Theory
 *   Center. Converted to MPI: George L. Gusciora, MHPCC (1/95)
 * Traduction en français des commentaires et modification pour travailler
 * à nombre total constant de lancers (IL 2017)
 *********************************************************************/
#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);
            printf("%8d %10.8f %10.8f \n",(DARTS * Nb_taches*(i + 1)),avepi,sumpi/Nb_taches);
        }
        sumpi=0;
        }
    }
    // partie monopro
    printf ("\nReal value of PI: 3.1415926535897 \n");

    return 0;
}
```

```
/***************************************************************************
 * subroutine dboard
 * DESCRIPTION:
 *   Used in pi calculation example codes.
 *   Throw darts at board.  Done by generating random numbers
 *   between 0 and 1 and converting them to values for x and y
 *   coordinates and then testing to see if they "land" in
 *   the circle."  If so, score is incremented.  After throwing the
 *   specified number of darts, pi is calculated.  The computed value
 *   of pi is returned as the value of this function, dboard
 *   Explanation of constants and variables used in this function:
 *   darts       = number of throws at dartboard
 *   score       = number of darts that hit circle
 *   n           = index variable
 *   r           = random number scaled between 0 and 1
 *   x_coord     = x coordinate, between -1 and 1
 *   y_coord     = y coordinate, between -1 and 1
 *   pi          = computed value of pi
 ****************************************************************************/

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);
}
```

Compilation du Code

Dans le répertoire du code (OpenMP), tapez les commandes suivantes

$make

$make install

Le Makefile est le suivant

```
EXEC=pi_omp

CC=g++
all: ${EXEC}

pi_omp      : pi_omp.cpp
      ${CC} -O3 -march=native -mtune=native -fopenmp -o ${EXEC} pi_omp.cpp

clean:
      rm -f ${EXEC}

install:
      rm -f ~/bin/${EXEC}
      ln -s ${PWD}/${EXEC} ~/bin/${EXEC}
```

Lancement du Code

Ici sur deux coeurs de calcul :

$ sbatch -c 2 batchplus
batchplus :

```
#!/bin/bash

#SBATCH --job-name pi_openMP
#SBATCH --nodes 1
#SBATCH --ntasks 1
##SBATCH --cpus-per-task  1
#SBATCH --partition=r424

# Set OMP_NUM_THREADS to the same value as -c
# with a fallback in case it isn't set.
if [ -n "$SLURM_CPUS_PER_TASK" ] ; then
    export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
else
    export OMP_NUM_THREADS=1
fi

EXEC=~/bin/pi_omp

echo ${SLURM_JOB_NUM_NODES} noeuds
echo ${SLURM_NTASKS} tâches "(au sens de Mpi)"
echo ${SLURM_CPUS_PER_TASK} "Cpu's par tâche (au sens threads OpenMP)"

