

**ÉCOLE NATIONALE SUPÉRIEURE DES TECHNIQUES AVANCÉES**  
**DIPLÔME D'INGÉNIEUR**  
**SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA**  
**COMMUNICATION**  
**IN203 - PROGRAMMATION PARALLÈLE**



MACEDO SANCHES Bruno

PROF. DR. JUVIGNY Xavier  
**RAPPORT TRAVAUX DIRIGÉ 1**  
**Échange de messages avec MPI**

PARIS  
14 NOVEMBRE DE 2021

# Índice

1.Objectifs	<b>2</b>
2. Exercices	<b>2</b>

# 1.Objectifs

Cette activité tient comme objective l'application des concepts vus en cours magistral sur la programmation parallèle et de l'interface MPI.

## 2. Exercices

### 1 - Un Hello World parallèle

Pour le premier exercice nous devons compiler et exécuter le programme HelloWorld.cpp fourni ci-dessous.

```
# include <iostream>
# include <cstdlib>
# include <mpi.h>
# include <sstream>
# include <fstream>

int main( int nargs, char* argv[] )
{
    MPI_Init(&nargs, &argv);
    int numero_du_processus, nombre_de_processus;

    MPI_Comm_rank(MPI_COMM_WORLD,
                  &numero_du_processus);
    MPI_Comm_size(MPI_COMM_WORLD,
                  &nombre_de_processus);
    std::cout << "Hello world from "
                << numero_du_processus << " in "
                << nombre_de_processus << " executed"
                << std::endl;
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Ce code initialise l'environnement de MPI et les processus correspondants, chaque processus doit afficher dans le console le message "Hello World from x in y executed". Nous pouvons observer le résultat de l'exécution avec 1, 2, 4 et 16 processus.

```
bruno@bruno-300ESK-300ESQ:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ mpirun --oversubscribe -np 1 ./HelloWorld.exe
Hello world from 0 in 1 executed
bruno@bruno-300ESK-300ESQ:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ mpirun --oversubscribe -np 2 ./HelloWorld.exe
Hello world from 1 in 2 executed
Hello world from 0 in 2 executed
^[[bruno@bruno-300ESK-300ESQ:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ mpirun --oversubscribe -np 4 ./HelloWorld.exe
Hello world from 1 in 4 executed
Hello world from 0 in 4 executed
Hello world from 2 in 4 executed
Hello world from 3 in 4 executed
bruno@bruno-300ESK-300ESQ:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ mpirun --oversubscribe -np 16 ./HelloWorld.exe
Hello world from 1 in 16 executed
Hello world from 5 in 16 executed
Hello world from 13 in 16 executed
Hello world from 12 in 16 executed
Hello world from 15 in 16 executed
Hello world from 2 in 16 executed
Hello world from 6 in 16 executed
Hello world from 0 in 16 executed
Hello world from 7 in 16 executed
Hello world from 8 in 16 executed
Hello world from 3 in 16 executed
Hello world from 9 in 16 executed
Hello world from 10 in 16 executed
Hello world from 14 in 16 executed
Hello world from 4 in 16 executed
Hello world from 11 in 16 executed
bruno@bruno-300ESK-300ESQ:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$
```

Image 1: Capture d'écran d'exécution du programme HelloWorld

Dans l'affichage c'est possible d'observer que l'exécution ne suit pas une ordre exact des rangs des processus.

Ensuite, nous sommes demandés d'implémenter l'affichage sur un fichier différent pour chaque. Le code ci-dessous implémente les changements.

