

MPI

1 – Vous rentrerez le programme suivant afin de tester votre cluster :

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Vous vérifierez que la fonction `MPI_Get_processor_name` affiche bien les différents noms de votre cluster.

2 – Vous récupérerez les fichiers : Disponibles sur :

<https://github.com/wesleykendall/mpitutorial/tree/gh-pages/tutorials/mpi-send-and-receive/code>

Questions :

a. Pour le programme `send_recv`, **combien** de nœuds vont travailler ?

```
1 // Author: Wes Kendall
2 // MPI_Send, MPI_Recv example. Communicates the number -1 from process 0
3 // to process 1.
4 #include <mpi.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main(int argc, char** argv) {
9     // Initialize the MPI environment
10    MPI_Init(NULL, NULL);
11    // Find out rank, size
12    int world_rank;
13    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
14    int world_size;
15    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17    // We are assuming at least 2 processes for this task
18    if (world_size < 2) {
19        fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
20        MPI_Abort(MPI_COMM_WORLD, 1);
21    }
22
23    int number;
24    if (world_rank == 0) {
25        // If we are rank 0, set the number to -1 and send it to process 1
26        number = -1;
27        MPI_Send(
28            /* data          = */ &number,
```

```

29      /* count          = */ 1,
30      /* datatype       = */ MPI_INT,
31      /* destination    = */ 1,
32      /* tag            = */ 0,
33      /* communicator    = */ MPI_COMM_WORLD);
34  } else if (world_rank == 1) {
35      MPI_Recv(
36          /* data         = */ &number,
37          /* count        = */ 1,
38          /* datatype     = */ MPI_INT,
39          /* source       = */ 0,
40          /* tag          = */ 0,
41          /* communicator = */ MPI_COMM_WORLD,
42          /* status       = */ MPI_STATUS_IGNORE);
43      printf("Process 1 received number %d from process 0\n", number);
44  }
45  MPI_Finalize();
46 }

```

- ◊ Ligne 18, on vérifie qu'il y a au moins deux nœuds lancés ;
- ◊ Ligne 24 : on teste si on est le nœud de rang 0 et dans ce cas là on envoie la valeur -1 de la variable de number vers le nœud de rang 1 ;
- ◊ Ligne 34 : on teste si on est le nœud de rang 1, et si c'est le cas on reçoit la valeur envoyée par le nœud de rang 0.

⇒ seuls deux nœuds vont travailler.

- b. Dans le programme ping_pong, comment sont choisis les « partenaires » de ping-pong ?

Il y a seulement deux nœuds utilisés dans le cluster qui exécutent le **même code**, il est donc nécessaire de **casser la symétrie** en utilisant le rang unique du nœud dans le cluster : chaque nœud va déterminer en ligne 27 son « partner » : $0 \Rightarrow 1$ et $1 \Rightarrow 0$.

```

1 // Author: Wes Kendall
2 // Ping pong example with MPI_Send and MPI_Recv. Two processes ping pong a
3 // number back and forth, incrementing it until it reaches a given value.
4 //
5 #include <mpi.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8
9 int main(int argc, char** argv) {
10     const int PING_PONG_LIMIT = 10;
11
12     // Initialize the MPI environment
13     MPI_Init(NULL, NULL);
14     // Find out rank, size
15     int world_rank;
16     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
17     int world_size;
18     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
19
20     // We are assuming at least 2 processes for this task
21     if (world_size != 2) {
22         fprintf(stderr, "World size must be two for %s\n", argv[0]);
23         MPI_Abort(MPI_COMM_WORLD, 1);
24     }
25
26     int ping_pong_count = 0;
27     int partner_rank = (world_rank + 1) % 2;
28     while (ping_pong_count < PING_PONG_LIMIT) {
29         if (world_rank == ping_pong_count % 2) {
30             // Increment the ping pong count before you send it
31             ping_pong_count++;
32             MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
33             printf("%d sent and incremented ping_pong_count %d to %d\n",
34                 world_rank, ping_pong_count, partner_rank);
35         } else {
36             MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD,
37                 MPI_STATUS_IGNORE);
38             printf("%d received ping_pong_count %d from %d\n",
39                 world_rank, ping_pong_count, partner_rank);
40         }
41     }
42     MPI_Finalize();
43 }

```

Est-ce que le programme peut **bloquer** ?

