Master informatique CHPS

UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

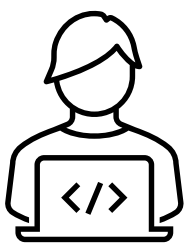
CHPS0701

Rapport : Projet

Timothé KRUK

2025

Table des matières



I. Introduction

Les problèmes universitaires sont l'occasion de pouvoir explorer de nouvelles méthodes qui peuvent in fine être appliquées à un contexte industriel. Aujourd'hui, nous nous intéressons à la résolution du problème de Langford. Il s'agit de mettre en application nos connaissances acquises pour la programmation parallèle sur CPU via les standards d'OpenMP et MPI. Par ailleurs, nous nous intéressons à l'optimisation de la pipeline de test pour accélérer notre travail. Cela passe par la réalisation de données qui sont stockées durant l'exécution puis traitées par des scripts Python pour générer une page Web permettant la visualisation de nos résultats.

II. Problème de Langford

A. Présentation

Le problème de Langford, également appelé série de Langford, consiste à trouver une séquence d'entiers où chaque entier k (allant de 1 à F) apparait exactement deux fois, et les deux occurrences de k sont séparées par exactement k positions.

Par exemple pour F = 3, une solution valide est : « 3, 1, 2, 1, 3, 2 » Dans cette séquence :

- Les deux 1 sont séparés par exactement 1 caractère.
- Les deux 2 sont séparés par exactement 2 caractères.
- Les deux 3 sont séparés par exactement 3 caractères.

Le problème de Langford est principalement utilisé pour étudier des concepts de combinatoire, programmation parallèle, et optimisation, car sa résolution demande une exploration efficace de nombreuses configurations possibles.

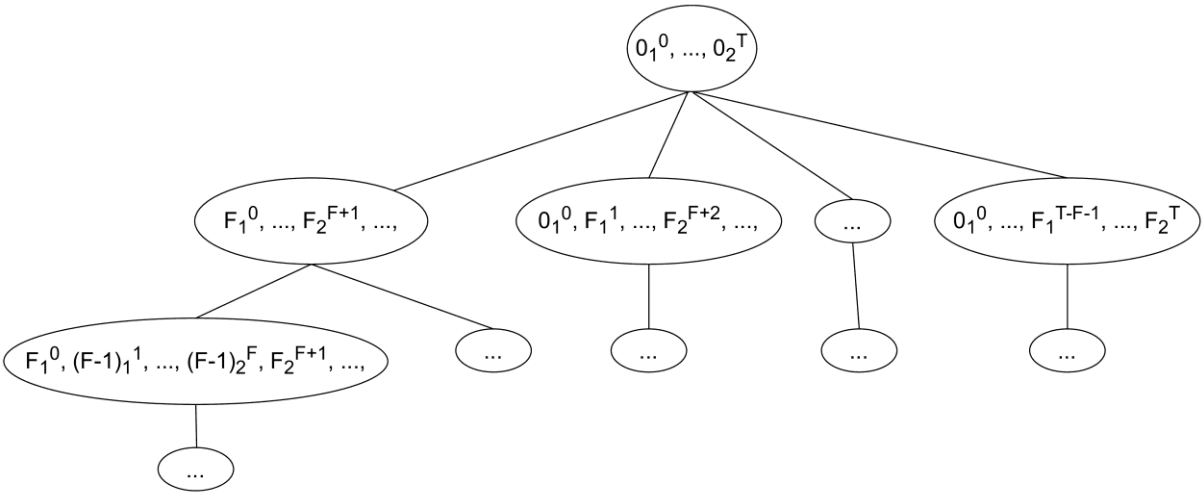
B. Modélisation

Nous notons le problème de Langford de la manière suivante : L(COUNT, FREQUENCY) avec les composantes suivantes.

COUNT (C) : Le nombre de répétions de chaque caractère (C = 2 dans notre étude).
FEQUENCY (F) : La fréquence d'apparition de chaque caractère.
LENGTH (T) : Le nombre de caractère totales dans chaque solution (LENGTH = COUNT * FREQUENCY)
$X_{ITTERATION}^{POSITION}$: X représente le caractère, POSITION indique sa position dans la séquence, ITTERATION correspond au nombre d'occurrences actuelles de X

Si on veut résoudre le problème de Langford, on peut représenter sa résolution sous forme d'arbre. La racine correspond simplement à une séquence initiale de zéros. À chaque

fois que l'on descend en profondeur, on place les caractères d'une fréquence inférieure à la précédente. Ainsi, si on a placé les caractères sans avoir brisé les règles de la séquence de Langford pour y parvenir, alors on a trouvé une solution. Voici une brève représentation d'un arbre générique pour un problème L(2, F). À noter que l'on ne représente pas tous les éléments, et que certaines parties de l'arbre ne sont pas explorées.



C. Algorithme récursive

Nous définissons un premier algorithme pour résoudre le problème de Langford. Il s'agit d'une première version qui va servir de base, enfin cette version est la plus intuitif.

```
Procédure resolve(in count : uint, in frequency : uint, in state : Langford, in
current_frequency : uint, out solutions : Langford[])
  Si current_frequency == 0 Alors solutions.push(state) /* Cas où tous les caractères sont
placés */
  Sinon
    Pour i allant de 0 à count Faire
      Si state[i] == 0 Alors
        new_state = state.copy()
        current_count ← 0, j ← i
        Tant que j < count * frequency ET current_count < count ET new_state[j] == 0 Faire
          new_state[j] ← current_frequency
          current_count++
          j ← j + current_frequency + 1
        FinTantQue
        Si current_count == count Alors
          resolve(count, frequency, new_state, current_frequency - 1)
        FinSi
      FinSi
    FinPour
  FinSi
```

```
FinProcédure
```

D. Optimisation

Lors de la résolution du problème de Langford, il est essentiel d'optimiser les calculs en évitant les explorations redondantes dues aux solutions symétriques. Une solution symétrique est une solution, qui après inversion, reste identique à une autre solution déjà trouvée.