```
# include <cstdlib>
# include <sstream>
# include <string>
# include <fstream>
# include <iostream>
# include <iomanip>
# include <mpi.h>

int main( int nargs, char* argv[] )
{

    MPI_Init( &nargs, &argv );

    MPI_Comm globComm;
    MPI_Comm_dup(MPI_COMM_WORLD, &globComm);

    int nbp;
    MPI_Comm_size(globComm, &nbp);

    int rank;
    MPI_Comm_rank(globComm, &rank);
```

```

std::stringstream fileName;
    fileName << "Output" << std::setfill('0') << std::setw(5) <<
rank << ".txt";
    std::ofstream output( fileName.str().c_str() );

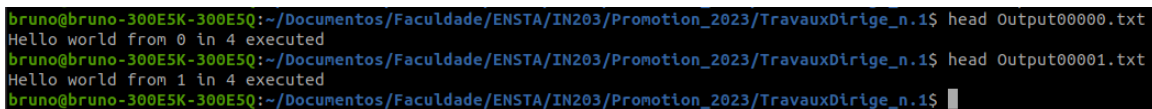
    output << "Hello world from "
            << rank << " in "
            << nbp << " executed"
            << std::endl;

    output.close();

    MPI_Finalize();
    return EXIT_SUCCESS;
}

```

Maintenant, c'est possible de regarder chaque sortie des processus de façon séparée en conséquence des altérations.



```

bruno@bruno-300ESK-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ head Output00000.txt
Hello world from 0 in 4 executed
bruno@bruno-300ESK-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ head Output00001.txt
Hello world from 1 in 4 executed
bruno@bruno-300ESK-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$

```

Image 2: Résultat de l'exécution du programme HelloWorld avec sortie dans fichiers séparés

En raison des sorties séparées, réaliser la débogage devient plus facile en sorte que les sorties ne soient pas toutes ensemble.

## 2 - Envoi bloquant et non-bloquant

Pour cet exercice nous considérons les codes suivants

### Envoi bloquant

```

myvar = 0;
for(i = 1; i < numtasks, ++i) {
    task = i;
    MPI_Send(&myvar, ...,
            ..., task, ...);
    myvar = myvar + 2;
    //Do some works
    ...
    ...
}

```

### Envoi non-bloquant

```

myvar = 0;
for(i = 1; i < numtasks, ++i) {
    task = i;
    MPI_Isend(&myvar, ...,
            ..., task, ...);
    myvar = myvar + 2;
    //Do some works
    ...
    MPI_Wait (...);
}

```

```
}
```

Pour le premier code, l'envoi est de type bloquant et nous sommes sûrs que le code suivant sera exécuté seulement après le fin de la communication, dans le envoi de type non-bloquant, le code suivant sera exécuté sans que le processus aie fini, l'exécution d'une instruction modifiant le buffer (dans ce cas la variable myvar) avant l'envoi va occasionner un changement de la message a envoyer et provoquer un comportement indésirable sur le programme. La solution qui permet de continuer avec un envoi non-bloquant c'est de modifier le code en ajoutant l'instruction `MPI_Wait (...)` avant la modification du buffer, nous pouvons déplacer quelque code qui ne dépend pas de la variable myvar avant l'instruction `MPI_Wait` pour ne perdre pas beaucoup de performance.

### 3 - Circulation d'un jeton dans un anneau

Dans cet exercice nous sommes demandés d'implémenter un algorithme de circulation d'un jeton dans un anneau en utilisant le MPI. Le code développé est fourni ci-dessous. Pour ne pas répéter les parties obligatoires pour le fonctionnement du MPI, seule la partie concernant est fourni.

```
int jeton, received;
MPI_Status status;
if(rank == 0) {
    jeton = 5;

    std::cout << "Processus " << rank
                << " envoie le jeton " << jeton
                << std::endl;

    MPI_Send(&jeton, 1, MPI_INT, (rank+1)%nbp, 0, globComm);
    MPI_Recv(&received, 1, MPI_INT, nbp-1, MPI_ANY_TAG, globComm,
             &status);

    std::cout << "Processus " << rank
                << " reçoit le jeton " << received
                << std::endl;
```

```

}
else {
    MPI_Recv(&received, 1, MPI_INT, rank-1, MPI_ANY_TAG, globComm,
    &status);

    std::cout << "Processus " << rank
                << " reçoit le jeton " << received
                << std::endl;


    jeton = received + 1;

    std::cout << "Processus " << rank
                << " envoie le jeton " << jeton
                << std::endl;

    MPI_Send(&jeton, 1, MPI_INT, (rank+1)%nbp, 0, globComm);
}

```

En exécutant ce code nous pouvons observer la circulation du jeton lors des sorties des programmes.