Il y a alternance des rôles d'envoi et de réception de la valeur de la variable `ping_pong_count` qui n'est incrémenter que par l'émetteur mais mise à jour chez le récepteur lors de la réception. Une

borne max est définie pour cette variable qui quand elle est atteinte force la terminaison du nœud $\Rightarrow$ il ne peut y avoir de blocage.

Est-il possible d'ajouter une **étiquette** différente par ping-pong ?

Oui, on peut se servir de la valeur de la variable `ping_pong_count` comme étiquette, en oubliant pas de l'incrémenter lors de la réception :

```

28 while (ping_pong_count < PING_PONG_LIMIT) {
29     if (world_rank == ping_pong_count % 2) {
30         // Increment the ping pong count before you send it
31         ping_pong_count++;
32         MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, ping_pong_count, MPI_COMM_WORLD);
33         printf("%d sent and incremented ping_pong_count %d to %d\n",
34                world_rank, ping_pong_count, partner_rank);
35     } else {
36         ping_pong_count++;
37         MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, ping_pong_count,
38                 MPI_COMM_WORLD,
39                 MPI_STATUS_IGNORE);
40         printf("%d received ping_pong_count %d from %d\n",
41                world_rank, ping_pong_count, partner_rank);
42     }
43 }

```

c. Pour le programme ring, comment est créer « l'anneau » ?

L'anneau est défini par le rang du nœud dans le cluster :

- ◇ `rang-1` désigne le nœud situé à gauche du nœud courant ;
- ◇ `rang+1` désigne le nœud situé à droite du nœud courant.
- ◇ le nœud de rang 0 est lié au nœud de rang `world_size - 1` et vice-versa grâce à l'expression `(world_rank + 1) % world_size`;

```

1 // Author: Wes Kendall
2 // Example using MPI_Send and MPI_Recv to pass a message around in a ring.
3 //
4 #include <mpi.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main(int argc, char** argv) {
9     // Initialize the MPI environment
10    MPI_Init(NULL, NULL);
11    // Find out rank, size
12    int world_rank;
13    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
14    int world_size;
15    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
16
17    int token;
18    // Receive from the lower process and send to the higher process. Take care
19    // of the special case when you are the first process to prevent deadlock.
20    if (world_rank != 0) {
21        MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
22                MPI_STATUS_IGNORE);
23        printf("Process %d received token %d from process %d\n", world_rank, token,
24               world_rank - 1);
25    } else {
26        // Set the token's value if you are process 0
27        token = -1;
28    }
29    MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0,
30            MPI_COMM_WORLD);
31    // Now process 0 can receive from the last process. This makes sure that at
32    // least one MPI_Send is initialized before all MPI_Recvs (again, to prevent
33    // deadlock)
34    if (world_rank == 0) {
35        MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD,
36                MPI_STATUS_IGNORE);
37        printf("Process %d received token %d from process %d\n", world_rank, token,
38               world_size - 1);
39    }
40    MPI_Finalize();
41 }

```

Pourquoi doit-on modifier le **comportement** du nœud de rang 0 ?

Car c'est le nœud de rang 0 qui transmet initialement le token et le reçoit à la fin du nœud de rang le plus élevé.

Pouvez vous ajouter un « token » qui s'incrémente à chaque envoi d'un nœud à l'autre ?

