Code CF Bakhouya lkmala dial dakchi li kan 3ndna ch7al hadi ahahahaha

Master Slave.c

#include <stdlib.h>

#include <stdio.h>

#include "mpi.h"

int main(int  argc,char\*\* argv)

{

     int myrank, nprocs, n, islave, master, t[10], s = 0;

     double t0,t1;

     MPI\_Status   status;

     int          ierr, resultlen;

     char         hostname[MPI\_MAX\_PROCESSOR\_NAME];

     MPI\_Init(& argc, & argv);

     MPI\_Comm\_rank(MPI\_COMM\_WORLD, & myrank);

     MPI\_Comm\_size(MPI\_COMM\_WORLD, & nprocs);

     MPI\_Get\_processor\_name(hostname, & resultlen);

     t0 = MPI\_Wtime();

     MPI\_Barrier (MPI\_COMM\_WORLD);

     t1 = MPI\_Wtime();

    // partie maitre: lit un entier sur l'entrée standard

    if ( myrank == 0 )

    {

        for(int i = 0; i < 10; i++)

        {

            t[i] = i + 1;

        }

        for(islave=1 ; islave < nprocs ; islave++)

        {

            MPI\_Send(& t, 10, MPI\_INT, islave, 10, MPI\_COMM\_WORLD);

        }

        for(islave=1 ; islave < nprocs ; islave++)

        {

            MPI\_Recv (& s, 1, MPI\_INT, islave, 10, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE );

        }

        printf("SOmme : %d\n", s);

    }

    // partie esclave: recoit le message avec l'entier

    else

    {

        master = 0;

        ierr = MPI\_Recv (& t, 10, MPI\_INT, master, 10, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE );

        for(int i = 0; i < 10; i++)

        {

            s += t[i];

        }

        MPI\_Send(& s, 1, MPI\_INT, master, 10, MPI\_COMM\_WORLD);

    }

    MPI\_Finalize();

}

Exercice5.c (Page 80)

#include <stdlib.h>

#include <stdio.h>

#include "mpi.h"

int main(int  argc,char\*\* argv)

{

     int myrank, nprocs, n, islave, master, x;

     double t0,t1;

     MPI\_Status   status;

     int          ierr, resultlen;

     char         hostname[MPI\_MAX\_PROCESSOR\_NAME];

     MPI\_Init(& argc, & argv);

     MPI\_Comm\_rank(MPI\_COMM\_WORLD, & myrank);

     MPI\_Comm\_size(MPI\_COMM\_WORLD, & nprocs);

     MPI\_Get\_processor\_name(hostname, & resultlen);

     t0 = MPI\_Wtime();

     MPI\_Barrier (MPI\_COMM\_WORLD);

     t1 = MPI\_Wtime();

    // Master Scan

    if ( myrank == 0 )

    {

        printf("Input : ");

        scanf("%d",&x);

        MPI\_Send(& x, 1, MPI\_INT, master, 10, MPI\_COMM\_WORLD);

        for(islave=1 ; islave < nprocs ; islave++)

        {

            MPI\_Send(& x, 1, MPI\_INT, islave, 10, MPI\_COMM\_WORLD);

        }

        MPI\_Recv (& x, 1, MPI\_INT, master, 10, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE );

        printf("Got : %d\n ", x);

    }

    else

    {

        master = 0;

        ierr = MPI\_Recv (& x, 1, MPI\_INT, master, 10, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE );

        printf("Got : %d\n ", x);

    }

    MPI\_Finalize();

}

Exercice 6.c (Page 80 – 81)

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

int main(int argc, char \* argv[])

{

    int wsize;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &wsize);

    // The number of nodes is the number of proccess

    int nnodes = wsize;

    int index[5] = { 1, 4, 6, 7, 9 };

    int edges[] = { 1, 0, 2, 3, 1, 3, 4, 3, 2 };

    // Create a new graph comm

    MPI\_Comm graph\_comm;

    MPI\_Graph\_create(MPI\_COMM\_WORLD, wsize, index, edges, 1, &graph\_comm);

    // Some testing

    int i, realRank, neighborsCount;

    // #1 Number of neighbors per node

    MPI\_Comm\_rank(graph\_comm, &realRank);

    MPI\_Graph\_neighbors\_count(graph\_comm, realRank, &neighborsCount);

    printf("Rank %d: Neighbors count = %d\n", realRank, neighborsCount);

