



Projet Branch-and-Bound

MA-PCM

Diego Villagrasa

Pierre-Benjamin Monaco

30 mai 2023

Clean repository : https://github.com/IamFonky/MA-PCM_TSPCC

Dev repository : <https://github.com/IamFonky/MA-PCM-Projet>

1 Introduction

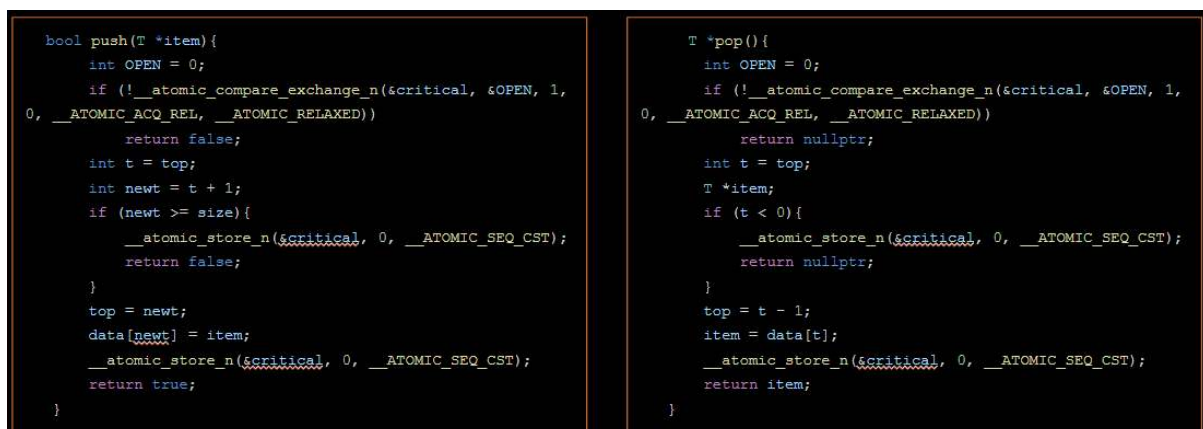
Pour le cours de programmation concurrente multicœurs, il nous est demandé d'implémenter une version parallélisable de l'algorithme de Branch and Bound pour la recherche du plus court chemin entre N villes. Le présent document résume le travail que nous avons effectué et les mesures obtenues au cours de notre travail.

2 Implémentations

Pour créer une version hautement parallélisée de l'algorithme "Branch and Bound", nous avons développé les implémentations suivantes :

2.1 Pile

Nous avons décidé d'utiliser une pile qui a pour rôle de stocker les tâches de Branch and Bound. Notre implémentation initiale de cette pile se basait sur le code fourni en cours mais ne fonctionnait pas correctement et se bloquait à certaines occasions. Nous avons donc décidé d'implémenter une sorte de mutex avec un compare exchange sur une variable partagée qui détermine si la section critique est déjà accédée ou pas.



```
bool push(T *item){
    int OPEN = 0;
    if (!__atomic_compare_exchange_n(&critical, &OPEN, 1,
    0, __ATOMIC_ACQ_REL, __ATOMIC_RELAXED))
        return false;
    int t = top;
    int newt = t + 1;
    if (newt >= size){
        __atomic_store_n(&critical, 0, __ATOMIC_SEQ_CST);
        return false;
    }
    top = newt;
    data[newt] = item;
    __atomic_store_n(&critical, 0, __ATOMIC_SEQ_CST);
    return true;
}

T *pop(){
    int OPEN = 0;
    if (!__atomic_compare_exchange_n(&critical, &OPEN, 1,
    0, __ATOMIC_ACQ_REL, __ATOMIC_RELAXED))
        return nullptr;
    int t = top;
    T *item;
    if (t < 0){
        __atomic_store_n(&critical, 0, __ATOMIC_SEQ_CST);
        return nullptr;
    }
    top = t - 1;
    item = data[t];
    __atomic_store_n(&critical, 0, __ATOMIC_SEQ_CST);
    return item;
}
```

FIGURE 1 – Pile

2.2 AtomicPath

Pour que les threads puissent déterminer si le chemin trouvé est plus petit que le plus court trouvé, nous avons décidé d'implémenter une classe AtomicPath qui garde un pointeur sur le chemin le plus court. Une méthode **copyIfShorter** permet de mettre à jour ce chemin s'il est effectivement plus court. Ceci est fait à l'aide d'un spinlock qui verrouille la méthode pour éviter des problèmes de concurrence. La méthode vérifie ensuite si le chemin est bien plus court et remplace le pointeur.