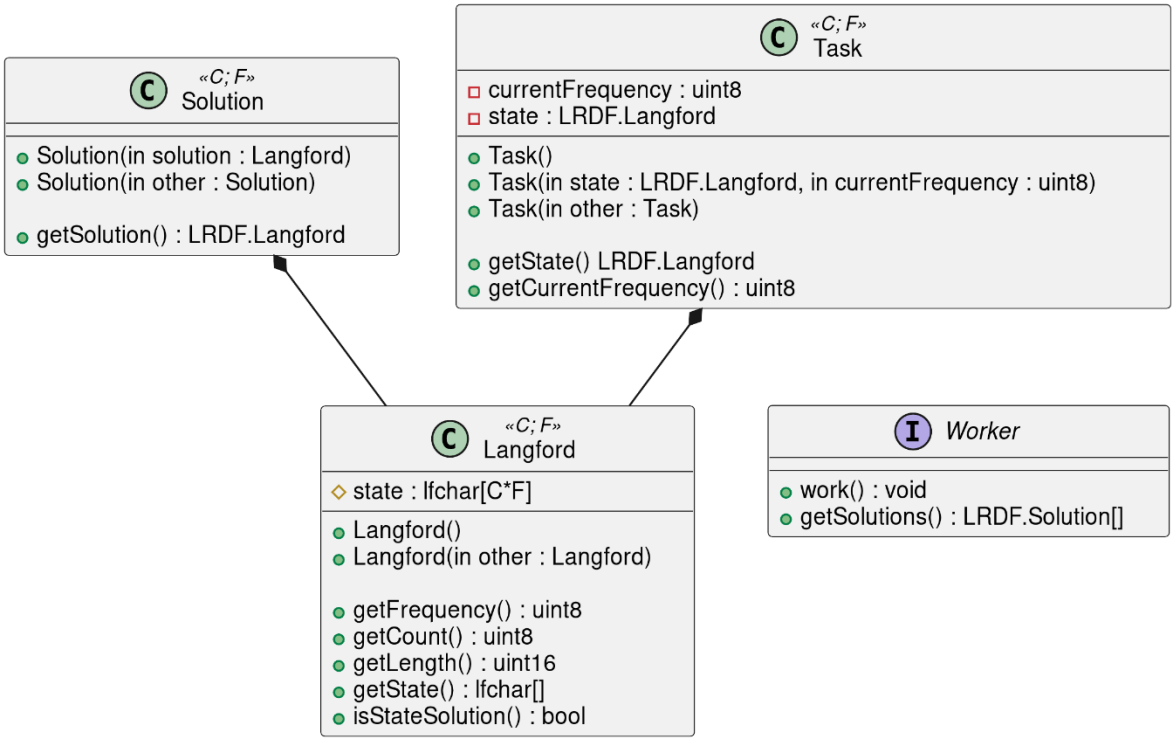
Voici la procédure modifiée intégrant la vérification de la symétrie.

```
Procédure resolve(in count : uint, in frequency : uint, in state : Langford, in
current_frequency : uint, in is_symmetry_check : bool, out solutions : Langford[])
  Si current_frequency == 0 Alors
    solutions.push(state) /* Cas où tous les caractères sont placés */
  Sinon
    is_pair ← current_frequency % 2 == 0
    Pour i allant de 0 à count Faire
      Si is_symmetry_check ET
        ((¬is_pair ET i + 1 > count * frequency - (i + current_frequency + 2)) OU
        (is_pair ET i + 1 > count * frequency - (i + current_frequency + 1))) Alors
        break /* Arrêt de l'exploration pour éviter les solutions symétriques */
      FinSi

      Si state[i] == 0 Alors
        new_state ← state.copy()
        current_count ← 0, j ← i

        Tant que j < count * frequency ET current_count < count ET new_state[j] == 0 Faire
          new_state[j] ← current_frequency
          current_count++
          j ← j + current_frequency + 1
        FinTantQue

        Si current_count == count Alors
          next_is_symmetry_check ← is_symmetry_check ET is_pair ET (i == count *
frequency - (i + current_frequency + 2))
          resolve(count, frequency, new_state, current_frequency - 1,
next_is_symmetry_check, solutions)
        FinSi
      FinSi
    FinPour
  FinSi
FinProcédure
```

III. Méthode Aruka

A. Objets essentiels

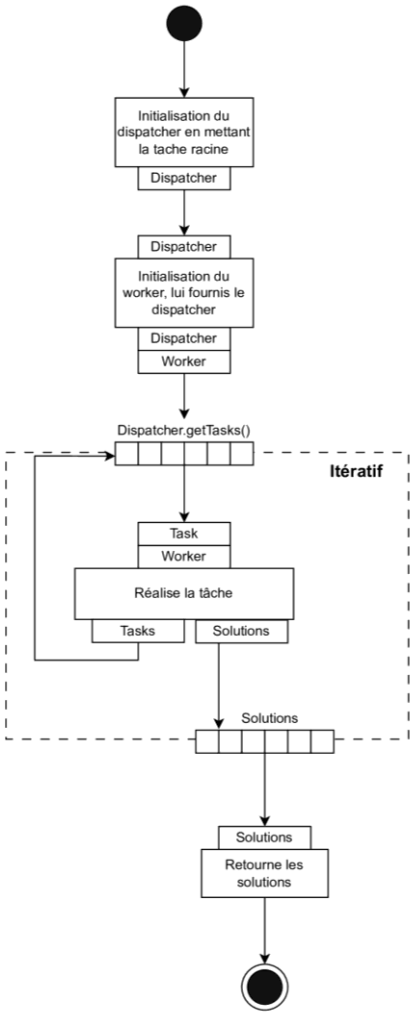
La première étape est de définir un ensemble d'objets fondamentaux que nous allons utiliser pour résoudre le problème de Langford dans nos différentes approches. Voici le MCD de ces dits objets.



L'objet Langford permet de stocker l'état d'une séquence de caractères. Il est à la fois utilisé dans l'objet Solution, qui représente une solution, et aussi dans l'objet Task. Enfin, nous définissons l'objet Worker. Les workers peuvent aussi bien représenter un thread qu'un programme en fonction du contexte. Le choix de l'utilisation de templates et de l'interface sera expliqué davantage dans la partie modélisation. Pour terminer, nous appelons l'objet Dispatcher, qui a pour but de stocker les différentes tâches que doit effectuer un ou un ensemble de workers.

B. Algorithme séquentielle de référence

La seconde étape est de convertir notre algorithme en un algorithme itératif pour plus d'efficacité. Pour cela, nous nous basons sur l'utilisation d'une structure de type file. Voici ci-dessous le diagramme d'activité expliquant notre algorithme.



C. Collecte de données

Notre objectif premier est de fournir une stratégie dont l'implémentation résout le problème de Langford le plus rapidement possible avec le minimum de mémoire utilisée. Il s'agit des deux points les plus importants. Néanmoins, pour comprendre l'efficacité de certaines stratégies, nous stockons d'autres métriques. Voici les différents éléments que l'on regarde avec leur but.

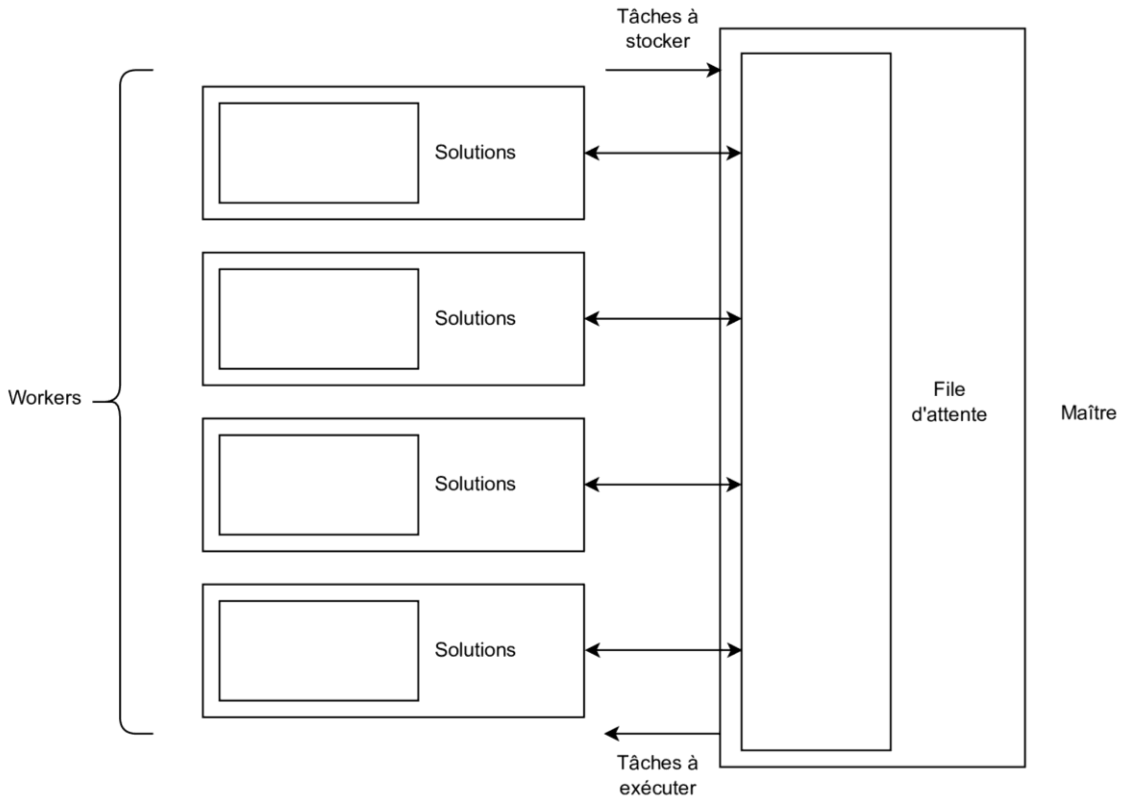
Nom	Description	Utilité
Temps de travail	Durée totale de travail d'un worker	Vise à maintenir un temps de travail homogène entre tous les workers.
Temps d'attente	Durée pendant laquelle un worker reste inactif	Vise à minimiser cette durée pour améliorer l'efficacité globale.
Temps de calcul	Durée nécessaire pour exécuter chaque tâche	Vise à équilibrer cette durée entre les différentes tâches.
Tâches effectuées	Nombre total de tâches accomplies par chaque worker	Cherche à répartir équitablement le nombre de tâches si leur durée est uniforme.

