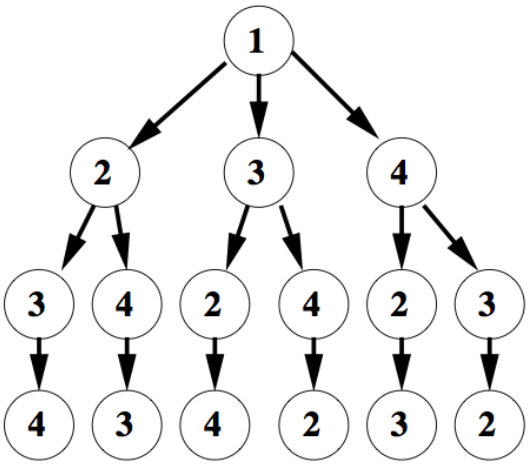
Projet PCM

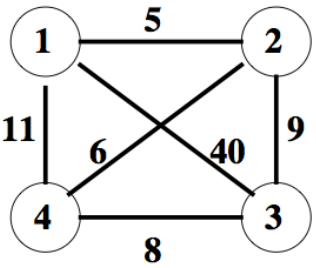
Thomas Zimmerman et Jean Nanchen

Introduction

Le but de ce projet est de travailler sur la parallélisation du problème du voyageur de commerce (TSP, Traveling Salesman Problem). Ce problème consiste en un voyageur qui doit passer par N villes une seule fois chacune en effectuant le chemin le plus court. Au départ d'une ville donnée, il y a (N-1)! circuits possibles. Le temps pour résoudre le problème est exponentiel, c'est pourquoi il est intéressant de le paralléliser sur plusieurs thread afin d'optimiser le temps de traitement.

Il existe différents algorithmes permettant de résoudre le problème du TSP. Celui que nous traiterons dans ce projet est le Branch-and-bound. L'idée est de séparer le problème en un arbre où chaque branche est un des chemins. La distance parcourue par le voyageur est calculée au fur et à mesure que les branches sont créées. Si une branche n'est pas terminée, mais est déjà plus longue que le chemin le plus court, elle est abandonnée. Cela permet de ne pas calculer la totalité des chemins possibles.



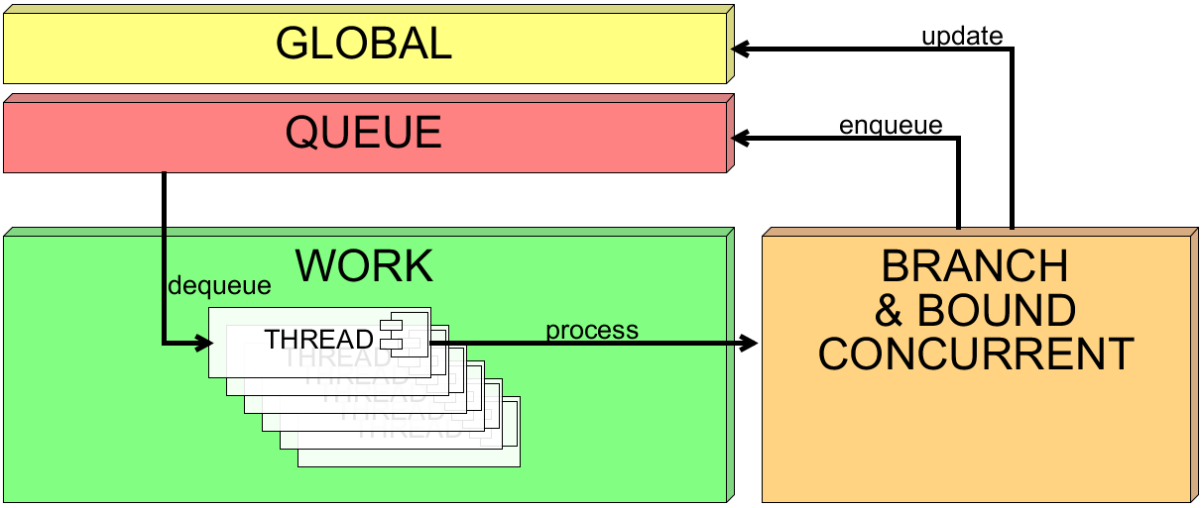


Ce rapport présente comment nous avons implémenté la parallélisation du problème, ainsi que la mise en place d'un cut off. Le cut off définit la profondeur à partir de laquelle les branches sont traitées de manière séquentielle. Nos résultats démontreront ensuite à quel point notre implémentation accélère le traitement du problème.

