Hategekimana Fabrice

Parallelism

Midterm exam

Je déclare sur l'honneur que je réponds aux questions de ce contrôle sans l'aide de mes camarades ni d'aucune autre personne que moi-même

Date: 11/11/20

Nom, Prénom: Hategekimana Fabrice

Signature: Fabrice

Hategekimana Fabrice

Exercice1

```
...
int part = n/nproc;
vector<int> localvector();
int total;
if(myRank == 0){
  vector<int>  initVector(int n)
}
  MPI_Scatter(initvector, data size), part, MPI_INT, localvector, data(), part, MPI_INT, 0)
int count = 0;
for(auto v: local vector){
    if(isprime(v)){
        count++;
    }
}

MPI_Reduce(&count, &total, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if(my Rank == 0){

    Std:: cout << "proportion :" << total << "/" << n << std::endl;

}
```

MPI_COMM_World

Hutegekimana Fabrice

Exercice 2

Extrait de code 1

Problème: - c'est un code qui donne un deadlock.
- La ligne 4 utilise une méthode blocante Ssend (safe send)
- ce n'est pas un problème pour le premier processus qui envoit
- c'est un problème pour le processus qui doit attendre un résultat

Solution: - on peut utiliser un Bsend pour éviter de rendre l'exécution blocante (à la place du Ssend)

Extrait de code 2

Problème: - Ligne 4 IRecv est un processus non-blocant et Ligne 5 utilise une valeur qu'on est pas sûr d'avoir reçus
- peut falsifier les résultats

Solution: - je propose d'utiliser la requête retourné par IRecv et de faire un .wait() avant d'exécuter la ligne 5

Extrait de code 3

Problème: - Deadlock car il risque d'avoir une incompatibilité sur les canneaux de transfert.
- Bcast est comme une barrière qui synchronise l'envoit de tout les processus

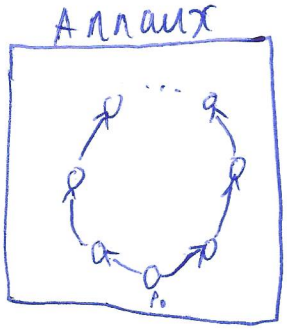
Solution: - éviter d'utiliser MPI_Recv et utiliser Bcast pour tout le monde.

Hate de Kimana Fabrice

Exercice 3

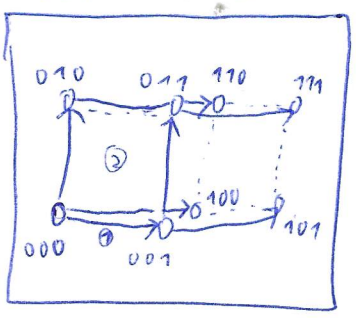
- Cette implémentation marche mais est peu recommandée car elle est lente du fait que le partage soit séquentiel et non parallèle.
- une solution serait d'utiliser MPI_Bcast qui est une primitive assez rapide.

- une autre solution serait d'utiliser un transfère en anneaux (chaque processus reçois de son voisin et envoit à son autre voisin)




```
if( Rank = 0 ){
    send voisingauche
    send voisin droite
else {
    if (à droite){
        reçoit voisingauche
        send voisin droite
    else{
        reçoit voisin droite
        send voisin gauche
```

- une autre solution encore serait d'utiliser la diffusion en hypercube (chaque processus envoit à son prochain voisin d'une autre dimension)




```
for(i=0, i·2^nProc < nProc, i = i·2^2)
    if (myRank & i = 0){
        reçoit( )
    }
    send (voisin, i)
}
```

Hategekimana Fabrice

Exercice 4

Extrait de code 1

Problème : - concurrence sur la variable globale partagée sum
- des valeurs peuvent se perdre

solution : utiliser un mutex pour créer une section critique avec : mutex mut; mut.lock() et mut.unlock() et faire une somme locale avant de toucher à sum :

```
void sumvec(int low, int up){
    int localsum = 0;
    for(int i = low; i < up; i++){
        localsum += vect[i];
    }
    mut.lock();
    sum += localsum;
    mut.unlock();
}
```

Extrait de code 2

Problème : - le thread principal risque de se terminer avant t1 et t2.
- t1 et t2 ne pourront peut-être pas afficher leur résultats

solution : - utiliser t1.join() et t2.join() pour obliger le thread principal à les attendre (il faut les placer juste avant return 0)

Extrait de code 3

~~problème~~ : ~~...~~

- Il y a en tout 9 threads qui executent la function :
- Le n=8 thread plus le thread principal.

Hategekimana Fabrice

Exercice 5

```
Array2D M, V, res;

mutex mut

void mvprod(int low, int up){
    for(int i=low; i<up; i++){
        int local res;
        local res = scalar product(M.getRow(i), V);
        mut.lock();
        res.at(i) = local res;
        mut.unlock();
    }
}

int main(int argc, char ** argv){
    int m= atoi(argv[1]);
    int n = atoi(argv[2]);
    int numberOfThreads = atoi(argv[3]);

    M = init Array(m,n);
    V = init Array(n,1);
    res = init Array(m,1);

    std::vector<std::thread> trheads;
    int rpt = m/numberOfThreads    //rpt = row per thread
    for(int i=0; i<numberOfThreads; i++){
        threads.push_back(std::thread(mvprod, i*rpt, (i*rpt)+rpt));
    }
    for(auto & t: threads) t.join();
                    std::cout<< "le vecteur final:" << res << std::endl
    return 0;
}
```