Mémoire utilisée	Quantité de mémoire consommée par chaque worker	Peut-être pré-allouée uniformément pour tous les workers, si nécessaire.
Solutions trouvées	Nombre de solutions identifiées par chaque worker	Sert à exclure les branches de recherche peu prometteuses.
Temps de récupération	Temps total incluant l'attente pour recevoir une nouvelle tâche	Critique pour optimiser la répartition dynamique des tâches.
Position des solutions	Emplacement des solutions dans l'arbre d'exploration	Permet de voir si les branches sont équilibrées

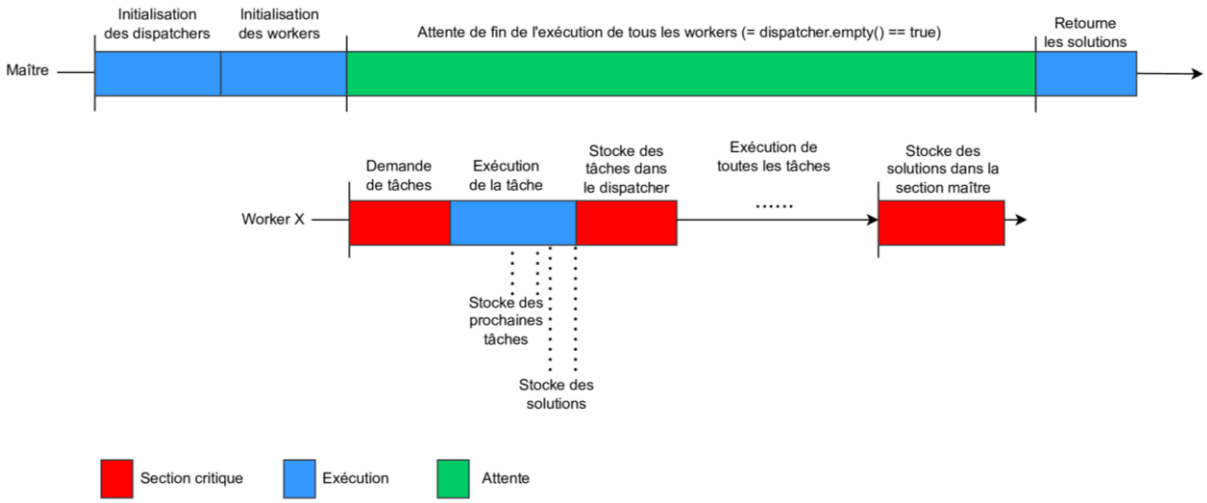
Les lignes avec un fond bleue signifient que ces métriques ne sont récoltées pour aucune des stratégies.

D. Stratégie par exécution de tâches parallèles

Cette première stratégie présentée est la plus simple et la plus intuitif. L'idée est de partir de l'exécution séquentielle est de parallélisé l'exécution de tâches. Pour cela, on définit un ensemble de Workers partageant tous une même instance de Dispatcher. Enfin quand on accède au Dispatcher on entre en section critique pour éviter la concurrence. Voici le schéma de la représentation mémoire.



Pour avoir une meilleure compréhension théorique des avantages et désavantages de notre stratégie, nous avons réalisé le schéma ci-dessous mettant en avant les zones critiques durant l'exécution du programme.

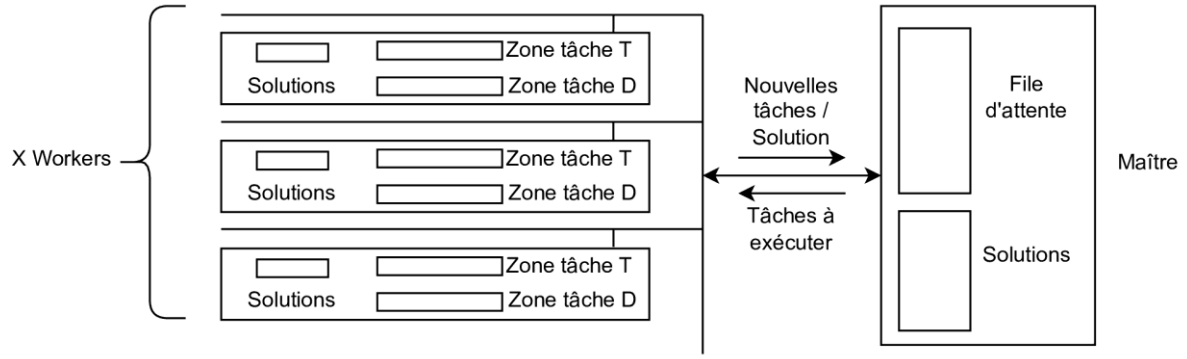


Cette méthode, bien que simple, n'est pas efficace puisqu'il y a un important goulot d'étranglement au niveau des accès au Dispatcher. Cela a pour conséquence que notre stratégie ne va pas bien scaler en fonction du nombre de Workers. Une piste pour réduire ce goulot serait de faire en sorte que le Dispatcher possède, par exemple, une file d'attente pour un groupe de Workers. Si cette file est vide pour effectuer l'opération « get », alors on regarde sur la file voisine. Cette piste n'a pas été implémentée et testée.

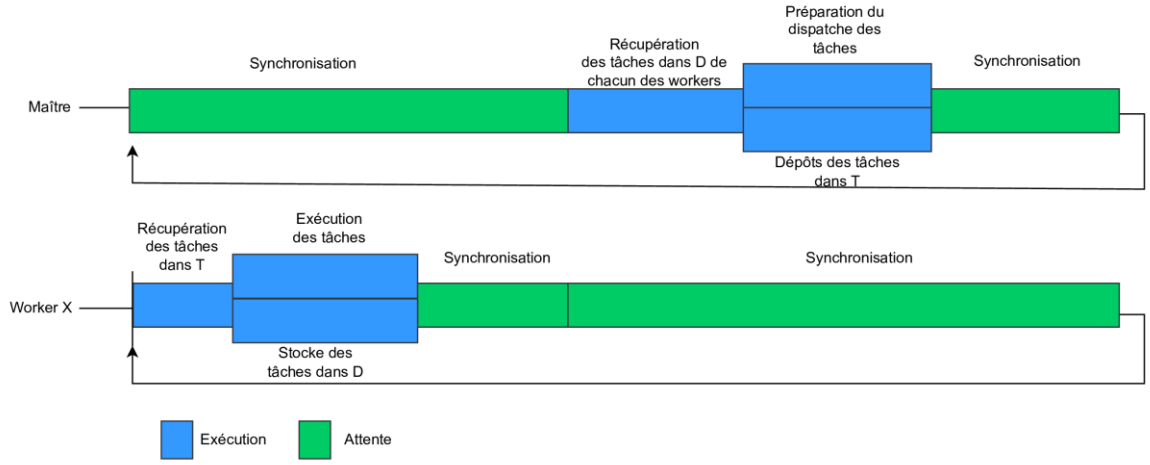
Pour l'implémentation de cette solution, il est nécessaire d'avoir un mécanisme d'arrêt pour s'assurer que tous les Workers s'arrêtent quand chacun n'a plus de tâches dans sa zone T. Pour cela, quand un Worker n'a plus de tâches, il demande l'arrêt au Dispatcher. Celui-ci incrémente un compteur des Workers en attente et les place en attente active. Si le compteur atteint le nombre total de Workers, alors ceux-ci peuvent s'arrêter. En revanche, si la file n'est plus vide, alors les Workers quittent leur attente et commencent une nouvelle itération de la boucle.