    // #2 Neighbors of each node

    int neighbors[5];

    MPI\_Graph\_neighbors(graph\_comm, realRank, 5, neighbors);

    for (i = 0; i < neighborsCount; i++)

    {

        printf("Rank %d: Neighbor %d\n", realRank, neighbors[i]);

    }

    MPI\_Finalize();

    return 0;

}

Arbre Hauteur Minimale ( Ex8-1: Identification des processus à l’aide d’un parcours en largeur P81)

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

// Fonction pour supprimer un élément d'un tableau

void removeElementFromArray(int array[], int \*length, int element)

{

    int i, pos = -1;

    for (i = 0; i < \*length; i++)

    {

        if (array[i] == element)

        {

            pos = i;

            break;

        }

    }

    // Si l'element a été retrouvé

    if (pos != -1)

    {

        for (i = pos; i < \*length - 1; i++)

        {

            array[i] = array[i + 1];

        }

        (\*length)--;

    }

}

/\*

    Messages utilisés (chacun son id):

        - 0: Traverse()

        - 1: Retour()

\*/

void sendMessage(int id, int dest, int count, MPI\_Comm comm)

{

    int buffer[2] = {id, count};

    MPI\_Send(buffer, 2, MPI\_INT, dest, 0, comm);

}

int main(int argc, char \*argv[])

{

    int wsize, pi;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &wsize);

    // Création du graphe

    int index[6] = {3, 4, 7, 11, 13, 16};

    int edges[16] = {1, 2, 3, 0, 0, 3, 5, 0, 2, 4, 5, 3, 5, 2, 3, 4};

    MPI\_Comm graph\_comm;

    MPI\_Graph\_create(MPI\_COMM\_WORLD, wsize, index, edges, 1, &graph\_comm);

    MPI\_Comm\_rank(graph\_comm, &pi);

    // Variables locales

    int message;

    int id;

    int suivant;

    int tailles[6];

    int countfils = 0;

    int fils[6];

    int rcv[2];

    int count;

    int maxNeighbors = 4;

    int neighborsCount;

    int neighbors[maxNeighbors]; // Liste des voisins

    MPI\_Graph\_neighbors\_count(graph\_comm, pi, &neighborsCount);

    MPI\_Graph\_neighbors(graph\_comm, pi, maxNeighbors, neighbors);

    int i;

    int visited = 0; // Si ce noeud a était visité ou non

    int pred = -1;   // Le prédécesseur de ce noeud

    MPI\_Status status;

    // A la récéption de INIT() par le processus 0

    if (pi == 0)

    {

    count = 1;

        visited = 1;

        printf("Node 0: INIT.\n");

        for (i = 0; i < neighborsCount; i++)

        {

            sendMessage(0, neighbors[i], 0, graph\_comm);

        }

    }

    while (1)

    {

        MPI\_Recv(&rcv, 2, MPI\_INT, MPI\_ANY\_SOURCE, 0, graph\_comm, &status);

        int pj = status.MPI\_SOURCE;

    message = rcv[0];

    //printf("message %d \n", message);

    count += rcv[1];

        // A la reception de Traverse() par pi depuis pj

        if (message == 0)

        {

            removeElementFromArray(neighbors, &neighborsCount, pj);

            if (!visited)

            {

                visited = 1;

                pred = pj;

                printf("Node %d: My predecessor is %d\n", pi, pred);

                if (neighborsCount == 0)

                {

                    sendMessage(1, pred, 1, graph\_comm);

                    break;

                }

                else

                {

                    for (i = 0; i

                     < neighborsCount; i++)

                    {

                        sendMessage(0, neighbors[i], 0, graph\_comm);

                    }

                }

            }

            else

            {

                sendMessage(1, pj, 0, graph\_comm);

            }

        }

        // A la reception de Retour() par pi depuis pj

        if (message == 1)

        {

            removeElementFromArray(neighbors, &neighborsCount, pj);

            if (rcv[1] > 0){

                tailles[pj] = rcv[1];

                fils[countfils] = pj;

                countfils++;

            }

            if (neighborsCount == 0)

            {

                if (pred == -1)

                {

                    printf("Node %d: END, node count: %d.\n", pi, count);

                    id = 0;

                    suivant = id + 1;

                    for (int k = 0; k < countfils; k++){

                        sendMessage(2, fils[k], suivant, graph\_comm);