```

bruno@bruno-300E5K-300E5Q:~/Documents/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$ mplexec --oversubscribe -np 4 ./Circulation_jeton.exe
Processus 0 envoie le jeton 5
Processus 1 reçoit le jeton 5
Processus 1 envoie le jeton 6
Processus 2 reçoit le jeton 6
Processus 2 envoie le jeton 7
Processus 3 reçoit le jeton 7
Processus 3 envoie le jeton 8
Processus 0 reçoit le jeton 8
bruno@bruno-300E5K-300E5Q:~/Documents/Faculdade/ENSTA/IN203/Promotion_2023/TravauxDirige_n.1$

```

Image 3: Sortie du programme en utilisant 4 processus

Dans cet exécution nous pouvons voir que les affichages se passent en ordre de rank sauf pour le processus 0 qui affiche en premier et en dernier en raison d'utilisation de méthodes bloquantes et que l'envoi du jeton dépend de la réception ultérieure.

Après, nous sommes demandés de changer le programme de façon que chaque processus génère son propre jeton et envoie a le processus suivant. Si nous changeons seulement la façon dont le jeton est généré le résultat affiché sera semblable au antérieure, une autre façon de faire c'est d'utiliser méthodes non-bloquantes, cette implémentation est fourni ci-dessous.

```

int jeton, received;
MPI_Status status;
MPI_Request request;

jeton = rank * 9 + 4;

std::cout << "Processus " << rank
           << " envoie le jeton " << jeton
           << std::endl;

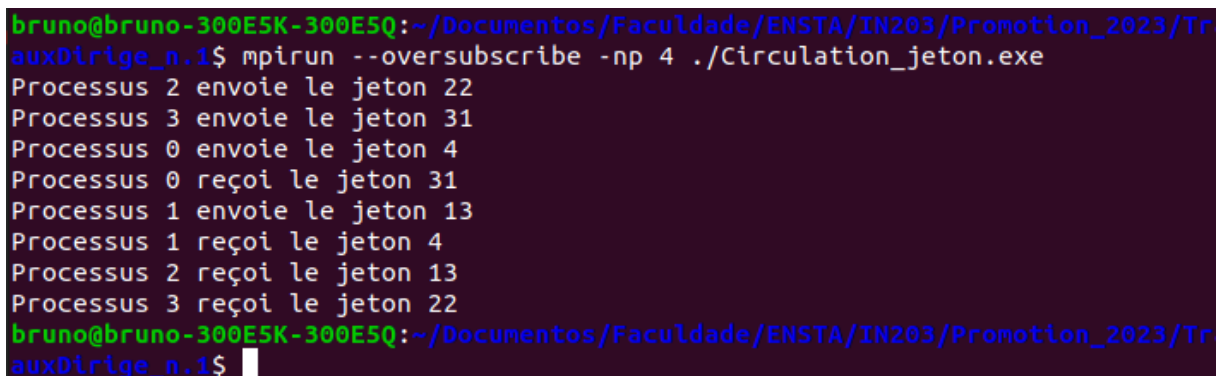
MPI_Isend(&jeton, 1, MPI_INT, (rank+1)%nbp, 0, globComm, &request);
MPI_Recv(&received, 1, MPI_INT, (rank-1 < 0 ? nbp-1 : rank-1),
MPI_ANY_TAG, globComm, &status);

std::cout << "Processus " << rank
           << " reçoit le jeton " << received
           << std::endl;

MPI_Wait(&request, &status);

```

Maintenant, le résultat affiché sur le console ne possède pas une ordre, et des exécutions différents peuvent afficher des résultats différents.