Implémentations développées

Les chiffres mentionnés dans cette partie sont présentés dans la partie “Expériences et mesures”

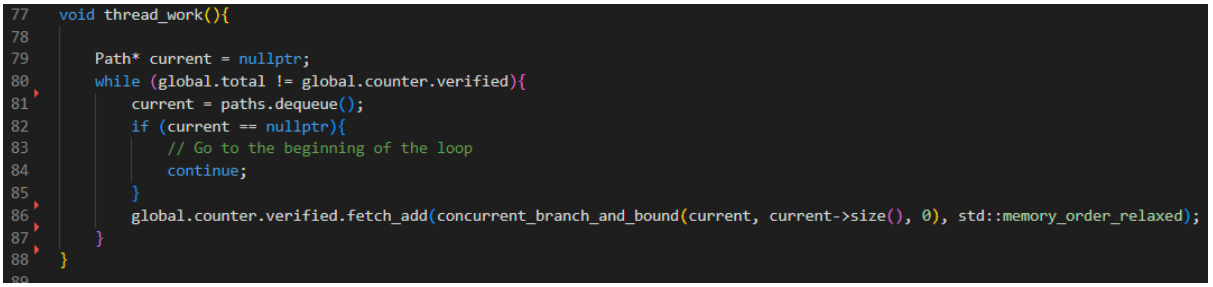
Fonctionnement général



La figure ci-dessus représente l’architecture software de notre programme. Une fonction nommée work contient une pool de thread. Chacun des threads va essayer de récupérer un travail dans la queue concurente à l’aide d’un dequeue. SI le thread à obtenu un travail, il exécute la fonction branch and bound concurrent. La fonction branch and bound concurrent est divisée en deux. Une partie qui crée des feuilles de et une autre partie qui calcul les chemins et compare le chemin le plus court actuel avec la structure globale. Nous avons implémenté un cutoff qui permet de créer à la fin des feuilles de manière séquentielle et non en remettant dans la queue à chaque fois.

Thread pool

Une pool de thread est créé et envoyé sur la fonction WORK. Ces threads vont donc récupérer un travail dans la queue concurrente et ensuite partir dans la fonction branch and bound concurrente. Le nombre de thread créé est défini au lancement du programme
La figure ci-dessous montre la fonction work ou la pool de thread pointe. On peut y observer la condition de sortie de la boucle. Les threads continuent de s’exécuter tant que tous les chemins n’ont pas été validés.



```
77  void thread_work(){
78
79      Path* current = nullptr;
80      while (global.total != global.counter.verified){
81          current = paths.dequeue();
82          if (current == nullptr){
83              // Go to the beginning of the loop
84              continue;
85          }
86          global.counter.verified.fetch_add(concurrent_branch_and_bound(current, current->size(), 0), std::memory_order_relaxed);
87      }
88  }
```

File concurrente

Comme expliqué dans le point précédent, les différents threads récupèrent le travail à effectuer dans une file. Cette file se doit d'être thread-safe car il est possible qu'un thread perde l'utilisation du processeur durant l'accès à un élément.

Afin de s'en assurer, nous avons converti le code java de file concurrente fourni dans le cours en c++.

Cut off

Comme expliqué brièvement dans l'introduction, cette fonctionnalité permet de définir une profondeur à partir de laquelle les branches sont traitées séquentiellement. Cela évite aux threads de passer plus de temps à essayer d'accéder au travail dans la file plutôt qu'à effectuer le travail nécessaire.

Lors de notre présentation, nous n'avions pas implémenté cette fonctionnalité. Ce qui limitait notre speed-up et nous ralentissait même au plus il y avait de threads. Une fois le cut off implémenté, notre speed-up s'est nettement amélioré et l'augmentation du nombre de threads a maintenant un effet positif sur la vitesse de traitement.

Incrément local

L'incrément local nous à aussi permis de gagner énormément de speed-up. Précédemment nous incrémentons constamment la variable atomique qui représente le nombre de chemins traités, ce qui prend énormément de temps. Maintenant nous avons une variable local qui compte le nombre de chemin traité qui est retourné par la fonction branch and bound et nous mettons à jour la variable atomique uniquement lors du retour de la fonction branch and bound dans la fonction work. Les lignes suivantes montrent l'emplacement des variables locales.

```
void thread_work(){
. . .
global.counter.verified.fetch_add(concurrent_branch_and_bound(current,
current->size(), 0), std::memory_order_relaxed);
. . .
```

```
}

static int concurrent_branch_and_bound(Path* current, int depth=0, uint64_t
localCounter=0) {

    if (current->leaf()){
        current->add(0);
        . . .
        return 1;
    }
    else {
        if (current->distance() <
global.shortest.load(std::memory_order_relaxed)->distance()) {
            . . .
            }
            else {
                for (int i=1; i<current->max(); i++) {
                    if (!current->contains(i)) {
                        current->add(i);
                        localCounter += concurrent_branch_and_bound(current,
current->size(), 0);
                        current->pop();
                    }
                }
            }
            return localCounter;
        }
        else {
            . . .
            localCounter += result;
            return localCounter;
        }
    }
}
```

Lookup table

Une lookup table à été créée pour éviter de calculer à chaque fois le nombre de chemins invalidés lors d'un bound.

