UNIVERSITÉ D'ORLÉANS **Programmation Parallèle**
M1 Info Année **2024 - 2025**

Série de Travaux Dirigés : **1** *- L'API MPI et la parallélisation*

1 Architecture à mémoire distribuée : L'API MPI

Exercice 1. Lecture de codes

A partir des codes suivants expliquez ce que fait le programme. Vous pouvez donner un exemple en indiquant ce qu'il se passe pour chaque processus. On supposera qu'on a exécuté le programme avec `mpirun -np 4`.

1. La fonction `CommAComprendre`

```
1  void CommAComprendre(int n, int* in, int* out)
2  {
3    int pid, nprocs;
4    MPI_Comm_rank(MPI_COMM_WORLD, &pid );
5    MPI_Comm_size (MPI_COMM_WORLD, &nprocs );
6    int* tabtmp = new int[n/2];
7    int pid_envoi = (pid+1)%nprocs;
8    int pid_recu = (pid-1+nprocs)%nprocs;
9    if (pid%2==0){
10     MPI_Send(in+n/2,n/2,MPI_INT,pid_envoi,34,MPI_COMM_WORLD);
11     MPI_Recv(out,n/2,MPI_INT,pid_recu,34,MPI_COMM_WORLD,MPI_STATUS_IGNORE);}
12   else {
13     MPI_Recv(out,n/2,MPI_INT,pid_recu,34,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
14     MPI_Send(in+n/2,n/2,MPI_INT,pid_envoi,34,MPI_COMM_WORLD);}
15   pid_envoi = (pid-1+nprocs)%nprocs;
16   pid_recu = (pid+1)%nprocs;
17   if (pid%2==0){
18     MPI_Send(in,n/2,MPI_INT,pid_envoi,34,MPI_COMM_WORLD);
19     MPI_Recv(out+n/2,n/2,MPI_INT,pid_recu,34,MPI_COMM_WORLD,&status);  }
20   else {
21     MPI_Recv(out+n/2,n/2,MPI_INT,pid_recu,34,MPI_COMM_WORLD,&status);
22     MPI_Send(in,n/2,MPI_INT,pid_envoi,34,MPI_COMM_WORLD); }
23 }
24 int main ( int argc , char **argv )
25 {
26   int pid, nprocs;
27   MPI_Init (&argc , &argv) ;
28   MPI_Comm_rank(MPI_COMM_WORLD, &pid ) ;
29   MPI_Comm_size (MPI_COMM_WORLD, &nprocs ) ;
30   int nlocal = 2*atoi(argv[1]);
31   int* tablocal = new int[nlocal];
32   int* tabres = new int[nlocal];
33   srand(time(NULL)+pid);
34   for (int i=0; i<nlocal; i++)
35     tablocal[i] = rand()%10;
36   CommAComprendre(nlocal,tablocal,tabres);
37   delete[] tablocal;
38   delete[] tabres;
39   MPI_Finalize() ;
40   return 0 ; }
```

2. Fonction Mystère

```
1  void Mystere(int* ptr_a, int root) {
2      int pid, nprocs;
3      MPI_Comm_rank(MPI_COMM_WORLD, &pid);
4      MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
5      MPI_Status status;
6
7      int nb_etape = ceil((log((double) nprocs) / log(2.0)));
8      for (int i=0; i<nb_etape; i++) {
9          int size = pow(2,i);
10         if ((pid-root+nprocs)%nprocs<size) {
11             int pid_dest = (pid+size+nprocs)%nprocs;
12             if ((pid-root+nprocs)%nprocs+size<nprocs)
13                 MPI_Ssend(ptr_a, 1, MPI_INT, pid_dest, 1, MPI_COMM_WORLD);
14             }
15         else if ((pid-root+nprocs)%nprocs<pow(2,i+1)){
16             int pid_src = (pid - size + nprocs)%nprocs;
17             MPI_Recv(ptr_a,1,MPI_INT,pid_src,1,MPI_COMM_WORLD,&status);
18         }
19     }
20 }
21 int main(int argc, char **argv) {
22     int pid, nprocs;
23     MPI_Init(&argc, &argv);
24     MPI_Comm_rank(MPI_COMM_WORLD, &pid);
25     MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
26     int root = atoi(argv[1]); // le processeur root
27     int a;
28     if (pid==root) {
29         srand(time(NULL));
30         a = rand() % 10;
31     }
32     Mystere(&a,root);
33     MPI_Finalize();
34     return 0;
35 }
```