```

bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Tr
auxDirige_n.1$ mpirun --oversubscribe -np 4 ./Circulation_jeton.exe
Processus 2 envoie le jeton 22
Processus 3 envoie le jeton 31
Processus 0 envoie le jeton 4
Processus 0 reçoit le jeton 31
Processus 1 envoie le jeton 13
Processus 1 reçoit le jeton 4
Processus 2 reçoit le jeton 13
Processus 3 reçoit le jeton 22
bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Tr
auxDirige_n.1$

```

Image 5: Résultat Circulation de jeton avec envoi non bloquant.

#### 4- Calcul de pi par lancer de fléchettes

Pour cet exercice nous voulons calculer la valeur de pi avec l'aide d'un algorithme stochastique, soit un carré de côté mesuré  $2r$  et surface de  $4r^2$ , le cercle circonscrit avec un rayon  $r$  et surface de  $\pi r^2$ , le rapport entre la surface de le cercle et du carré est de  $R = \pi / 4$ , donc en utilisant une lois bidimensionnelle uniforme, la probabilité de que un point soit dans le cercle est de  $R$ , en conclusion, c'est possible



de calculer la valeur de pi en utilisant cette méthode et avec le MPI, c'est possible de paralléliser la génération des points dans le cercle.

Le programme développé est fourni ci-dessous, il utilise la fonction fournie par l'exercice de façon à calculer pi adapté pour recevoir un argument indiquant le numéro de processus qui feront le calcul.

```

MPI_Status status;
int nbSamples = 100000000;
if (rank == 0) {
    double pi;
    if(nbp == 1) {
        pi = approximate_pi(nbp, nbSamples);
    }
    else {
        double ratio;
        pi = 0;
        for(int i = 1; i < nbp; i++) {
            MPI_Recv(&ratio, 1, MPI_DOUBLE, i, MPI_ANY_TAG, globComm,
&status);
            pi += ratio;
        }
        output << "Calculated pi = " << pi << std::endl;
    }
    else {
        double ratio;
        // Faire l'appelle avec nbp-1, seulement nbp-1 processus vont
realiser le calcul
        ratio = approximate_pi(nbp-1, nbSamples);
        output << "Calculated Ratio = " << ratio << std::endl;
        MPI_Send(&ratio, 1, MPI_DOUBLE, 0, 1234, globComm);
    }
}

```

Ce programme permet le calcul de pi par 1 ou plusieurs processus. Maintenant nous essayons de changer le programme pour que le processus maître (processus 0) soit capable de recevoir les résultats avec une méthode non bloquante. Pour cette implémentation, il est nécessaire de changer juste la partie qui le processus 0 reçoit la valeur des autres processus.

```

std::vector<MPI_Request> requests(nbp-1);
std::vector<MPI_Status> status(nbp-1);

std::vector<double> ratios(nbp-1);

```

```

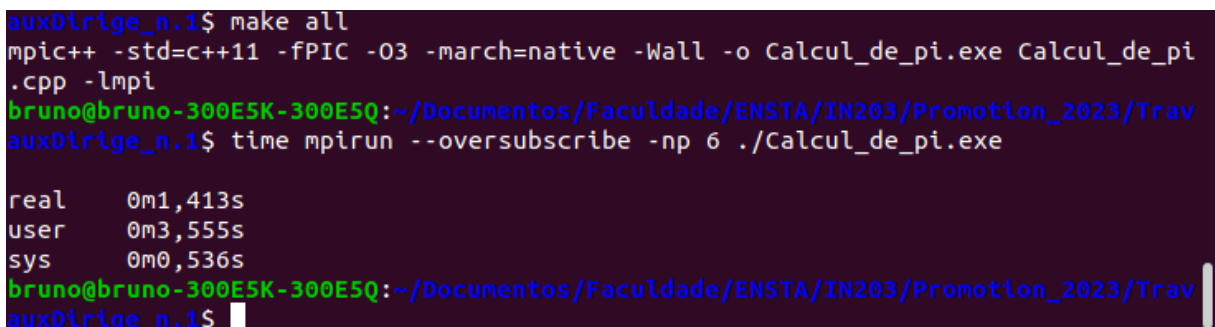
pi = 0;
for(int i = 1; i < nbp; i++) {
    MPI_Irecv(&ratios[i-1], 1, MPI_DOUBLE, i, MPI_ANY_TAG, globComm,
    &requests[i-1]);
}
MPI_Request *request_array = &requests[0];
MPI_Status *status_array = &status[0];