E. Stratégie par exécution de tâches synchronisées

Cette stratégie est la seconde mise en place. Durant chaque itération, les workers possèdent un ensemble de tâches qu'ils doivent résoudre, se trouvant dans la zone Tâche. À chaque fois qu'une tâche est résolue, une solution ou plusieurs tâches sont trouvées, celles-ci sont respectivement stockées dans un tableau Solution et la zone de Dépôt D. À la fin de l'itération, quand toutes les tâches dans la zone T ont été exécutées, il y a synchronisation. Puis le Maître récupère toutes les tâches dans la zone D de tous les workers pour les stocker dans sa file d'attente. Ensuite, toutes les tâches sont redistribuées dans la zone T de chaque worker de manière équitable. La boucle principale se termine à la fin d'une itération, quand toutes les zones D sont vides. Enfin, les solutions peuvent être récupérées à chaque itération ou à la fin de la boucle principale. Par simplicité, nous avons opté pour la première solution. Voici le schéma représentant la mémoire durant la boucle principale.



Nous avons réalisé le schéma ci-dessous mettant en avant les zones critiques durant une itération de la boucle.

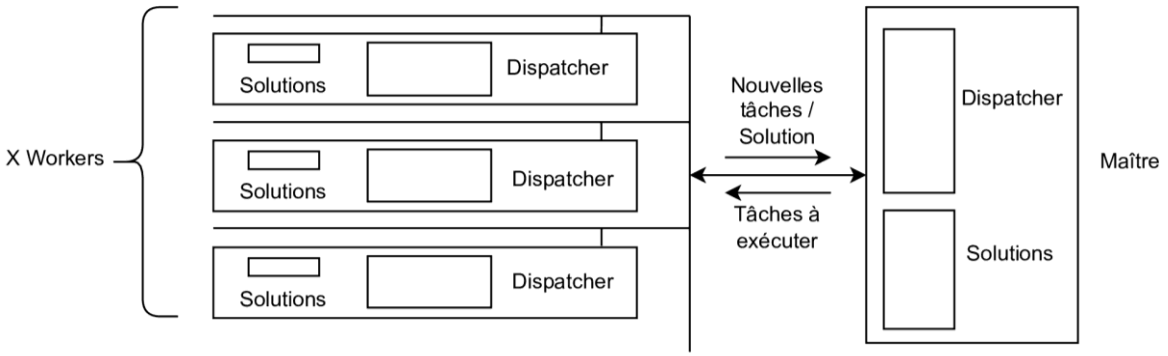


Comme on peut le voir, le grand désavantage de cette stratégie est la synchronisation nécessaire à chaque fin d'itération. Cela signifie que, dans le pire des cas, on peut avoir seulement un worker qui travaille pendant que les autres l'attendent. De plus, à chaque fin d'itération, le maître est le seul à s'exécuter, ne profitant donc pas du parallélisme. Néanmoins, comme cette étape est rapide, on peut la considérer comme négligeable. Le principal avantage de cette stratégie est le fait que l'on peut avoir une utilisation mémoire qui correspond à ce qui est réellement utilisé. Il est essentiel de veiller à allouer une quantité suffisante de mémoire dans les zones de dépôt et de tâches afin d'éviter les goulots d'étranglement entre les workers lors de la réallocation de ces zones quand on travaille avec des threads.

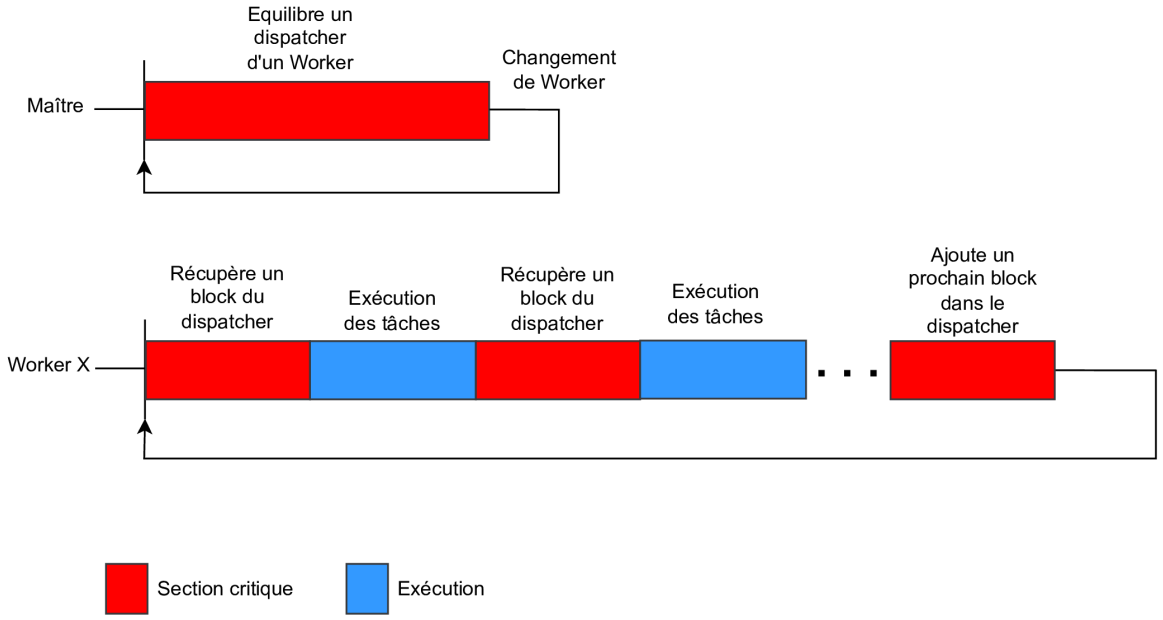
F. Stratégie par exécution de tâches parallèles régularisées

Nous enchaînons désormais sur notre stratégie suivante, que nous n'avons pas implémentée mais seulement abordée d'un point de vue théorique. Celle-ci repose sur l'idée de vouloir réduire la synchronisation et de ne pas avoir accès à des sections critiques qui engendrent des goulots d'étranglement importants. Pour cela, on a les Workers qui

