Programme 1 : Satisfiabilité d'un circuit

```
/*  Programme qui permet de determiner si un circuit donne est satisfiable.  Le circuit est code
"a la dure" dans la fonction verifierCircuit  */

#include <stdio.h>
#include <mpi.h>  /*bibliotheque MPI*/

/*Retourne 1 si le ieme bit de n est 1, retourne 0 sinon*/
#define EXTRAIRE_BIT(n,i) ((n&(1<<i))?1:0)

void verifierCircuit(int id, int z) {
/*Verifie si la combinaison entree en parametre satisfait le circuit et l'affiche le cas echeant
id : Rang du processeur     z : Entier representant une combinaison sur 16 bits*/

    int v[16];  /*Chaque element est un bit de z*/
    int i;

    for (i = 0; i < 16; i++)
        v[i] = EXTRAIRE_BIT(z,i);     /*Transformation de l'entier en une chaine de 16 bits*/
    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3]) /*Verification du circuit*/
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {

        printf("Combinaison trouvee sur le processeur no. %d : %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n",
                id,v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8], v[9],v[10],v[11],v[12],v[13],v[14],v[15]);
        fflush(stdout);
    }
}

int main(int argc, char *argv[]) {
    int i;
    int id;  /*Rang du Processeur*/
    int p;   /*Nombre de processeurs*/

    MPI_Init(&argc, &argv);                 /*Initialisation de l'environnement MPI*/
    MPI_Comm_rank(MPI_COMM_WORLD, &id);     /*Affecte a id le rang du processeur (entre 0 et p-1)*/
    MPI_Comm_size(MPI_COMM_WORLD, &p);      /*Affecte a p le nombre de processeurs*/

    for (i = id; i < 65536; i += p)
        verifierCircuit(id,i);
    printf("Le processeur %d a termine son execution\n", id);
    fflush(stdout);
    MPI_Finalize();                         /*Libere les ressources utilisees par MPI*/
    return 0;
}
```

Programme 2 : Ajout de communications collectives (Réduction)

```
/*
Programme qui permet de determiner si un circuit donne est satisfiable.  Le circuit est code
"a la dure" dans la fonction verifierCircuit.
A la fin du programme, le processeur 0 imprime le nombre total de solutions
Pierre Delisle
*/

#include <stdio.h>
#include <mpi.h>  /*bibliotheque MPI*/

/*Retourne 1 si le ieme bit de n est 1, retourne 0 sinon*/
#define EXTRAIRE_BIT(n,i) ((n&(1<<i))?1:0)

int verifierCircuit(int id, int z) {
/*Verifie si la combinaison entree en parametre satisfait le circuit et l'affiche le cas echeant
Retourne 1 si la combinaison satisfait le circuit, 0 sinon
id : Rang du processeur
z : Entier representant une combinaison sur 16 bits*/

    int v[16];  /*Chaque element est un bit de z*/
    int i;
    int Trouve;

    for (i = 0; i < 16; i++)
        v[i] = EXTRAIRE_BIT(z,i);     /*Transformation de l'entier en une chaine de 16 bits*/

    Trouve = 0;
    if ((v[0] || v[1]) && (!v[1] || !v[3]) && (v[2] || v[3]) /*Verification du circuit*/
        && (!v[3] || !v[4]) && (v[4] || !v[5])
        && (v[5] || !v[6]) && (v[5] || v[6])
        && (v[6] || !v[15]) && (v[7] || !v[8])
        && (!v[7] || !v[13]) && (v[8] || v[9])
        && (v[8] || !v[9]) && (!v[9] || !v[10])
        && (v[9] || v[11]) && (v[10] || v[11])
        && (v[12] || v[13]) && (v[13] || !v[14])
        && (v[14] || v[15])) {

        printf("Combinaison trouvee sur le processeur no. %d : %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d\n",
                id,v[0],v[1],v[2],v[3],v[4],v[5],v[6],v[7],v[8], v[9],v[10],v[11],v[12],v[13],v[14],v[15]);
        fflush(stdout);
        Trouve = 1;
    }
    return Trouve;
}

int main(int argc, char *argv[]) {
    int i;
    int id;                 /*Rang du Processeur*/
    int p;                  /*Nombre de processeurs*/
    int nbSolutions;        /*Nombre de solutions trouvees par un processeur*/
    int nbTotalSolutions;   /*Nombre total de solutions trouvees par les processeurs*/

    MPI_Init(&argc, &argv);                     /*Initialisation de l'environnement MPI*/
    MPI_Comm_rank(MPI_COMM_WORLD, &id);         /*Affecte a id le rang du processeur (entre 0 et p-1)*/
    MPI_Comm_size(MPI_COMM_WORLD, &p);          /*Affecte a p le nombre de processeurs*/
```

```
    nbSolutions = 0;
    for (i = id; i < 65536; i += p)
        nbSolutions += verifierCircuit(id,i);

    printf("Le processeur %d a trouve %d solutions\n", id, nbSolutions);
    MPI_Reduce(&nbSolutions, &nbTotalSolutions, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    printf("Le processeur %d a termine son execution\n", id);
    fflush(stdout);
    MPI_Finalize();                         /*Libere les ressources utilisees par MPI*/
    if (id == 0)
        printf("Un total de %d combinaisons ont satisfait le circuit\n", nbTotalSolutions);
    return 0;

}
```