                        suivant = suivant + tailles[fils[k]];

                    }

                }

                else

                {

                    sendMessage(1, pred, count + 1, graph\_comm);

                }

                break;

            }

        }

    }

    while (1){

        MPI\_Recv(&rcv, 2, MPI\_INT, MPI\_ANY\_SOURCE, 0, graph\_comm, &status);

        int pj = status.MPI\_SOURCE;

    message = rcv[0];

    //printf("message %d \n", message);

    count += rcv[1];

        // A la reception de Traverse() par pi depuis pj

        if (message == 2){

            //printf("AA");

            id = rcv[1];

            if (countfils > 0){

                suivant = rcv[1] + 1;

                for (int k = 0; k < countfils; k++){

                    sendMessage(2, fils[k], suivant, graph\_comm);

                            suivant = suivant + tailles[fils[k]];

                }

            }else{

                sendMessage(3, pred, 0, graph\_comm);

                printf("\nnode %d id: %d\n", pi, id);

                break;

            }

        }

        if (message == 3){

            removeElementFromArray(fils, &countfils, pj);

            if (countfils == 0){

                if (pred == -1){

                    printf("\nnode %d id: %d\n", pi, id);

                    printf("FIN\n");

                }else{

                    sendMessage(3, pred, 0, graph\_comm);

                    printf("\nnode %d id: %d\n", pi, id);

                }

                break;

            }

        }

    }

    MPI\_Finalize();

    return 0;

}

Arbre Hauteur Minimale (Ex 8 Part 2 Page 81)

// Propagation contrôlée

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

/\*

    Messages utilisés (chacun son id):

        - 0: Traverse()

        - 1: Retour()

\*/

void sendMessage(int id,int data, int dest, MPI\_Comm comm)

{

   int buffer[2];

    buffer[0] = id;

    buffer[1] = data;

    MPI\_Send(buffer,2, MPI\_INT, dest, 0, comm);

}

int main(int argc, char \*argv[])

{

    int wsize, pi;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &wsize);

    // Création du graphe

    int index[6] = { 2,5,8,12,14,16 };

    int edges[] = {1,2,0,2,3,0,1,3,1,2,4,5,3,5,3,4};

    MPI\_Comm graph\_comm;

    MPI\_Graph\_create(MPI\_COMM\_WORLD, wsize, index, edges, 1, &graph\_comm);

    MPI\_Comm\_rank(graph\_comm, &pi);

    // Variables locales

    int buffer[2];

    int message;

    int maxNeighbors = 5;

    int neighborsCount;

    int neighbors[maxNeighbors]; // Liste des voisins

    MPI\_Graph\_neighbors\_count(graph\_comm, pi, &neighborsCount);

    MPI\_Graph\_neighbors(graph\_comm, pi, maxNeighbors, neighbors);

    int c = 1;

    int i;

    int visited = 0; // Si ce noeud a était visité ou non

    int pred = -1;   // Le prédécesseur de ce noeud

    int h = 90;

    int retours=0;

    MPI\_Status status;

    // A la récéption de INIT() par le processus 0

    if (pi == 0)

    {

        h = 0;

        printf("Node 0: INIT.\n");

        for (i = 0; i < neighborsCount; i++)

        {

            sendMessage(0 , h, neighbors[i], graph\_comm);

            retours++;

        }

    }

    while (1)

    {

        MPI\_Recv(buffer, 2, MPI\_INT, MPI\_ANY\_SOURCE, 0, graph\_comm, &status);

        int pj = status.MPI\_SOURCE;

        // A la reception de Traverse() par pi depuis pj

        if (buffer[0] == 0)

        {

            if ( h> buffer[1]+1)

            {

                if (retours > 0)

                {

                    sendMessage(1 , -1,pred, graph\_comm);

                }

                // pour afficher l'ancienne hauteur

                       printf("l'ancienne hauteur du neoud numero %d est= %d \n", pi , h);

              h = buffer[1]+1;

             // pour afficher la nouvelle hauteur

                      printf("la nouvelle hauteur du neoud numero %d est= %d \n", pi , h);

              pred = pj;

                    for (i = 0; i < neighborsCount; i++)

                    {

                    if (neighbors[i] == pj){

                     continue;

                     }

                        sendMessage(0, h, neighbors[i], graph\_comm);

                        retours++;

                    }

            }

            else

            {

                sendMessage(1,-1 ,pj, graph\_comm);