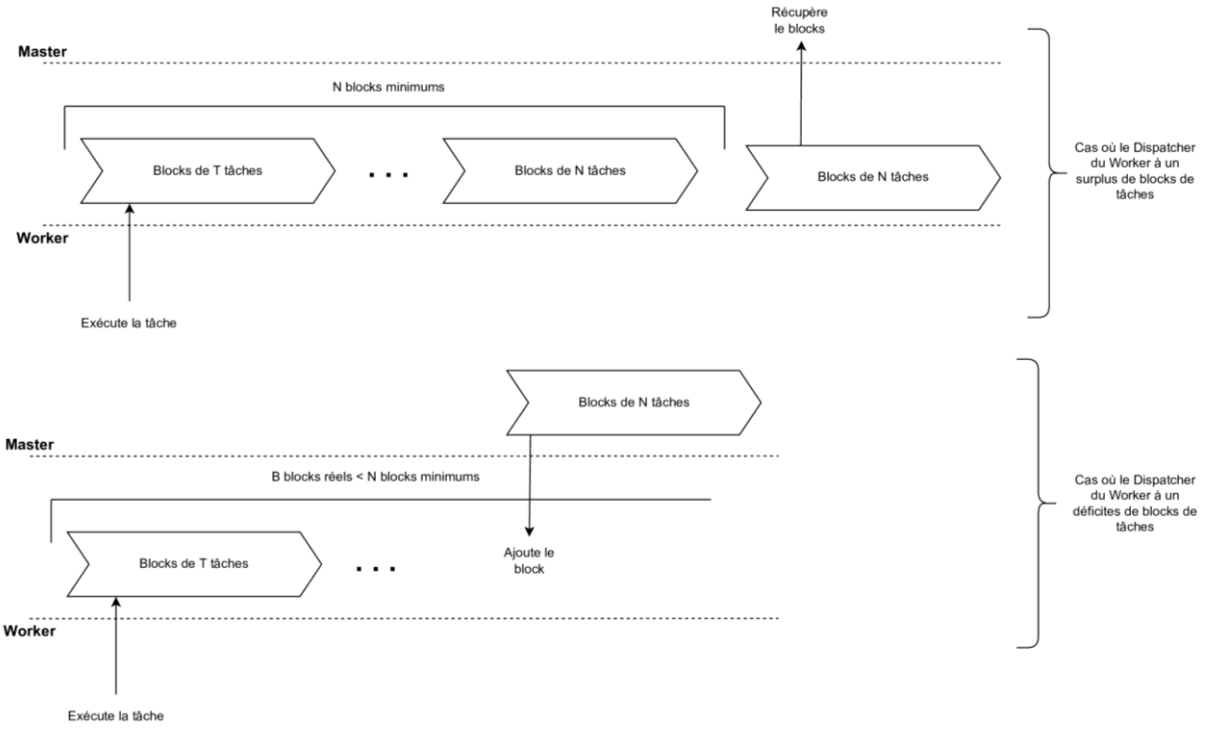
fonctionnent de la même façon que dans la stratégie de tâches parallélisées, où ils exécutent les tâches jusqu'à ce qu'il n'y en ait plus. En revanche, ils ont leur propre Dispatcher local où ils ajoutent ou retirent des tâches uniquement dans cet objet local. La différence est que le Worker maître n'est plus passif mais actif, en récupérant des tâches si un Worker en a trop et en donnant si un n'en a pas assez. Cela permet, durant l'exécution, de rééquilibrer le nombre de tâches entre Workers sans faire une étape de synchronisation et de rééquilibrage. Voici le schéma représentant la mémoire.



Voici le schéma mettant en évidence les zones critiques dans la boucle principal de l'algorithme.



Comme vous pouvez le voir, les accès à une zone critique sont récurrents. Pour réduire les accès du Worker, le Dispatcher stocke plutôt un bloc de tâches que le Worker va récupérer. Par ailleurs, le Worker stocke ses tâches trouvées dans un bloc local qu'il va ensuite ajouter à son Dispatcher une fois plein. Voici un schéma sur la structure du Dispatcher.

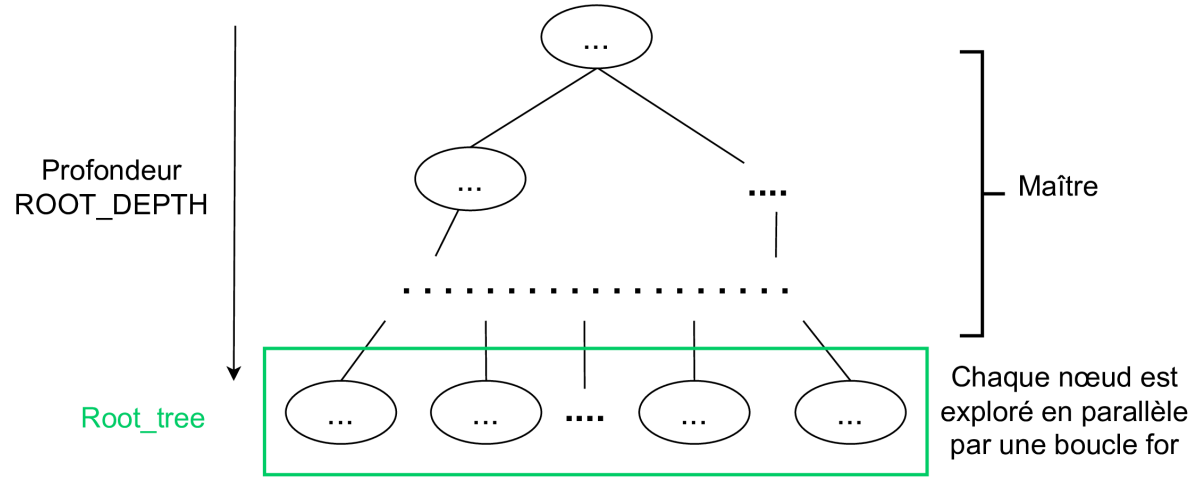


Pour un bon fonctionnement, il faut s'assurer d'avoir un surplus de tâches en stock afin de permettre un équilibrage au cours du temps. Cette stratégie semble donc intéressante pour ne pas avoir de mémoire allouée pour le programme mais qui n'est pas utilisée. De plus, après un bon équilibrage entre les différentes variables du programme, on peut avoir une exécution qui n'est pas diminuée par la concurrence pour des accès critiques. Enfin, un désavantage par rapport aux autres méthodes est que, lorsqu'on a un maître actif, cela oblige à allouer un thread ou un processus pour son exécution. Alors que, pour les autres méthodes, un thread peut très bien être le Maître et un Worker. L'importance de ce désavantage dépend de si notre machine possède peu ou beaucoup de threads.

G. Stratégie de travail hybride statique

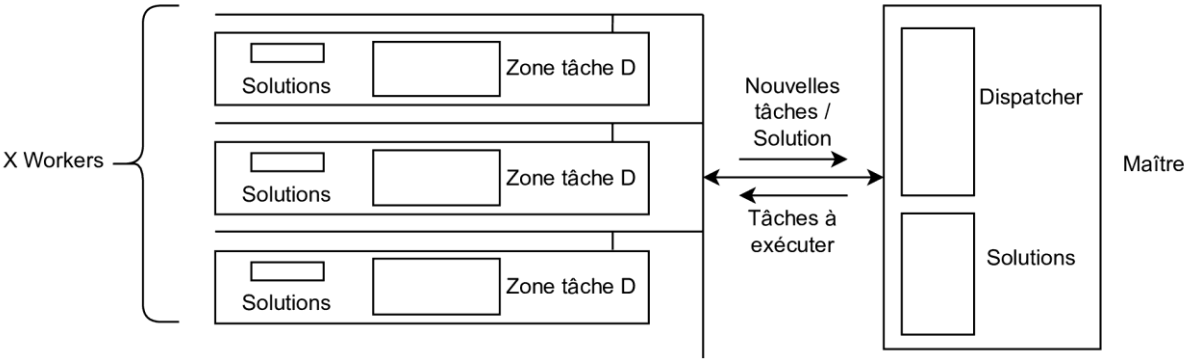
Nous allons présenter la dernière stratégie à laquelle nous avons pensé. Celle-ci s'appuie essentiellement sur le fait que la résolution du problème de Langford peut être représentée sous forme d'arbre. De plus, nous utilisons notre expérience cumulée des autres stratégies pour en tirer le meilleur. Ainsi, nous définissons deux grandes étapes dans notre stratégie.

