Projet d'OS202: Feux de forêts

Pierre Bordeau, Gabriel Dupuis

14 mars 2025

Table des matières

1 Première étape

1.1 Question 1

Voici un exemple de ce que `lscpu` donne :

```
Architecture :                                  x86_64
  Mode(s) opératoire(s) des processeurs :       32-bit, 64-bit
  Address sizes:                                48 bits physical, 48 bits
   ↪  virtual
  Boutisme :                                    Little Endian
Processeur(s) :                                 16
  Liste de processeur(s) en ligne :             0-15
Identifiant constructeur :                      AuthenticAMD
  Nom de modèle :                               AMD Ryzen 7 7735U with Radeon
   ↪  Graphics
    Famille de processeur :                     25
    Modèle :                                    68
    Thread(s) par cœur :                        2
    Cœur(s) par socket :                        8
    Socket(s) :                                 1
    Révision :                                  1
    Vitesse maximale du processeur en MHz :     4819,0000
    Vitesse minimale du processeur en MHz :     400,0000
    BogoMIPS :                                  5390.15
Virtualization features:
  Virtualisation :                              AMD-V
Caches (sum of all):
  L1d:                                          256 KiB (8 instances)
  L1i:                                          256 KiB (8 instances)
  L2:                                           4 MiB (8 instances)
  L3:                                           16 MiB (1 instance)
NUMA:
  Nœud(s) NUMA :                                1
  Nœud NUMA 0 de processeur(s) :                0-15
Vulnerabilities:
  Gather data sampling:                         Not affected
  Itlb multihit:                                Not affected
  L1tf:                                         Not affected
```

```
Mds:                                    Not affected
Meltdown:                               Not affected
Mmio stale data:                        Not affected
Reg file data sampling:                 Not affected
Retbleed:                               Not affected
Spec rstack overflow:                   Vulnerable: Safe RET, no
↪ microcode
Spec store bypass:                      Mitigation; Speculative Store
↪ Bypass disabled via prctl
Spectre v1:                             Mitigation; usercopy/swapgs
↪ barriers and __user pointer sanitization
Spectre v2:                             Mitigation; Retpolines; IBPB
↪ conditional; IBRS_FW; STIBP always-on; RSB filling; PBRSB-eIBRS
                                         Not affected; BHI Not
                                         ↪ affected
Srbds:                                  Not affected
Tsx async abort:                        Not affected
```

On en extrait sur les différentes machines qui seront utilisées pour les calculs :

Machine	Gabriel	Pierre	ENSTA - serveur info1
Coeurs	8	2	80
Mémoire cache L1	512 KiB	64 KiB	2,6 MiB
Mémoire cache L2	4 MiB	1 MiB	40 MiB
Mémoire cache L3	16 MiB	4 MiB	55 MiB x 2 instances

TABLE 1 – Table

Après avoir créé une fonction mesurant le temps d'exécution d'une méthode donnée, on obtient :

Moyenne temporelle du pas de temps	8	2
Moyenne temporelle de l'affichage L1	512 KiB	64 KiB

TABLE 2 – Table

1.2 Question 2

On écrit une fonction permettant de mesurer le temps d'éxecution d'une méthode associée à un objet donné :

```cpp
// Fonction générique pour mesurer le temps d'exécution d'une méthode
template<typename Obj, typename Method, typename... Args>
auto measure_time(bool condition, Obj&& objet, Method&& methode, Args&&...
    args) {
    if(condition){
        auto start = std::chrono::high_resolution_clock::now();

        auto result = (std::forward<Obj>(objet).*std::forward<Method>(method
            e))(std::forward<Args>(args)...);

        auto stop = std::chrono::high_resolution_clock::now();
        auto duration =
            std::chrono::duration_cast<std::chrono::microseconds>(stop -
            start);

        std::cout << "Temps d'exécution : " << duration.count() << "
            microsecondes" << std::endl;
        return result;
    }
    else{
        auto result = (std::forward<Obj>(objet).*std::forward<Method>(method
            e))(std::forward<Args>(args)...);
        return result;
    }
}
```

On obtient alors le tableau de moyennes :

n	Moyenne temps d'exécution pas de temps durant la simulation (μs)	Moyenne temps d'exécut
20	215	
50	755	
100	1590	

TABLE 3 – Table

1.3 Question 3

On parallélise l'avancement en temps (mise à jour de la simulation au fil du temps) avec OpenMP. C'est donc la méthode `Model::update()` qui est parallélisée.

D'abord, en analysant bien le code actuel de la méthode, nous avons remarqué qu'il ne se prête pas très bien à la parallélisation. En effet, on boucle sur chaque cellule du feu actuel (`m_fire_front`), et on met à jour la cellule actuelle ainsi que ses voisines si besoin. Le problème est donc que lorsque qu'on arrive sur une cellule, elle peut avoir déjà été modifiée lorsqu'on étudiait une de ses voisines. C'est notamment le cas pour la carte de l'incendie, `m_fire_map`.

On aura donc du mal à reproduire exactement la même simulation avec un code parallélisé, car on ne peut pas garantir que les cellules sont mises à jour dans le même ordre, ce n'est pas possible avec plusieurs threads.

La parallélisation a été faite en créant d'abord des vecteurs dans lesquels les threads pourront stocker les cellules à mettre à jour et à supprimer. Ensuite, une fois que tous les calculs sont effectués en parallèle, un unique thread met à jour les cartes globales.