Programme 3 : Calculer le temps d'exécution

```
int main(int argc, char *argv[])
{
    int i;
    int id;                 /*Rang du Processeur*/
    int p;                  /*Nombre de processeurs*/
    int nbSolutions;        /*Nombre de solutions trouvees par un processeur*/
    int nbTotalSolutions;   /*Nombre total de solutions trouvees par les processeurs*/
    double tempsCPU;        /*Compteur de temps*/

    MPI_Init(&argc, &argv);         /*Initialisation de l'environnement MPI*/

    MPI_Barrier(MPI_COMM_WORLD);            /*Synchronise l'ensemble des processeurs*/
    tempsCPU = - MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &id);     /*Affecte a id le rang du processeur (entre 0 et p-1)*/
    MPI_Comm_size(MPI_COMM_WORLD, &p);      /*Affecte a p le nombre de processeurs*/

    nbSolutions = 0;
    for (i = id; i < 65536; i += p)
        nbSolutions += verifierCircuit(id,i);

    printf("Le processeur %d a trouve %d solutions\n", id, nbSolutions);
    MPI_Reduce(&nbSolutions, &nbTotalSolutions, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    tempsCPU += MPI_Wtime();

    printf("Le processeur %d a termine son execution\n", id);
    fflush(stdout);
    MPI_Finalize();                         /*Libere les ressources utilisees par MPI*/
    if (id == 0) {
        printf("Un total de %d combinaisons ont satisfait le circuit\n", nbTotalSolutions);
        printf("En %lf secondes\n", tempsCPU);
    }
    return 0;

}
```

Exercice 1 – Addition de n nombres

Écrivez un programme parallèle C/MPI qui calcule la somme 1 + 2 + ... + p de la façon suivante : chaque processeur i assigne la valeur i+1 à un entier, et ensuite les processeurs effectuent une réduction de ces valeurs. Le processeur maître (indice 0) affiche ensuite le résultat de la réduction. Pour vérifier que votre programme est correct, le processeur 0 calculera et affichera la valeur p(p+1) / 2.

Exercice 2 – Communications point à point

Écrivez un programme qui donne le même résultat que celui de l'exercice 1. Vous devez toutefois simuler une architecture en forme d'anneau et utiliser seulement les communications point à point MPI_Send et MPI_Recv.

Exercice 3 – Communications collectives (Diffusion)

Écrivez un programme où chacun des processeurs initialise un tableau de 5 réels générés aléatoirement entre 0 et 20. Ensuite, le processeur 0 tire au hasard une valeur *x* entre 0 et 20. Vous devez déterminer, parmi l'ensemble des valeurs générées initialement par les processeurs, le nombre de valeurs supérieures à *x*. Vous devez aussi déterminer le nombre maximal de valeurs supérieurs à *x* présentes sur un seul processeur. Le programme doit afficher les valeurs générées par chaque processeur, la valeur de *x* tirée, le nombre total de valeurs supérieures à *x*, ainsi que le nombre maximal de valeurs supérieures à *x* trouvées sur un seul processeur (pas besoin de savoir quel est le processeur en question).

Annexe – Fonctions MPI vues durant le cours

MPI_Init (&argc, &argv)

MPI_Finalize ()

MPI_Comm_rank (MPI_comm comm, int *rank)

MPI_Reduce (void *operand, void *result, int count, MPI_Datatype type, MPI_Op operator, int root, MPI_Comm comm)

MPI_Barrier (MPI_Comm comm)

MPI_Send (void *message, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

MPI_Recv (void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *Status)

MPI_Bcast (void *buffer, int count, MPI_Data_Type datatype, int root, MPI_Comm comm)