            }

        }

        // A la reception de Retour() par pi depuis pj

        if (buffer[0] == 1)

        {

        retours--;

           if (retours == 0){

               if (pred == -1){

               printf("\n fin");

               }

               else {

               printf("Node %d:  predecessor is %d hauteur: %d\n ", pi, pred,h);

               sendMessage(1,-1 ,pred, graph\_comm);

               }

                break;

            }

        }

    }

    MPI\_Finalize();

    return 0; }

Algorithme Hirsch-Berg et Sinclair : Election de processus

// Election de processus dans un anneau bidirectionnel

// Critere de l'election : un nombre r aleatoire

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

typedef enum { NON\_CONCERNE,

               CANDIDAT,

               BATTU,

               ELU } etats;

/\*

    Messages utilisésèèèèèèèèèèèèèèèèèèèèè (kulla 3ndo l id dialo):

        - 0: Candidature(r, lg, lgmax, origin) // origine : contient le p\_id du processus envoyant la candidature

                                                                                        pour controler la fin de l'algo

        - 1: Reponse(bool, pi)

        - 2: Terminer(pi)

\*/

void sendMessage(int id, int value1, int value2, int value3,int value4, int dest, MPI\_Comm comm)

{

    int buffer[5];

    buffer[0] = id;

    buffer[1] = value1;

    buffer[2] = value2;

    buffer[3] = value3;

    buffer[4] = value4;

    MPI\_Send(buffer, 5, MPI\_INT, dest, 0, comm);

}

void faire\_suivre(int buffer[5], int pj, int neighbors[2], MPI\_Comm comm)

{

    if (pj == neighbors[0])

    {

        MPI\_Send(buffer, 5, MPI\_INT, neighbors[1], 0, comm);

    }

    else if (pj == neighbors[1])

    {

        MPI\_Send(buffer, 5, MPI\_INT, neighbors[0], 0, comm);

    }

}

int main(int argc, char \*argv[])

{

    int wsize, pi;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &wsize);

    // Création du graphe

    int index[10] = {2, 4, 6, 8, 10,12,14,16,18,20};

    int edges[] = {1,7,0,4,4,5,8,9,1,2,2,6,5,9,0,8,3,7,3,6};

    MPI\_Comm graph\_comm;

    MPI\_Graph\_create(MPI\_COMM\_WORLD, wsize, index, edges, 1, &graph\_comm);

    MPI\_Comm\_rank(graph\_comm, &pi);

    srand(time(0)+pi);

    // Variables locales

    int mon\_numero = pi;

    etats etat = NON\_CONCERNE;

    // L'état du processus ∈ { NON\_CONCERNE, CANDIDAT, BATTU, ELU } initialisé à NON\_CONCERNE

    int lgmax = 0;             // Entier représentant lenth dial chemin d chi iteration m3tia

    int vainqueur = -1;

    int nbrRep = 0;             // Entier entre 0 et 2

    int repOK = 0;             // Booléen initialisé à Vrai

    int r = 0;            // Entier random (pour faire l'election a base de cet entier )

    int i, buffer[5];

    MPI\_Status status;

    int neighbors[2]; // jiran

    MPI\_Graph\_neighbors(graph\_comm, pi, 2, neighbors);

    // Tous les processus provoquent une élection en même temps

    etat = CANDIDAT;

    lgmax = 1;

    nbrRep = 0;

    repOK = 1;

    r = rand();

    sendMessage(0, r, 0, lgmax, pi, neighbors[0], graph\_comm);

    sendMessage(0, r, 0, lgmax, pi, neighbors[1], graph\_comm);

    while (1)

    {

        MPI\_Recv(buffer, 5, MPI\_INT, MPI\_ANY\_SOURCE, 0, graph\_comm, &status);

        int pj = status.MPI\_SOURCE;

        // A la reception de Candidature(num, lg, lgmax) par pi depuis pj

        if (buffer[0] == 0)

        {

            int num = buffer[1];

            int lg = buffer[2];

            int lgmax = buffer[3];

            int origine = buffer[4]; // pid of process who started la condidature

            if (num < r)

            {

                sendMessage(1, 0, pj, -1,-1, pj, graph\_comm);

                if (etat == NON\_CONCERNE)