La première étape est l'exploration de l'arbre uniquement par le maître jusqu'à une profondeur « ROOT_DEPTH » définie par l'utilisateur. À la fin, nous obtenons un ensemble de nœuds qui vont devenir des racines et que l'on va explorer en parallèle. Cela permet d'avoir un ensemble d'arbres qui sont individuellement petits et dont leur exploration parallèle ne nécessite pas de coûts mémoire importants, à la différence des autres techniques. Voici un schéma récapitulant ce paragraphe :

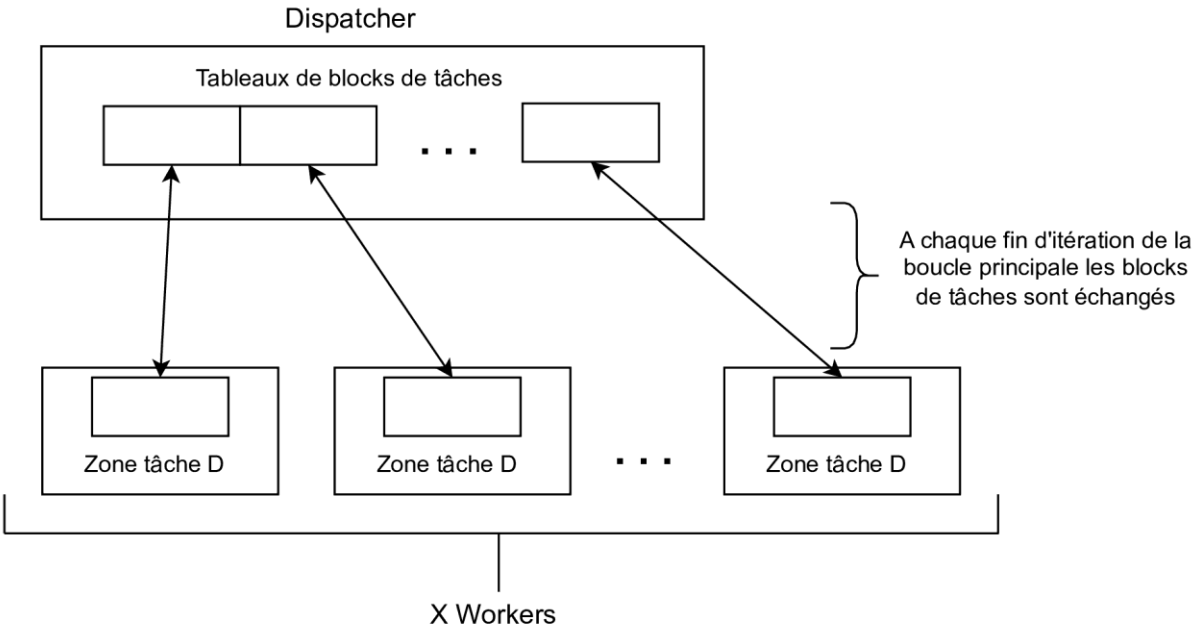


Ensuite, chaque Root_tree est exploré en parallèle. Nous pouvons allouer un Worker pour l'exploration d'un Root_tree ou bien un ensemble de Workers, auquel cas on a une parallélisation imbriquée. Pour cette deuxième parallélisation, nous pouvons utiliser l'une des autres stratégies vues précédemment ou une nouvelle méthode que nous allons proposer.

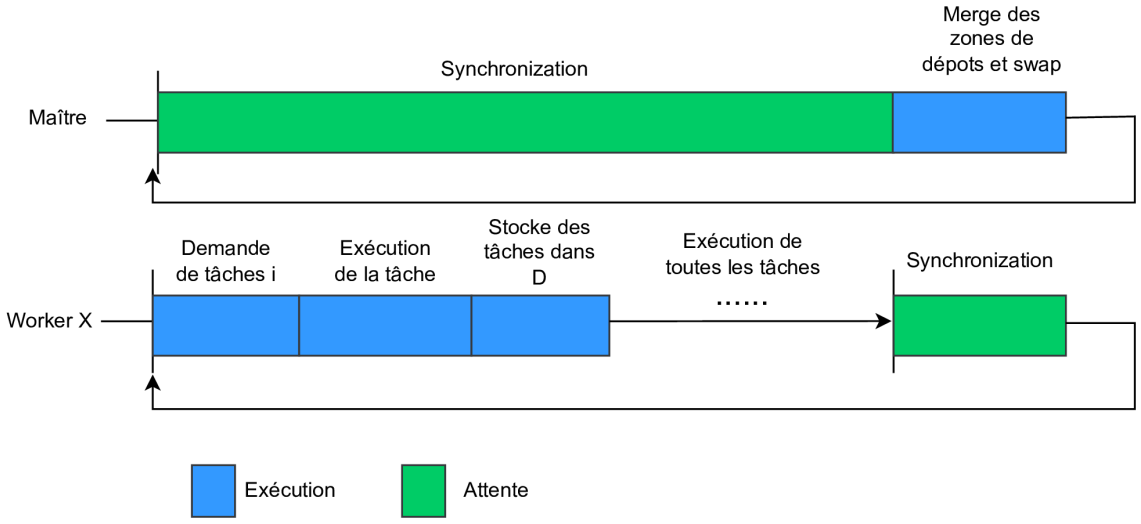
Notre stratégie d'exploration d'un Root_tree repose sur la même base de structure que celle que nous utilisons jusqu'à maintenant. Voici sa représentation ci-dessous :



Pour avoir un partage fluide, nous synchronisons avec un ensemble de tâches qui doivent être réalisées dans le dispatcher. Ensuite, celles-ci sont réalisées par les workers et les nouvelles tâches trouvées sont stockées dans la zone D. L'innovation ici repose sur la structure du dispatcher, qui ne distribue pas les tâches à la demande avec une file d'attente, mais elles sont plutôt distribuées en avance. En effet, nous connaissons le nombre de tâches et nous permettons un accès random en O(1), ce qui permet à chaque Worker d'avoir les tâches à l'aide d'une boucle for. Pour rappel, une boucle for permet une parallélisation efficace, en particulier dans un environnement local multithread. Ceci peut être réalisé grâce à la structure du dispatcher, qui possède un tableau de zone de tâches D, donc un ensemble de blocs de tâches. Ensuite, à chaque fin d'itération de la boucle principale, les Workers échangent leur tableau avec leur équivalent dans le Dispatcher ; cette opération d'échange est effectuée en O(1). Le schéma ci-dessous résume la structure du Dispatcher.



Enfin, le schéma suivant met en évidence les sections critiques durant la boucle principale pour l'exploration d'un Root_tree. Comme on peut le voir, cette stratégie permet, de toutes celles que l'on a vues, d'avoir le moins de synchronisations et d'entrées en section critique. Ceci permet d'affirmer qu'il s'agit de la meilleure, d'un point de vue théorique, que l'on propose.