Nous avons décidé d'utiliser un spinlock car cette méthode n'est appelée que rarement et ne devrait pas avoir trop de concurrence.

```

class AtomicPath : public Path {
private:
    std::atomic_flag _lock = ATOMIC_FLAG_INIT;
public:
    AtomicPath(Graph* graph):Path(graph) {}

    void copyIfShorter(Path* o){
        while (_lock.test_and_set(std::memory_order_acquire)) {
            // Spin while the lock is not available
        }

        if(o->distance() < distance()){
            copy(o);
        }

        _lock.clear(std::memory_order_release); // Release the lock
    }
};

```

FIGURE 2 – AtomicPath

2.3 Distribution des tâches

Pour paralléliser l'algorithme, nous avons implémenté le modèle suivant :

1. Le programme démarre en plaçant un Path contenant le premier sommet dans la pile.
2. Un nombre N de thread consommateur sont créés et démarrés
3. Un thread récupère le Path de la pile, il incrémente un compteur atomique (std : :atomic_int) qui indique le nombre de threads actifs et effectue le Branch and Bound
4. Quand le thread termine l'étape du Branch and Bound, il regarde s'il a atteint la distance de cutoff. Si ce n'est pas le cas, il regarde s'il reste de la place dans la pile est push la suite du branch and bound dessus. Dans le cas contraire, il continue le branch and bound sur son thread.
5. Les autres threads piochent dans la pile et font le même travail.
6. Quand un thread arrive à une feuille, il vérifie si le chemin trouvé est plus court que celui trouvé globalement et si c'est le cas, le remplace grâce à un spinlock. En sortant, le compteur des threads actifs est décrémenté et le thread regarde s'il reste des threads actifs. Si ce n'est pas le cas le thread se termine, sinon il continue la boucle en tentant de récupérer un élément de la pile.

```

std::atomic_int nb_thread_running;

static void *branch_and_bound_task(void *arg)
{
    do
    {
        Path *job_to_do = global.jobs->pop();
        while (job_to_do != nullptr)
        {
            global.nb_thread_running++;
            branch_and_bound(job_to_do);
            global.nb_thread_running--;
            job_to_do = global.jobs->pop();
        }
    } while (global.nb_thread_running > 0);
    return 0;
}

static void branch_and_bound(Path *current){
    [...]
    bool pushed = false;
    if ((current->size() < (current->max() - config.cutoff_depth))
        && (global.jobs->get_size() < config.stack_size)){
        Path *newPath = new Path(current);
        pushed = global.jobs->push(newPath);
        if (!pushed)
            delete newPath;
    }
    if (!pushed)
        branch_and_bound(current);
    current->pop();
}

```

FIGURE 3 – Threaded Branch and Bound

3 Expériences

Pour commencer, nous avons commencé avec des tests en local pour définir la taille optimale de la stack. Des tests ont été faits d'abord avec 10 et 12 villes et pour un nombre de threads entre en 1 et 100 (sachant que nos ordinateurs n'ont que 12 ou 16 threads matérielles, il n'était pas nécessaire d'aller aussi loin).

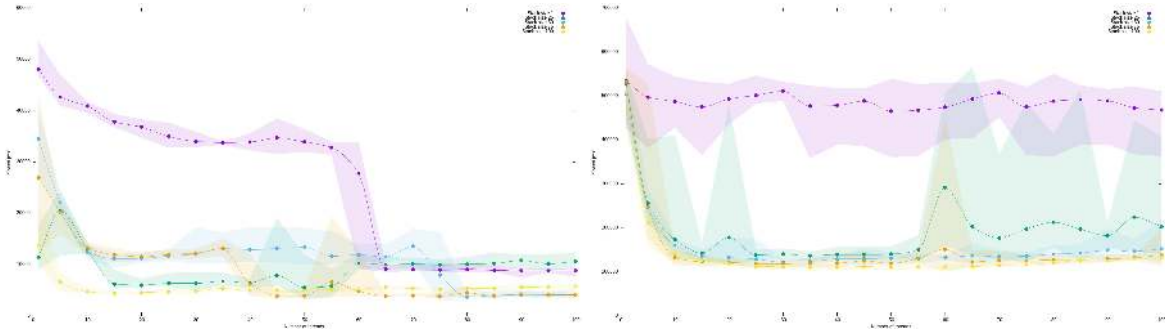


FIGURE 4 – Testing stack size for N=10 and N=12

Voyant que plus la stack était grande, plus la vitesse était grande, nous avons décidé de tester avec des stacks de 1k, 2k, 3k et 4k avec N=17 sur le serveur. Avec les résultats obtenus nous avons décidé de bloquer la taille de la stack à 1024.