                {

                    // Provoke election

                    etat = CANDIDAT;

                    lgmax = 1;

                    nbrRep = 0;

                    repOK = 1;

                    sendMessage(0, r, 0, lgmax, origine, neighbors[0], graph\_comm);

                    sendMessage(0, r, 0, lgmax, origine, neighbors[1], graph\_comm);

                }

            }

            else if (num > r)

            {

                etat = BATTU;

                lg++;

                if (lg < lgmax)

                {

                    faire\_suivre(buffer, pj, neighbors, graph\_comm);

                }

                else

                {

                    sendMessage(1, 1, pj, -1,-1, pj, graph\_comm);

                }

            }

            // if r = num

            else

                        {

                                if(pi==origine) //if: pi = origine => l message taydir dora 3la l anneau

                                {

                                        if (etat != ELU)

                                        {

                                                printf("Processus %d: Je suis le vainqueur r=%d.\n", mon\_numero,r);

                                                etat = ELU;

                                                vainqueur = pi;

                                                buffer[0] = 2; // end

                                                buffer[1] = vainqueur;

                                                faire\_suivre(buffer, pj, neighbors, graph\_comm);

                                                break;

                                        }

                                }

                                else // (pi != origine) dans le cas ou' deux processus ont le meme nombre r

                        {

                                // on donne la main au process dont l'id est petit

                                if (pi < origine)

                                {

                                     sendMessage(1, 0, pj, -1,-1, pj, graph\_comm);

                                            if (etat == NON\_CONCERNE)

                                            {

                                                // Provoquer election

                                                etat = CANDIDAT;

                                                lgmax = 1;

                                                nbrRep = 0;

                                                repOK = 1;

                                                sendMessage(0, r, 0, lgmax, origine, neighbors[0], graph\_comm);

                                                sendMessage(0, r, 0, lgmax, origine, neighbors[1], graph\_comm);

                                            }

}

else

{

 etat = BATTU;

                                              lg++;

                                              if (lg < lgmax)

                                              {

                                                  faire\_suivre(buffer, pj, neighbors, graph\_comm);

                                              }

                                              else

                                              {

                                                  sendMessage(1, 1, pj, -1,-1, pj, graph\_comm);

                                              }

 }

 }

 }

 }

        if (buffer[0] == 1)

        {

            int bool = buffer[1];

            int id = buffer[2];

            if (id == pi)

            {

                nbrRep++;

                repOK = repOK && bool;

                // Provoke election (tsuzuku)

                if (nbrRep == 2)

                {

                    if (repOK == 0)

                    {

                        etat == BATTU;

                    }

                    lgmax = 2 \* lgmax;

                    if (etat == CANDIDAT)

                    {

                        nbrRep = 0;

                        repOK = 1;

                        sendMessage(0, r, 0, lgmax, pi, neighbors[0], graph\_comm);

                        sendMessage(0, r, 0, lgmax, pi, neighbors[1], graph\_comm);

                    }

                }

            }

            else

            {

                faire\_suivre(buffer, pj, neighbors, graph\_comm);

            }

        }

        // A la reception de Terminer(num) par pi depuis pj

        if (buffer[0] == 2)

        {

            int id = buffer[1];

            if (vainqueur != id)

            {

                faire\_suivre(buffer, pj, neighbors, graph\_comm);

                vainqueur = id;

                etat = NON\_CONCERNE;

                printf("Processus %d r=%d: Le vainqueur est %d.\n", mon\_numero,r, vainqueur);

                break;

            }

        }

    }

    MPI\_Finalize();

    return 0;

}

Naimi trehel :

// Algorithme de Naimi et Tréhel

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <unistd.h>

#include <pthread.h>

typedef struct

{

    int pi;

    int \*demandeur;

    int \*privilege;

    int \*suivant;

    MPI\_Comm graph\_comm;

} SC\_Infos;

/\*

    Messages utilisés (chacun son id):

        - 0: REQUETE(j)

        - 1: JETON()

\*/

void sendMessage(int id, int data, int dest, MPI\_Comm comm)

{

    int buffer[2];

    buffer[0] = id;

    buffer[1] = data;

    MPI\_Send(buffer, 2, MPI\_INT, dest, 0, comm);

}

void \*section\_critique(void \*v\_infos)

{

    SC\_Infos \*infos = (SC\_Infos \*)v\_infos;

    printf("Processus %d: J'entre dans la section critique.\n", infos->pi);