D'un point de vue global sur notre stratégie, celle-ci s'intègre très bien à une implémentation aussi bien pour OpenMP que MPI. En effet, les éléments parallélisables sont une première boucle for pour traiter plusieurs Root_tree, puis une deuxième pour explorer un Root_tree. Comme les boucles for sont efficacement parallélisables dans OpenMP, cela fait que cette stratégie est efficace. En revanche, pour MPI, il semble bon d'uniquement paralléliser la première boucle for pour traiter plusieurs Root_tree en même temps. Cela est dû au fait que la deuxième boucle demanderait beaucoup d'envois de messages, car il y a beaucoup d'itérations, alors que la première reste faible en itérations. De plus, la quantité de données pour chaque itération est uniquement un objet Task, qui est très petit. Si nous

sommes dans un environnement avec plusieurs machines différentes, le meilleur choix est d'utiliser MPI pour partager les Root_tree entre chaque machine, puis d'utiliser OpenMP pour les explorer.

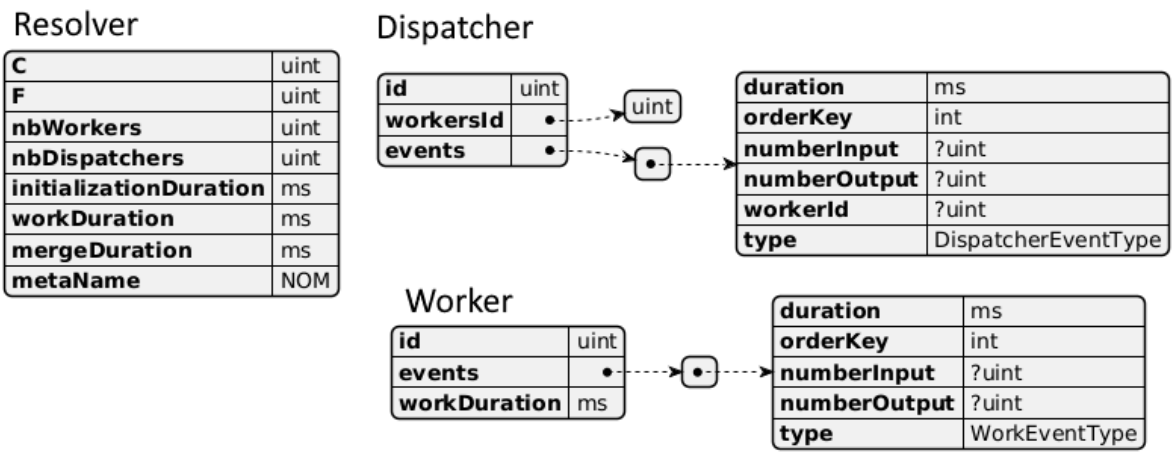
H. Modélisation

Nous faisons le premier choix de modéliser notre application à partir de la programmation objet. Cela a pour bénéfice de réfléchir en avance à nos choix et de s'assurer que ceux-ci soient cohérents et robustes. Nous avons pour but d'avoir un seul fichier main et de simplement changer l'objet représentant notre stratégie pour résoudre le problème de Langford. Dans ce cadre, nous définissons donc trois objets principaux :

- Resolver : Le principal objet à instancier qui prend en paramètre les conditions du problème (COUNT et FREQUENCY) de Langford et le résoud.
- Worker : Cette objet s'occupe de résoudre les tâches, il y a une instance dédiée pour chaque thread (OpenMP) ou processus (MPI).
- Dispatcher : Il stocke les tâches à exécuter permettant au moins deux opérations « get » et « add ». Il peut être partagé entre plusieurs workers en fonction de la stratégie.

Naturellement, on définirait bien une classe abstraite puis une classe concrète pour chaque stratégie. Mais comme notre objectif est la rapidité d'exécution, nous ne faisons pas cela, car à chaque appel de fonction de ces objets, il est nécessaire de passer par la virtual table. Par conséquent, nous définissons les classes concrètes par des templates devant implémenter une interface. Dans les faits, passer par une virtual table pourrait être négligeable, mais nous passons par les templates notamment pour tester de nouvelles choses et élargir notre expérience générale. Par ailleurs, le template reste particulièrement utile pour définir notre structure de Langford, puisque l'on peut lui mettre un attribut de tableau statique et non dynamique.

Nous modélisons ainsi notre application en passant par le langage textuel de PlantUML pour faire l'UML. Ensuite, nous ajoutons un ensemble d'objets de type traqueur, dont les fonctions sont appelées à des moments donnés pour y enregistrer un évènement. À la fin du programme, leurs informations stockées sont enregistrées dans un fichier JSON, puis peuvent être post-traitées. Ces objets répondent essentiellement au cycle de vie des 3 objets principaux énoncés précédemment. Voici les données qui sont stockées ; nous n'allons pas toutes les expliquer, puisque certaines sont uniquement utilisées pour certaines stratégies.



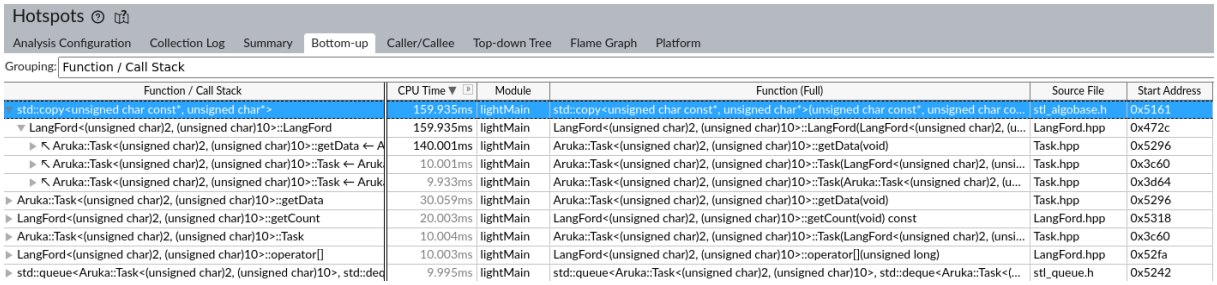
Enfin, stocker les informations au cours du programme a un coût. Par conséquent, il y a des classes concrètes de traqueurs dont les fonctions qu'elles implémentent sont vides, permettant ainsi de réaliser l'algorithme principal sans ralentissement ni coût supplémentaire en mémoire.

IV. Résultats

Les résultats présentés ont été obtenus avec une machine équipée d'un processeur Intel(R) Core(TM) i9-14900K possédant 16 cœurs avec 2 threads logiques par cœur, pour un total de 32 threads possibles. Initialement sous Windows, nous avons exécuté les programmes dans WSL 2 avec Ubuntu. Par ailleurs, il s'agit d'une machine personnelle, ce qui fait que d'autres processus fonctionnent, et par conséquent, quand on définit un grand nombre de processus ou threads, l'accélération est par défaut biaisée.

A. Optimisation

Nous avons utilisé l'outil VTune Profiler d'Intel pour voir les zones qui prenaient du temps et ainsi optimiser l'algorithme. Par itération, nous avons pu améliorer ses performances et l'accélérer de 15 % pour l'algorithme séquentiel. Nous n'allons pas vous montrer en détail cette étape d'optimisation, mais nous vous montrons ci-dessous un exemple de résultat indiquant les fonctions les plus consommatrices en temps CPU.