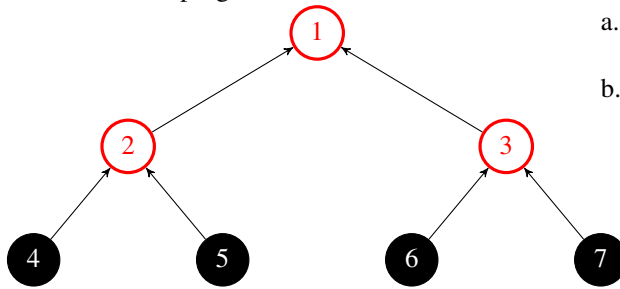
Oui, il suffit d'en incrémenter la valeur juste avant de l'émettre :

```
29 token++;  
30 MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0,  
31 MPI_COMM_WORLD);
```

Est-ce que le nœud 0 peut connaître le **nombre de nœuds** par la valeur du token qu'il reçoit au final ?
Par rapport à la modification précédente, le nœud 0 reçoit le rang de processus le plus élevé il lui suffit de l'incrémenter pour déterminer le nombre total de nœuds.

```
xterm  
$ mpirun -n 4 ./ring  
Process 1 received token 0 from process 0  
Process 0 received token 3 from process 3  
Process 2 received token 1 from process 1  
Process 3 received token 2 from process 2
```

3 – Vous écrirez un programme réalisant la **somme d'entiers en utilisant une structure d'arbre** :



- Comment chaque nœud va connaître le rang de son parent ?
- les nœuds 4,5,6 et 7 vont chacun envoyer un entier à leur parent respectif 2 et 3 qui va en faire la somme, avant de l'envoyer au nœud 1.

Écrivez un programme MPI réalisant ce travail en utilisant le rang du nœud feuille comme valeur entière à transmettre.

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#define PARENT(n) (ceil(n/2)-1)
#define ENFANTG(n) (n*2-1)
#define ENFANTD(n) (n*2)

int main (int argc, char *argv[])
{
    int rank, nprocs;
    int niveaux, numero;
    int premier = 0;
    int dernier = 0;
    long int resultat = 0;
    MPI_Status status;
    long int resultat_enfantd = 0;
    long int resultat_enfantg = 0;

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &nprocs);

    assert (!(nprocs+1-pow(2, log2(nprocs+1))));
    numero = rank+1;
    niveaux = log2(nprocs+1);
    premier = 1<<(niveaux-1);
    dernier = (1<<niveaux)-1;

    if (numero == 1)
        /* Racine de l'arbre */
        printf("Niveaux : %d\n",niveaux);
        printf("Numero: %d, dernier niveau de %d a %d\n", numero, premier, dernier);
        printf("Numero: %d reception de %d\n", numero, ENFANTD(numero)+1);
        printf("Numero: %d reception de %d\n", numero, ENFANTG(numero)+1);
        MPI_Recv (&resultat_enfantd, 1, MPI_LONG_LONG, ENFANTD(numero), 1, MPI_COMM_WORLD,
        &status);
        MPI_Recv (&resultat_enfantg, 1, MPI_LONG_LONG, ENFANTG(numero), 1, MPI_COMM_WORLD,
        &status);
        resultat = numero + resultat_enfantg + resultat_enfantd;
        printf("La somme de l'arbre vaut %ld\n", resultat);
        MPI_Finalize();
        return 0;
    }
    if ((numero >= premier) && (numero <= dernier))
        /* Noeud au dernier niveau */
        resultat = numero;
        printf("Numero: %d envoi de %ld\n", numero, resultat);
        MPI_Send(&resultat, 1, MPI_LONG_LONG, PARENT(numero), 1, MPI_COMM_WORLD);
        MPI_Finalize ();
        return 0;
    }
    /* Noeud de l'arbre avec des enfants */
    printf("Numero: %d reception de %d\n", numero, ENFANTD(numero)+1);
    printf("Numero: %d reception de %d\n", numero, ENFANTG(numero)+1);
    MPI_Recv (&resultat_enfantd, 1, MPI_LONG_LONG, ENFANTD(numero), 1, MPI_COMM_WORLD,
    &status);
    MPI_Recv (&resultat_enfantg, 1, MPI_LONG_LONG, ENFANTG(numero), 1, MPI_COMM_WORLD,
    &status);
    resultat = numero + resultat_enfantg + resultat_enfantd;
    printf("Numero: %d envoi de %ld\n", numero, resultat);
    MPI_Send (&resultat, 1, MPI_LONG_LONG, PARENT(numero), 1, MPI_COMM_WORLD);
    MPI_Finalize ();
    return 0;
}
  
```