Exercice 2. Écriture de codes

1. Calcul du maximum d'un tableau d'entiers

   Complétez et modifiez le code ci-dessous afin que la fonction `calcul_max` renvoie le maximum d'un tableau d'entiers.

```
1  int fn_max(int n, int* tab) {
2      int max = tab[0];
3      for (int i=1; i<n; i++)
4          if (tab[i]>max)
5              max = tab[i];
6      return max;
7  }
8
9  int  calcul_max(int max_local, int root)
10 {
```

```
11      // ?????
12  }
13
14  int main ( int argc , char **argv )
15  {
16    int pid, nprocs;
17    MPI_Init (&argc , &argv) ;
18    MPI_Comm_rank(MPI_COMM_WORLD, &pid ) ;
19    MPI_Comm_size (MPI_COMM_WORLD, &nprocs ) ;
20
21    int n = atoi(argv[1]);
22    int root = atoi(argv[2]);
23
24
25    int* tab;
26    if (pid==root) {
27        tab = new int[n];
28        srand(time(NULL));
29        for (int i = 0; i < n; i++)
30            tab[i] = rand() % 1000;
31    }
32
33    // ??????
34
35
36    if (pid==root)
37        cout << "max final = " << max << endl;
38
39
40   if (pid==root)
41      delete[] tab;
42    MPI_Finalize() ;
43    return 0 ;
44  }
```

2. Calcul du maximum et de la position de ce maximum dans un tableau d'entiers

   Modifiez le code précédent pour l'adapter à la recherche du maximum d'un tableau et de sa position dans ce tableau. On pourra remplacer la fonction `calcul_max` par une fonction avec comme signature :

```
1  void calcul_max_et_position(int n, int* tab, int root, int* maxtab)
```

2 Architecture à mémoire distribuée : Parallélisation

Exercice 3. Une suite de Syracuse

Une suite de Syracuse est une suite telle que $U_0 = x$ avec $x > 0$ et

$$U_i = \begin{cases} \frac{U_{i-1}}{2} & si\ U_{i-1}\ est\ pair \\ 3U_{i-1}+1 & sinon \end{cases}$$

On souhaite vérifier si un tableau d'entiers U de taille n correspond à une suite de Syracuse alors que ce tableau est initialement sur le processeur `root`.

1. Décrivez le principe de la parallélisation en indiquant un ordre de grandeur de la complexité parallèle. Vous pouvez faire un schéma sur un exemple.

2. Est-il nécessaire d'avoir des communications au-delà de la distribution initiale de U.

3. Quelles sont les fonctions MPI que vous allez utiliser (en justifiant) ?

Exercice 4. Automate cellulaire : Le jeu de la vie

Le jeu de la vie est un automate cellulaire dont les règles ont été définies par J. Conway en 1970. Un automate cellulaire consiste en une grille de *cellules* pouvant chacune prendre à un instant donné un *état* parmi un ensemble fini. Le temps est également discret et l'état d'une cellule au temps n est fonction de l'état au temps $n-1$ d'un nombre fini de cellules appelé son *voisinage*. À chaque nouvelle unité de temps, les mêmes règles sont appliquées pour toutes les cellules de la grille, produisant une nouvelle *génération* de cellules dépendant entièrement de la génération précédente.

Dans le jeu de la vie, il n'y a que deux états : cellule occupée ou cellule vide. L'évolution de l'état d'une cellule dépend de l'état de ses huit plus proches voisins. Dans l'automate de Conway, les règles sont les suivantes :

- une cellule vide à l'étape $n-1$ et ayant exactement 3 voisins sera occupée à l'étape suivante (naissance liée à un environnement optimal) ;
- une cellule occupée à l'étape $n-1$ et ayant 2 ou 3 voisins sera maintenue à l'étape n sinon elle est vidée (destruction par désertification ou surpopulation).

C'est l'analogie entre ces règles et certains critères d'évolution de populations de bactéries qui a conduit à donner à cet automate le nom de jeu de la vie.

Dans sa version standard, l'automate se déroule sur un plan infini. Dans les représentations informatiques il est programmé sur un tore. La ligne supérieure est considérée comme jointive de la ligne inférieure, la colonne de gauche est jointive avec celle de droite.

Par exemple si on a l'état ci-dessous à l'étape en cours :

		x		
		x		
		x		

à l'étape suivante l'état sera :

	x	x	x	

On suppose que la grille de cellules est de très grande taille et nécessite d'être distribuée sur une architecture à mémoire distribuée pour que le calcul des itérations soit performant.

1. Quel est le principe de votre parallélisation ?

2. Comment faut-il distribuer les données ? Comment assurer l'équilibrage de charge ?

3. Que se passe-t-il entre deux itérations ? Est-il nécessaire d'échanger des informations ?

4. Quelles sont les fonctions MPI nécessaires à la mise en œuvre ?