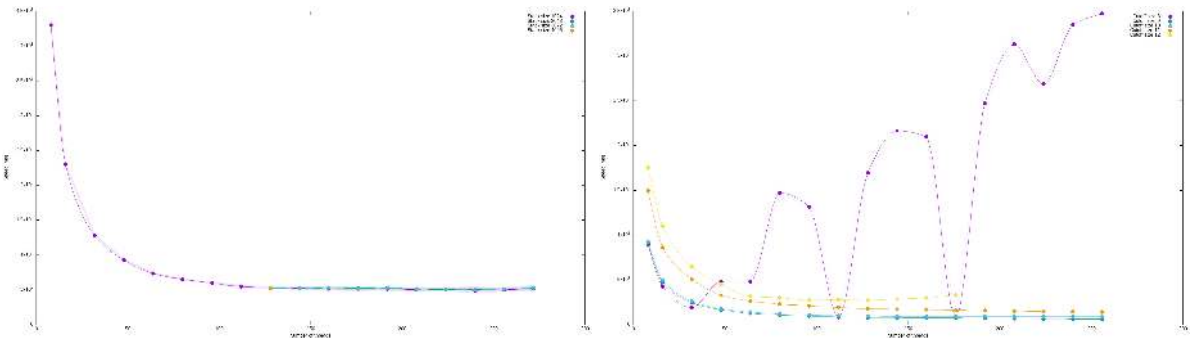


FIGURE 5 – Testing stack size for N=17 and cutoff size for N=16

Nous avons ensuite tenté de tuner la valeur du cutoff (la taille minimum d'un chemin à stocker dans la stack). La meilleure valeur trouvée est un cutoff à 9, avec un arbre plus petit les résultats deviennent trop variables.

En augmentant le nombre de villes, nous avons commencé à avoir des résultats de longueur de chemin farfelus. C'est à ce moment-là que nous avons compris que le code utilisé pour implémenter la stack concurrente n'était pas juste et qu'il était toujours possible pour deux threads de se retrouver dans la section critique en même temps. Comme nous n'avions plus beaucoup de temps à disposition, nous avons décidé d'opter pour une solution simple, la linéarisation totale du pop et du push mais sans spinlock. Si une thread n'arrive pas à pop, c'est n'est pas grave car elle se trouve dans une boucle et si elle n'arrive pas à push, il lui suffit d'allonger son chemin d'une ville et de réessayer après, autant que le temps d'attente soit utilisé pour calculer le plus court chemin.

Nous avons donc testé deux implémentations, une avec un mutex (et un trylock) et une autre avec une variable atomique utilisée comme un mutex (un loquet) qui passe à 1 quand une thread se trouve dans une section critique et qui repasse à 0 quand elle y sort.

Les résultats (Figure 6 gauche) ont montré que la solution d'une variable atomique est nettement plus performante que celle avec le mutex, c'est donc celle-ci qui a été choisie comme implémentation principale.

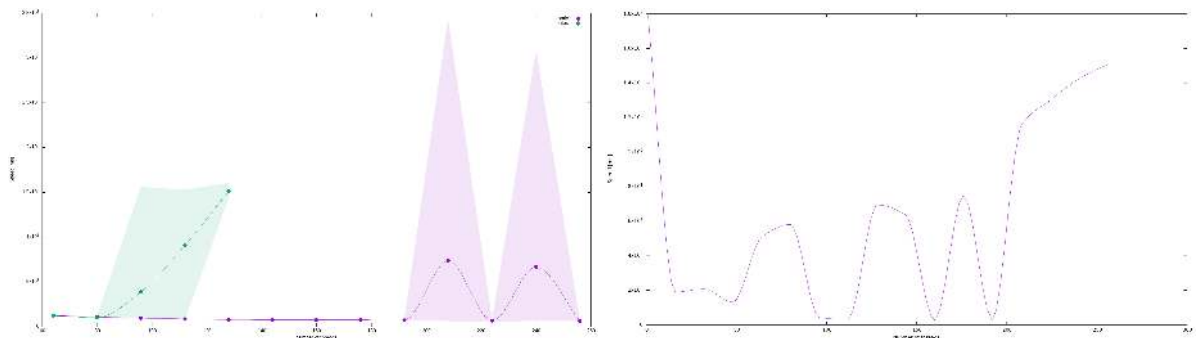


FIGURE 6 – Testing two ways of protecting a critical section and tests with $N=16$, $t=[1,256]$, $stack=1024$

Avec un système normalement bien tuné, des tests à plus grande échelle avec un nombre de threads variant entre 1 et 256 et avec un N de 16 a été effectué et les résultats n'ont pas du tout données les valeurs escomptées. La Figure 6 (droite) présentées en classe illustre le plus gros problème rencontré. Nous avons décidé d'enquêter sur les raisons d'une telle différence de vitesse et le manque de cohérence.