time ${EXEC}
```

Manipulation :

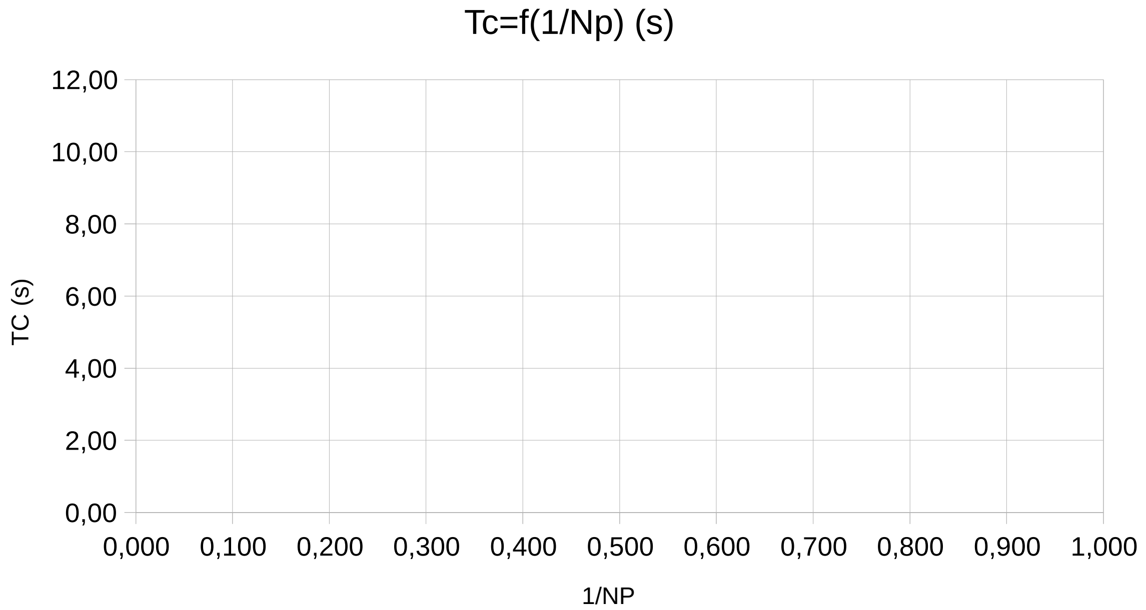
1) Lancez le code avec la commande ***sbatch -c NP batchplus***

2) Notez les temps de calcul pour chaque nombre de processeurs listés dans le tableau ci-dessous

résultats des temps de calcul :

NP	1/NP	TC (s)	TC recalc (s)	Acc	Acc théo
1	1,000				1
2	0,500				2
4	0,250				4
6	0,167				6
7	0,143				7
8	0,125				8
9	0,111				9
10	0,100				10
11	0,091				11
12	0,083				12
13	0,077				13

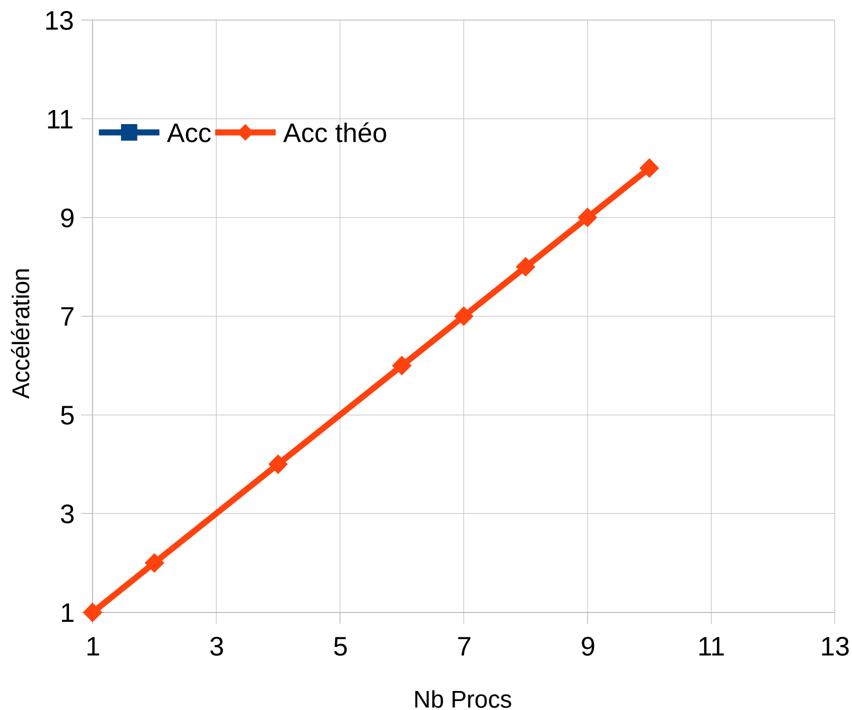
- tracez le graphe Temps de calcul = f(1/ Nprocesseurs)



On appelle accélération du code le rapport :

$$A_{(NP)}=\frac{TC(1P)}{TC(NP)}$$

- Remplissez le tableau précédent en calculant ce rapport et faites le graphe A = f(NP).
- Comparez à la valeur idéale de l'accélération théorique (A = NP)



- Comparez les deux méthodes de parallélisation pour **ce** problème.

- Testez le changement suivant pour les deux programmes avec 6 processeurs utilisés :

```
#define DARTS  20000    /* Nombre de lancer de fléchettes dans la cible par tour  */
#define ROUNDS 100000       /* Nombre de tours de lancers */
```

- Conclusions ?