    // On sleep 5 secondes pour simuler du travail

    sleep(10);

    printf("Processus %d: Je sors de la section critique.\n", infos->pi);

    \*(infos->demandeur) = 0;

    if (\*(infos->suivant) != -1)

    {

        printf("Processus %d: Je passe le jeton à %d.\n", infos->pi, \*(infos->suivant));

        \*(infos->privilege) = 0;

        sendMessage(1, -1, \*(infos->suivant), infos->graph\_comm);

        \*(infos->suivant) = -1;

    }

    return NULL;

}

void entrer\_dans\_la\_section\_critique(int pi, int \*demandeur, int \*privilege, int \*suivant, MPI\_Comm graph\_comm)

{

    SC\_Infos \*infos = (SC\_Infos \*)malloc(sizeof(SC\_Infos));

    infos->pi = pi;

    infos->demandeur = demandeur;

    infos->privilege = privilege;

    infos->suivant = suivant;

    infos->graph\_comm = graph\_comm;

    // Création d'un thread pour exécuter la section critique

    pthread\_t thread;

    pthread\_create(&thread, NULL, section\_critique, &(\*infos));

}

int main(int argc, char \*argv[])

{

    int wsize, pi;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &wsize);

    // Création du graphe

    int index[5] = {4, 8, 12, 16, 20};

    int edges[] = {1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0, 1, 2, 4, 0, 1, 2, 3};

    MPI\_Comm graph\_comm;

    MPI\_Graph\_create(MPI\_COMM\_WORLD, wsize, index, edges, 1, &graph\_comm);

    MPI\_Comm\_rank(graph\_comm, &pi);

    // Variables locales

    int privilege = (pi == 0);        // Booléen init à FAUX sauf pour la première racine. Il vaut VRAI si Pi possède le jeton.

    int dernier = (pi == 0 ? -1 : 0); // Pointeur sur le dernier demandeur connu par Pi.

    int suivant = -1;                 // Indique le successeur de Pi pour entrer dans la <SC>

    int demandeur = 0;                // Booléen qui vaut VRAI si le site est demandeur de la <SC>

    int i, buffer[2];

    MPI\_Status status;

    int neighborsCount;

    MPI\_Graph\_neighbors\_count(graph\_comm, pi, &neighborsCount);

    int neighbors[neighborsCount]; // Liste des voisins

    MPI\_Graph\_neighbors(graph\_comm, pi, neighborsCount, neighbors);

    // Le processus 0 demande l'entrée en ressource critique en premier

    // Le processus 3 demande l'entrée en ressource critique après 10 secondes

    if (pi == 0 || pi == 3)

    {

        if (pi == 3)

            sleep(2);

        demandeur = 1;

        if (dernier != -1)

        {

            printf("Processus %d: Je demande le jeton.\n", pi);

            sendMessage(0, pi, dernier, graph\_comm);

            dernier = -1;

        }

        else if (privilege) // Cas où la racine demande l'accès et a déjà le jeton (comme le processus 0 au début)

        {

            entrer\_dans\_la\_section\_critique(pi, &demandeur, &privilege, &suivant, graph\_comm);

        }

    }

    while (1)

    {

        MPI\_Recv(buffer, 2, MPI\_INT, MPI\_ANY\_SOURCE, 0, graph\_comm, &status);

        int pj = status.MPI\_SOURCE;

        // A la reception de REQUETE(j) par pi depuis pj

        if (buffer[0] == 0)

        {

            int j = buffer[1];

            if (dernier == -1)

            {

                if (demandeur == 1)

                {

                    suivant = j;

                    printf("Processus %d: Mon suivant est %d.\n", pi, suivant);

                }

                else

                {

                    printf("Processus %d: Je relâche le jeton à %d.\n", pi, j);

                    sendMessage(1, -1, j, graph\_comm);

                    privilege = 0;

                }

            }

            else

            {

                printf("Processus %d: Je passe la demande du jeton depuis %d à mon dernier %d.\n", pi, j, dernier);

                sendMessage(0, j, dernier, graph\_comm);

            }

            dernier = j;

        }

        // A la reception de JETON() par pi depuis pj

        if (buffer[0] == 1)

        {

            printf("Processus %d: Jeton reçu.\n", pi);

            privilege = 1;

            entrer\_dans\_la\_section\_critique(pi, &demandeur, &privilege, &suivant, graph\_comm);

        }

    }

    MPI\_Finalize();

    return 0;

}