Nous avons défini 3 scénarios qui pourraient potentiellement amener à de tels résultats :

- Le nombre d'interactions avec la stack peuvent être très variables en fonctions des chemins pris et l'implémentation de celle-ci ralentirait tout le programme
- Le nombre de tests et de remplacement des plus court chemins pourraient ralentirait le programme
- La condition de terminaison des threads est trop laxiste et des threads terminerait leur exécution avant la fin de tous les calculs.

Pour découvrir lequel/lesquels de ces trois points est vrai, nous avons ajoutés des compteurs dans le code : un compteur de nombre d'appels à la stack (pop, push) et un compteur d'appel au test et remplacement du plus court chemin.

Ci-dessous, le tableau 1 montre un exemple des différences de timings entre 3 échantillons avec des paramètres similaires. On voit que les tailles de chemins sont toujours les mêmes, on voit aussi que le nombre de pop correspond toujours au nombre de push + 1 (car on push le premier chemin avant de lancer les threads). On observe également que le temps de complétion ne dépend pas du nombre d'accès à la stack ou au plus test du plus court chemin. Il ne reste plus qu'une solution parmi les 3 listées ci-dessus, les threads se terminent avant que calculs soient terminés.

Timing analysis						
Nb threads	Stack size	TSP result	TTC [μ s]	Nb pop	Nb push	Nb SP tests
80	1024	864	586450607	1866866	1866865	72
80	1024	864	605548064	663139	663138	76
80	1024	864	46634375	2986987	2986986	75

TABLE 1 – Example of discrepancies between samples with similar parameters

Un nouveau test a été effectué en calculant le temps entre l'arrêt de la première et la dernière thread et les résultats montrent que toutes les threads sont vivantes jusqu'au bout du calcul.

Timing analysis					
Nb threads	TSP result	TTC [μ s]	Nb pop	Nb SP tests	delta dead thread [μ s]
96	864	52603388	1721045	75	12094
80	864	449738919	661505	67	10226

TABLE 2 – Testing delta time between first and last dead thread

Un dernier test à été effectué en calculant le nombre de spinlock lorsqu'un nouveau chemin est trouvé. Malheureusement il ne donne pas plus d'information sur les problèmes de ralentissement.

Timing analysis					
Nb threads	TSP result	TTC [μ s]	Nb pop	Nb SP tests	Nb spinlocks
80	864	72672960	3535823	81	0
80	864	705625395	680561	68	0

TABLE 3 – Testing number of spinlocks

Le problème n'a pas été résolu cependant, il est clair que celui-ci survient quand les threads ont trop peu de travail ou quand le travail est trop petit. Afin d'avoir des mesures significatives, le cutoff a été relevé à 11 (pour rappel, une thread en avec N=12 met environ une seconde à s'effectuer, faire les tests avec un cutoff de 11 ne va donc normalement pas trop déséquilibrer la répartition de la charge sur les threads).