La version initiale du code utilise une boucle sur les éléments de la `std::unordered_map` pour traiter les cellules. Le problème est que comme c'est une structure de données non ordonnée, il n'y a pas d'ordre d'itération garanti. On ne peut donc pas utiliser cette structure pour la parallélisation. Ainsi, il faut commencer par créer un vecteur contenant les clés qui nous seront utiles pour la boucle `for`. parallélisée.

Voici la méthode modifiée, qui indique dans l'ensemble la logique de parallélisation :

```
1  bool Model::update() {
2      // Copie de m_fire_front pour que tous les threads travaillent sur le
       ↪  même état initial
3      std::unordered_map<std::size_t, std::uint8_t> current_front =
       ↪  m_fire_front;
4
5      // Collection des clés pour la parallélisation
6      std::vector<std::size_t> m_keys;
7      for (const auto& f : current_front) {
8          m_keys.push_back(f.first);
9      }
10
11     // On créé des containers pour stocker les chgangelments
12     std::vector<std::unordered_map<std::size_t, std::uint8_t>>
       ↪  thread_local_additions;
13     std::vector<std::vector<std::size_t>> thread_local_removals;
14
15     #pragma omp parallel
16     {
```

```
17          #pragma omp single
18          {
19              int num_threads = omp_get_num_threads();
20              thread_local_additions.resize(num_threads);
21              thread_local_removals.resize(num_threads);
22          }
23
24          int thread_id = omp_get_thread_num();
25          auto& local_additions = thread_local_additions[thread_id];
26          auto& local_removals = thread_local_removals[thread_id];
27
28          #pragma omp for schedule(dynamic, 64)
29          for (size_t i = 0; i < m_keys.size(); ++i) {
30          ...
31          }
32        ...
33  }
```

Les résultats sont très mauvais. La version séquentielle est plus rapide que la version parallélisée quand celle-ci ne dispose que de peu de cœurs. Ceci est sûrement dû au fait que la version parallélisée doit créer des vecteurs pour chaque thread pour gérer la concurrence, ce qui ralentit le tout. De plus, une partie du travail ne peut toujours être effectuée que par un seul thread, c'est notamment le cas de la mise à jour de la carte globale avec les modifications calculées en parallèle.

Une explication pourrait être que le programme est *memory bound*, c'est-à-dire que la vitesse d'exécution est limitée par la vitesse d'accès à la mémoire, notamment quand on crée des vecteurs, enregistre les clés, met à jour les cartes, etc.

Voici quelques résultats obtenus sur les serveurs de l'ENSTA, en désactivant l'affichage et pour une carte de 300x300.

Version	Temps d'exécution (s)	Speedup
Version initiale	4,1 s	1,00
Parralèle - 1 coeur	7,55 s	0,54
Parralèle - 2 coeurs	6,76 s	0,61
Parralèle - 4 coeurs	6,98 s	0,59
Parralèle - 8 coeurs	6,35 s	0,65
Parralèle - 16 coeurs	6,34 s	0,65
Parralèle - 80 coeurs	6,03 s	0,68

TABLE 4 – Temps d'exécution et speedup, sans affichage

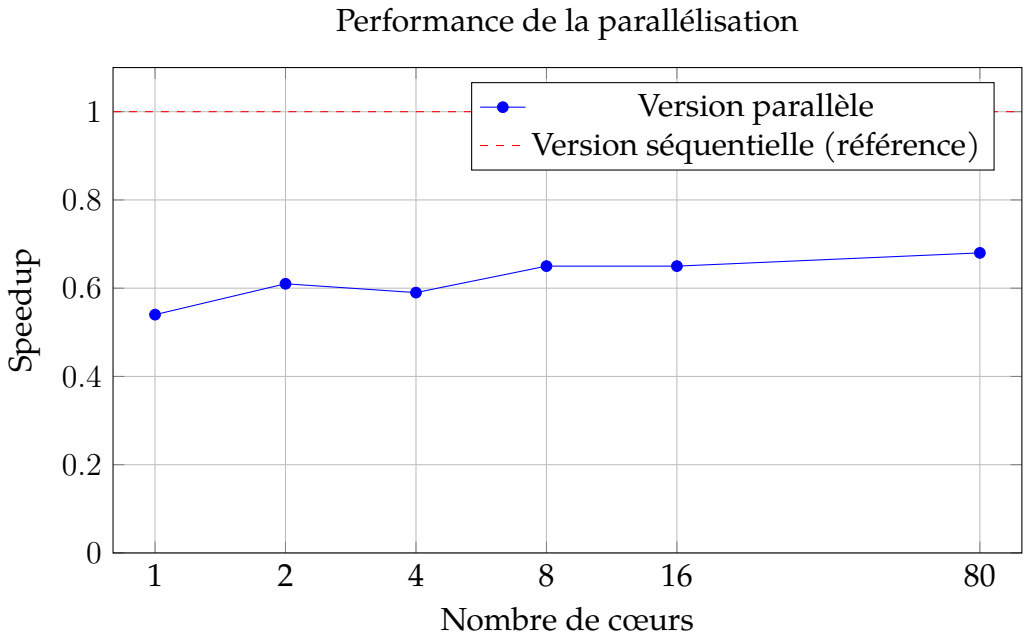

FIGURE 1 – Comparaison des speedups en fonction du nombre de cœurs (échelle logarithmique)

On peut voir que le speedup obtenu lors de l'utilisation de la version parallélisée est plus petit que 1…, ce qui est complètement inutile.