Optimisations à la compilation

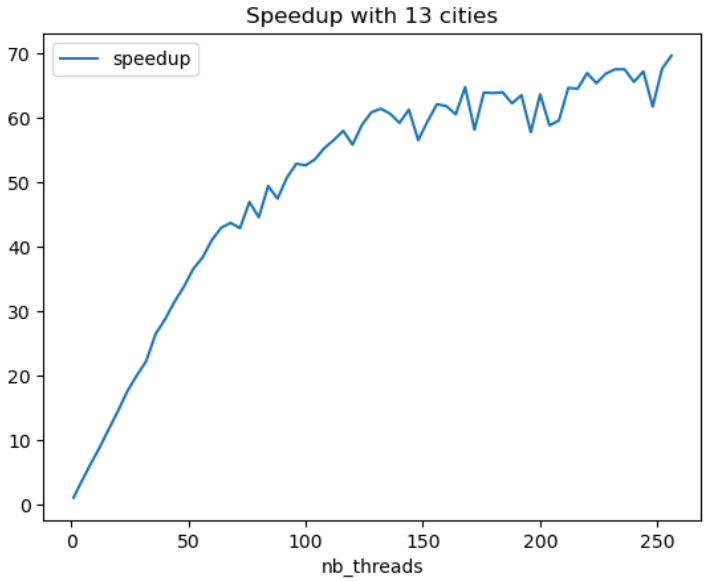
Un gain d'environ de 10 peut-être gagné si l'on compile à l'aide de l'instruction -O3.

```
g++ -g tspcc.cpp -o heyy.exe -latomic -pthread -O3
```

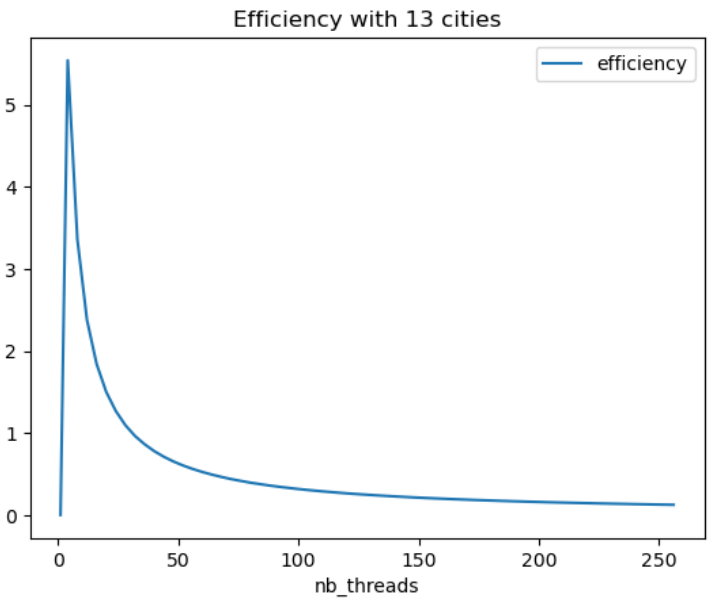
Expériences et mesures

Efficiency / Speedup

La mise en place de toutes les fonctionnalités discutées nous permet de baisser grandement le temps nécessaire à effectuer le problème en tirant parti de la parallélisation. Les graphs suivants présentent les résultats obtenus pour un problème à 13 ville:

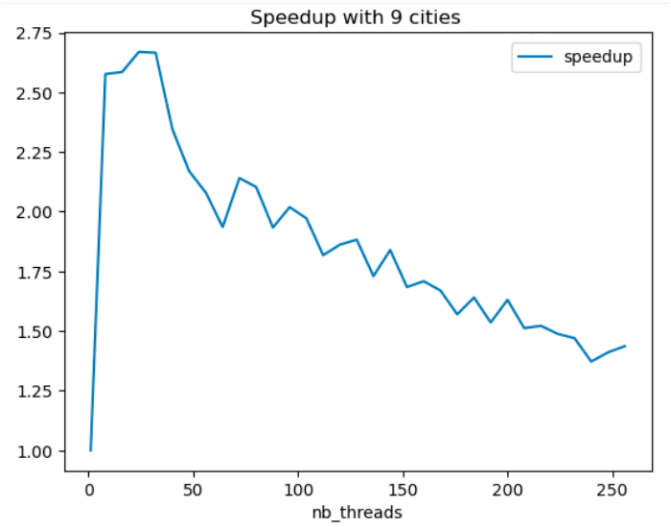


Le speedup exprime le rapport entre le temps nécessaire pour effectuer le problème avec N threads et le temps nécessaire pour effectuer le problème avec 1 thread. On peut remarquer qu'au plus on ajoute des threads, au moins ces threads font baisser le temps de traitement du problème. Le graph d'efficience ci-après présente cet effet:



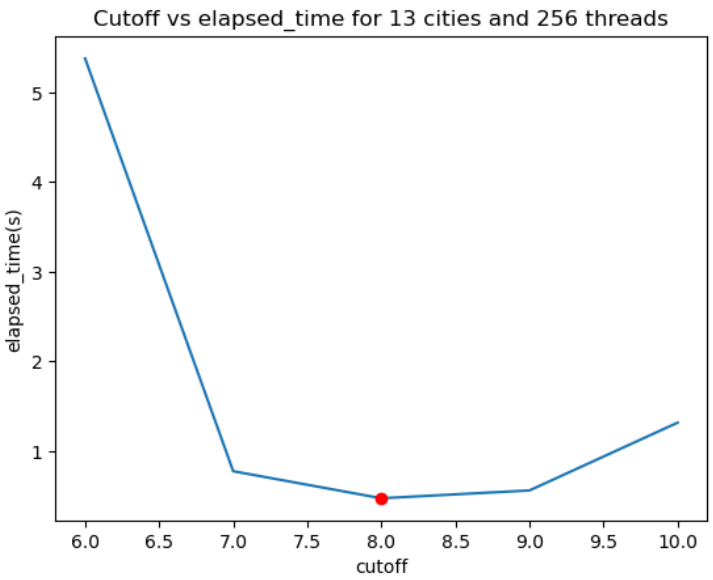
Cutoff

Comme expliqué précédemment, sans cutoff, l'augmentation du nombre de threads menait à une augmentation du temps de traitement. La figure ci-dessous présente cette augmentation:



La baisse du speedup traduit une augmentation du temps de traitement, ce comportement anormal est dû à l'absence de cutoff, les threads passent plus de temps à accéder à la queue qu'à effectuer le travail.

Nous avons donc implémenté le cutoff pour définir le meilleur, nous avons testé différentes valeur en utilisant 256 threads pour résoudre le problème à 13 villes.



Étant donné que nous avions évoqué en cours que la meilleure valeur se situait vers 8 ou 9, nous avons testé toutes les valeurs de 6 à 10. Un cutoff à 8 semble être la meilleure valeur.

Lancement du programme

Le programme se lance comme suit :

```
./heyy.exe -v2 dj38.tsp MIN_THREAD MAX_THREAD THREAD_INCREMENT
MIN_CITY MAX_CITY AVG CUTOFF_MIN CUTOFF_MAX
```

- MIN_THREAD : nombre de thread minimum
- MAX_THREAD : nombre de thread maximum
- THREAD_INCREMENT : incrément de THREAD_INCREMENT de MIN_THREAD jusqu'a MAX_THREAD
- MIN_CITY : nombre minimal de ville
- MAX_CITY : nombre maximal de ville
- AVG : nombre d'average effectué par la boucle
- CUTOFF_MIN : nombre minimal de cutoff
- CUTOFF_MAX : nombre maximal de cutoff

Le programme effectue différentes boucles pour faire des rampes du nombre de threads, du nombre de villes et de la valeur de cutoff. Il est possible aussi d'ajouter un average qui va relancer le programme plusieurs fois.

Un script do-it.sh à la racine du projet permet de lancer la compilation et le programme.

Conclusion

Dans ce projet nous avons pu transformer un algorithme séquentiel en algorithme parallèle et nous avons dû pour cela :

- Créer une pool de thread
- Créer une queue concurrente à l'aide d'opérations atomiques
- Rendre notre fonction branch and bound concurrente à l'aide de variables atomiques
- Réaliser un cutoff permettant un travail séquentiel à partir d'un palier
- Optimiser l'incrémentation des variables atomique pour augmenter la rapidité de notre fonction de branch and bound
- Optimiser la valeur cutoff avec notre algorithme

Nous avons pu grâce au l'exécution parallèle de notre programme passer de TODO secondes sur un thread à TODO secondes sur 256 threads.

Le maximum de villes que nous avons traité est de 16 et l'exécution prend 16 minutes avec 256 threads. Nous n'avons pas testé avec plus de villes car nous devions libérer la machine Phi.