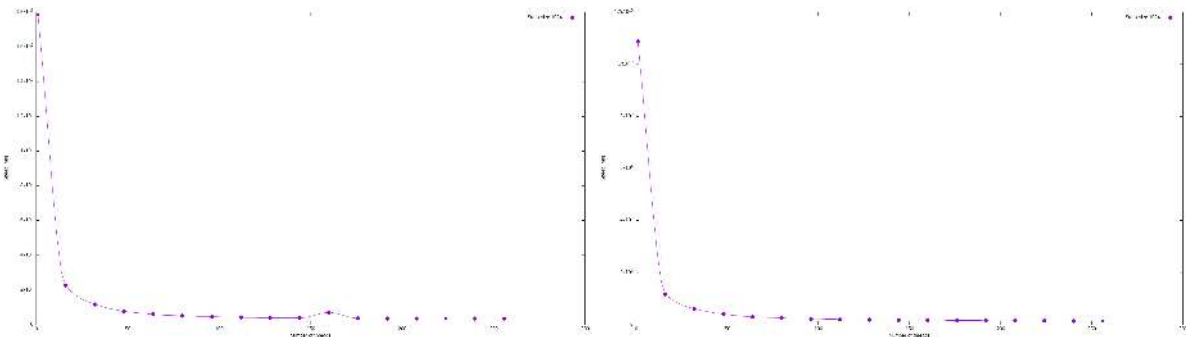


FIGURE 7 – Testing speed from 1 to 256 thread with N=17 and N=18

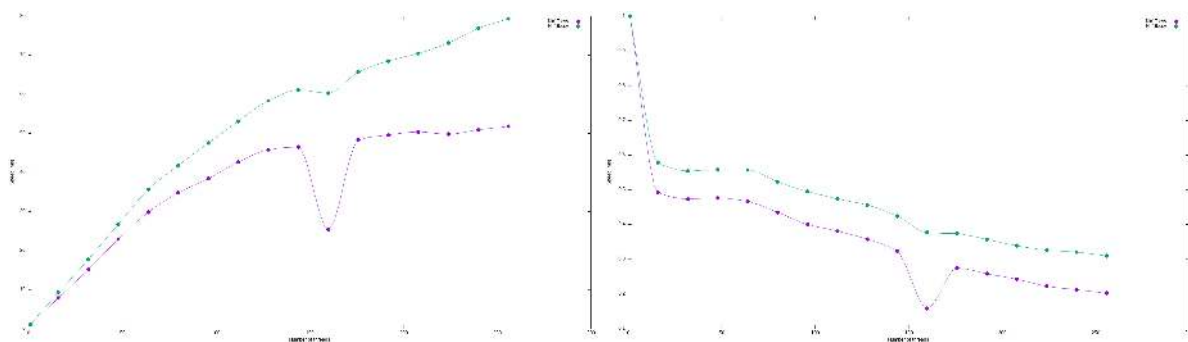


FIGURE 8 – Speedups and efficiency for N=17 and N=18

La figure 8 ci-dessus montrent que plus le nombre de villes est grand, plus le speedup et parallèlement l'efficacité augmente. On atteint finalement un speedup de x79 pour N=18 et de x51 avec N=17 pour 256 threads. L'efficacité mesurée pour 256 threads est respectivement de 20% pour N=17 et 30% pour N=18. Au final, la recherche du plus court chemin prend 34 secondes pour N=17 et 136 secondes pour N=18.

4 Conclusion

Le projet Branch and Bound s'inscrit dans le cours MA Programmation Concurrente Multicœurs. Dans le cadre de ce projet, nous avons pu explorer l'implémentation d'un algorithme parallélisable. Cet algorithme n'est pas trivialement parallélisable (comme du traitement de signal par convolution ou de la simulation de corps célestes) car les calculs dépendent des résultats précédents. Les calculs doivent être assignés dynamiquement pendant l'exécution du programme.

Les choix de structure de données et de contrôle ont été motivés à partir de modèles vus en classe et de mesures effectuées durant le projet. Pour des questions de temps, il n'a pas été possible d'explorer la variété de possibilités offertes pour résoudre le problème mais juste un aperçu des solutions possibles. Plusieurs problèmes et améliorations restent sans réponse mais certaines pistes ont déjà été explorées et écartées. Le code fourni contient une multitude de configurations et de tests qui permettront à un futur développeur motivé de continuer la recherche.

Au final, le fait de concrétiser les concepts vus en classe a été très formateur et nous avons appris par la pratique certains points cités en cours, notamment que l'efficacité de la parallélisation diminue avec le nombre de cœurs/threads et qu'il n'est pas possible de multiplier indéfiniment la vitesse d'un problème juste en y ajoutant des ressources matérielles. La deuxième observation que nous faisons est la difficulté de déboguer un code massivement parallèle. En effet, l'effort de profiling est d'autant plus grand qu'il n'y a de possibilité de parallélisme dans un code et le temps nécessaire pour effectuer des mesures significatives est énorme. Par exemple, si une modification dans le code est effectuée pour améliorer la performance lors de l'exécution avec 256 threads, des tests doivent aussi être conduits avec moins de threads (jusqu'à idéalement une seule thread) pour s'assurer que le code fonctionne encore dans tout son spectre d'utilisation. Un test qui prend 136 secondes avec 256 threads prendra 3h pour une seule thread, tout ce qui peut être automatisé est donc bienvenu encore plus dans ces circonstances.