MPI_Waitall(nbp-1, request_array, status_array);
for(int i = 1; i < nbp; i++) {
    pi += ratios[i-1];
}

```

Dans cet exemple, le changement ne doit pas modifier beaucoup la performance en vue que le processus doit attendre la finalisation des autres de même façon.

Le programme a été testé et son temps d'exécution affiché, donc nous pouvons comparer les implémentations.



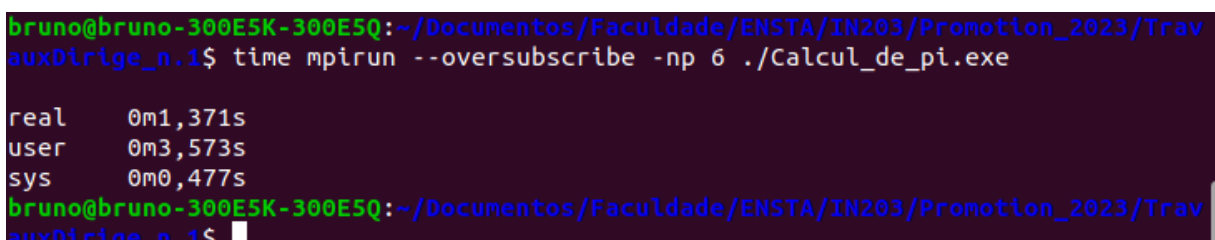
```

auxDirige_n.1$ make all
mpic++ -std=c++11 -fPIC -O3 -march=native -Wall -o Calcul_de_pi.exe Calcul_de_pi
.cpp -lmpi
bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Trav
auxDirige_n.1$ time mpirun --oversubscribe -np 6 ./Calcul_de_pi.exe

real    0m1,413s
user    0m3,555s
sys     0m0,536s
bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Trav
auxDirige_n.1$

```

Image 6: Temps d'exécution du programme avec les méthodes bloquantes.



```

bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Trav
auxDirige_n.1$ time mpirun --oversubscribe -np 6 ./Calcul_de_pi.exe

real    0m1,371s
user    0m3,573s
sys     0m0,477s
bruno@bruno-300E5K-300E5Q:~/Documentos/Faculdade/ENSTA/IN203/Promotion_2023/Trav
auxDirige_n.1$

```

Image 7: Temps d'exécution du programme avec la méthode non-bloquante.

En conclusion c'est possible d'apercevoir que pour la méthode non-bloquante le programme a un temps d'exécution moins important que celui avec la méthode bloquante.

## 5 - Diffusion d'un entier dans un réseau hypercube

Pour cet exercice nous avons été demandés de développer un algorithme en utilisant le MPI pour faire circuler un jeton dans un hypercube de dimension  $d$  en  $d$  étapes. L'algorithme développé est fourni ci-dessus, chaque processus, sauf le processus 0 attende que autre processus lui envoie le jeton, ça correspond à étape correspondant au numéro du processus, par exemple, le processus 3 faire recevra le jeton dans l'étape 2 e pour un hypercube de dimension plus grande ou égal à 3, il va envoyer le jeton a partir de l'étape 3.

```
int jeton = 89;
int dimension = (int) log2(nbp);
MPI_Status status;
if (rank != 0) {
    // Doit attendre que autre process realise la communication
    std::cout << rank << std::endl;
    MPI_Recv(&jeton, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, globComm,
    &status);
    output << rank << " <--- " << status.MPI_SOURCE << ": " << jeton <<
    std::endl;
}

for(int i = rank == 0 ? rank : (int) log2(rank) + 1; i < dimension; i++) {
    output << rank << " ---> " << rank + (int) pow(2, i) << ": " << jeton <<
    std::endl;
    MPI_Send(&jeton, 1, MPI_INT, rank + (int) pow(2, i), 1234, globComm);
}
```