Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Parallélisme & Applications

Examen – décembre 2024

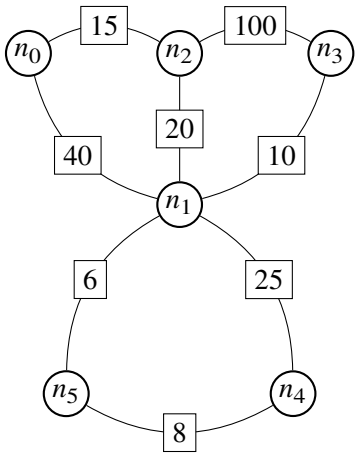
Durée : 1h30 — Documents autorisés

1 – 14pts Parallélisation de l'algorithme du « *Shortest Path Tree* » de Dijkstra :

d'après Wikipedia

Au départ, on considère que les distances de chaque nœud au **nœud de départ** sont infinies, sauf pour le **nœud de départ** pour lequel la distance est nulle.
Au départ, l'ensemble des nœuds sélectionnés est vide.
Au cours de chaque itération, on choisit un **nœud sélectionné** de distance minimale parmi les nœuds non encore sélectionnés, et on l'ajoute aux nœuds sélectionnés.
Ensuite, on met à jour la distance de chaque nœud voisin du **nœud sélectionné** : la nouvelle distance du **nœud voisin** est le minimum entre la distance existante et la distance calculée en faisant la somme de :

- ▷ la distance entre le **nœud de départ** et le **noeud sélectionné** (contenue dans la table des distances)
- ▷ la distance du **nœud sélectionné** à ce **nœud voisin** (contenue dans la matrice d'adjacence).



Le graphe exprimé en matrice d'adjacence :

```
#define MAX_INT 32768

#define    N  6
#define TRUE  1
#define FALSE 0

int graphe[N][N] = {
    { 0, 40,  15,   0,  0, 0 },
    {40,  0,  20,  10, 25, 6 },
    {15, 20,   0, 100,  0, 0 },
    { 0, 10, 100,   0,  0, 0 },
    { 0, 25,   0,   0,  0, 8},
    { 0,  6,   0,   0,  8, 0},
};
```

Les prototypes des fonctions à écrire/utiliser :

```
// Une fonction qui retourne l'indice du  noeud avec la distance minimale
// depuis l'ensemble des noeuds non déjà inclus dans le Shortest Path Tree
int distanceMin(int distance[], int ensembleSPT[]) {
      int min = MAX_INT, index_min;
...

// La fonction qui implémente l'algorithme du Shortest Path de Dijkstra
// depuis un noeud de depart pour un graphe représenté par une table d'adjacence
void dijkstra(int graphe[N][N], int depart) {
      int distance[N];  // Le tableau de sortie qui contiendra la distance la plus
                        // courte depuis le noeud de depart vers i

    int ensembleSPT[N]; // ensembleSPT[i] est vrai si le noeud est inclus dans le
                        // shortest path tree
...
```

Questions :

a. Exécuter l'algorithme à la main sur le graphe et donnez le tableau `distance` résultat. *(2pts)*
En combien d'étapes l'algorithme arrive au résultat ?

b. Écrivez le code de la fonction `distanceMin` utilisant le code fourni. *(2pts)*
Pourquoi a-t-on besoin de `MAX_INT` ?

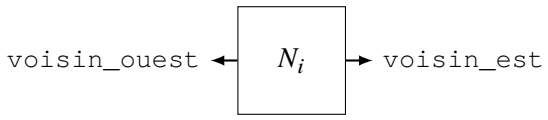
c. Écrivez le code de la fonction `Dijkstra` en version séquentielle utilisant le code fourni. *(3pts)*

d. **On considère toujours la construction du « Shortest Path Tree » pour un seul nœud de départ**. *(6pts)*
Quelles sont les sources potentielles de parallèlisation exploitables avec OpenMP dans cet algorithme ?
Quelle type de parallélisme allez vous exploiter ?
Pour les différentes threads OpenMP, quelles sont les données de l'agorithme :
◇ privées ;
◇ partagées ;
Est-ce qu'il y a des risques/problèmes pour l'accès à ces données ?
Est-ce que les « *tasks* » sont intéressantes ou non ?
Vous expliquerez pourquoi.
Décrivez la parallélisation de la fonction `Dijkstra` que vous allez appliquer.
Donnez le code utilisant OpenMP correspondant à votre proposition de parallélisation.

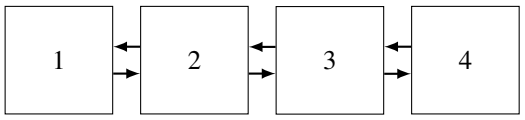
e. Est-ce que votre solution est à **coût optimal** ? *(1pt)*

2 – 6pts Pour un algorithme, on veut mettre en place une structure d'**anneau** sur un « *cluster MPI* ».

Chaque nœud du cluster, disposera d'au plus 2 voisins :



Exemple sur un cluster de 4 nœuds définissant un anneau de 4 :



On considérera que le nombre de nœuds de l'anneau est choisi par l'utilisateur au lancement du programme.

Questions :

a. Donnez le code réalisant le traitement suivant : *(1pt)*
▷ intégrer le nœud dans l'anneau ;
▷ définir par rapport au rang d'un nœud, le rang dans l'anneau de chacun de ses voisins s'il existe.

b. Donnez le code pour envoyer sucessivement par chaque nœud, une valeur entière contenue dans la variable `coefficient` présente sur chaque nœud à chacun de ses voisins, s'il existe, d'abord vers `voisin_est`, puis vers `voisin_ouest`. *(2pts)*

c. Lors de la réception de la valeur `coefficient` par un nœud depuis chacun de ses voisins : *(2pts)*
◇ est-ce que l'envoi et la réception vont être **synchronisés** ?
◇ est-ce que le nœud va recevoir les valeurs de chacun de ses voisins dans un **ordre prédéfini** ? Pourquoi ?
◇ est-ce que le nœud peut **savoir de quel nœud** il reçoit chaque valeur ?
Est-ce que cela peut **introduire des retards** dans le traitement de chaque réception ?
Comment **remédier** à ces problèmes ?
Vous donnerez les instructions à utiliser pour la réception des différents voisins.

d. À la fin de l'agorithme, un des nœuds de l'anneau est chargé de récupérer depuis chacun des nœuds de l'anneau, une valeur finale, stockée dans la variable flottante `résultat` présente sur chaque nœud. *(1pt)*

Quelle opération MPI allez vous utiliser pour réaliser cette réception depuis chaque nœud de l'anneau, sachant que c'est le premier nœud de l'anneau qui sera chargé de récupérer toutes ces valeurs.