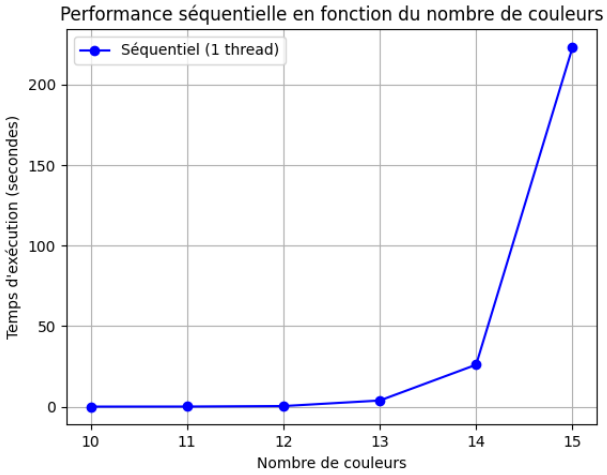


Nous pouvons comprendre, dans ce cas, que la copie des données est un des principaux enjeux et montre l'importance de la gestion mémoire. Enfin, VTune Profiler offre

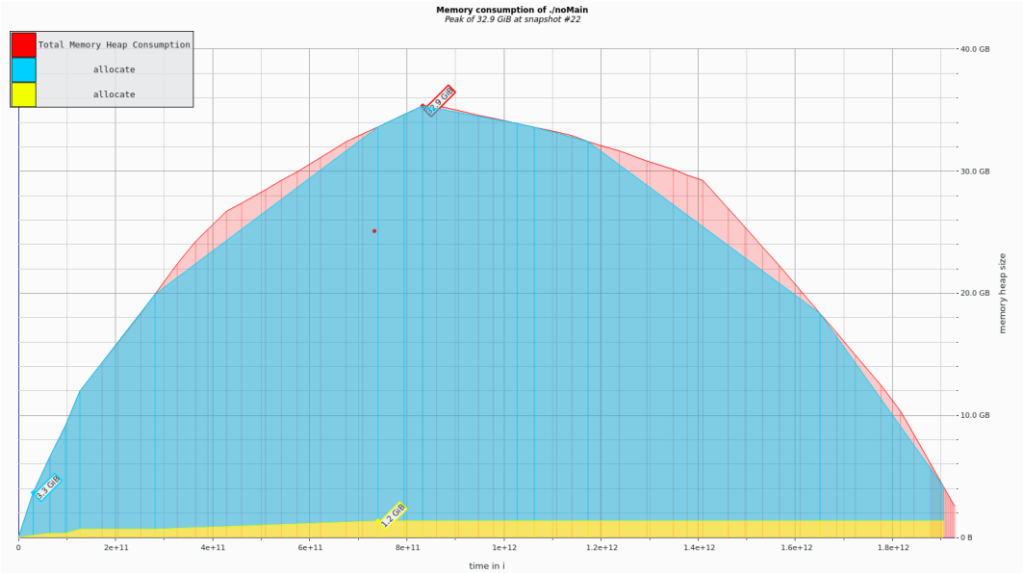
des informations plus précises sur d'autres panels, notamment concernant la consommation mémoire ainsi que la bonne gestion des threads.

B. Algorithme séquentiel

Nous allons utiliser l'algorithme séquentiel comme référence pour le comparer à nos différentes stratégies de parallélisation afin de pouvoir juger de leur efficacité. Bien que nous n'allions pas rentrer dans les détails de la raison pour laquelle nous avons obtenu ces résultats, il est néanmoins intéressant de comprendre les enjeux. En premier lieu, voici le temps nécessaire pour résoudre le problème de Langford en fonction de la fréquence.



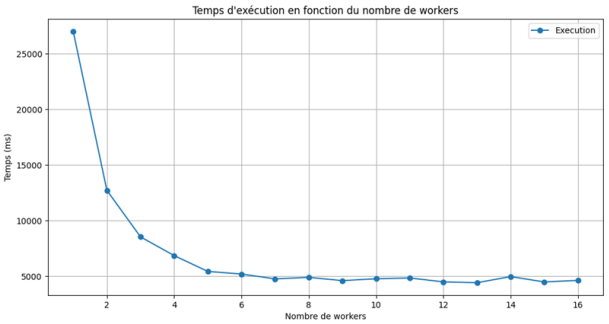
Nous pouvons voir que le problème n'est pas résolu instantanément à partir d'une fréquence de 13. Cette courbe démontre bien qu'il s'agit d'un problème complexe. Par ailleurs, pour tester l'efficacité de nos algorithmes parallélisés, il convient de les tester pour une fréquence de 14 ou 15. Pour comprendre les enjeux liés à la mémoire, voici le rapport effectué par Valgrind couplé avec Massif pour une fréquence de 14.

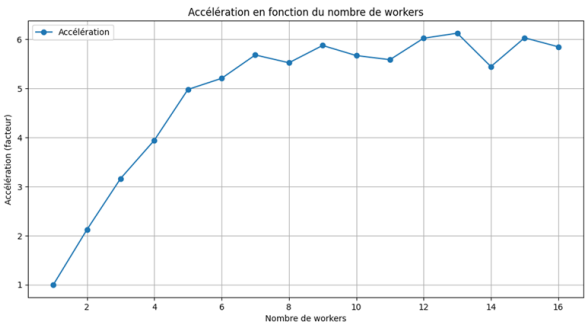


Comme on peut le voir, on utilise au maximum 33 Go de mémoire, ce qui démontre un enjeu certain par rapport à l'utilisation de la mémoire.

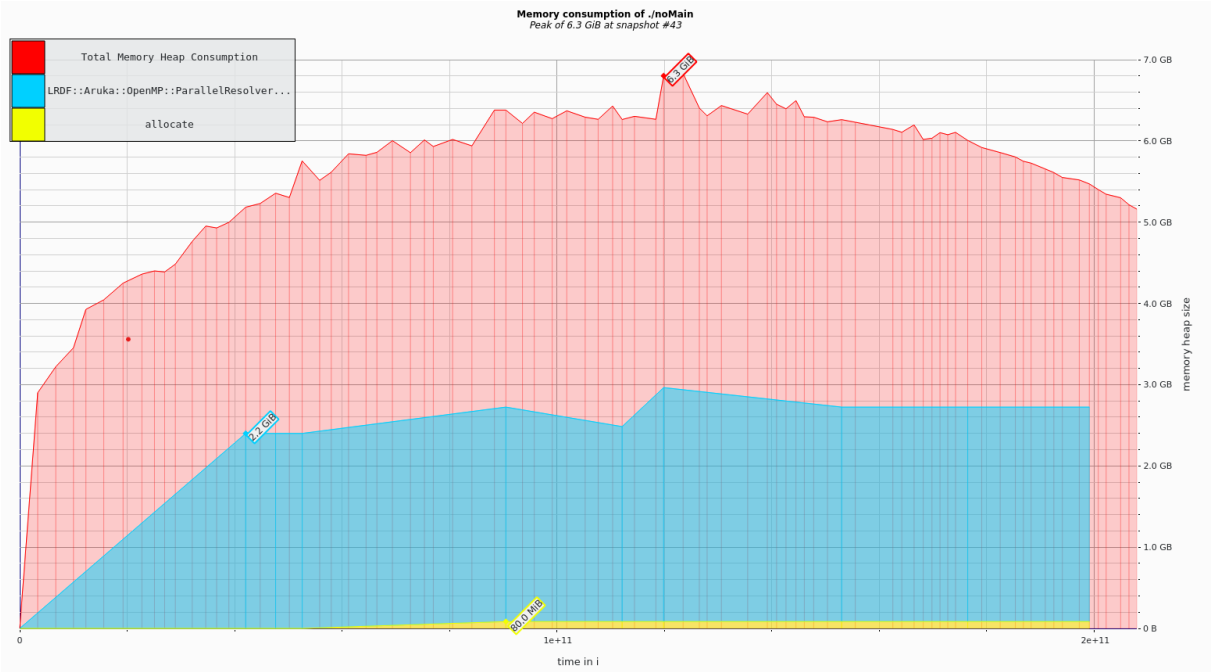
C. Algorithme par exécution de tâches parallèles avec OpenMP

Voici les résultats concernant la vitesse de notre algorithme pour une fréquence de 14.

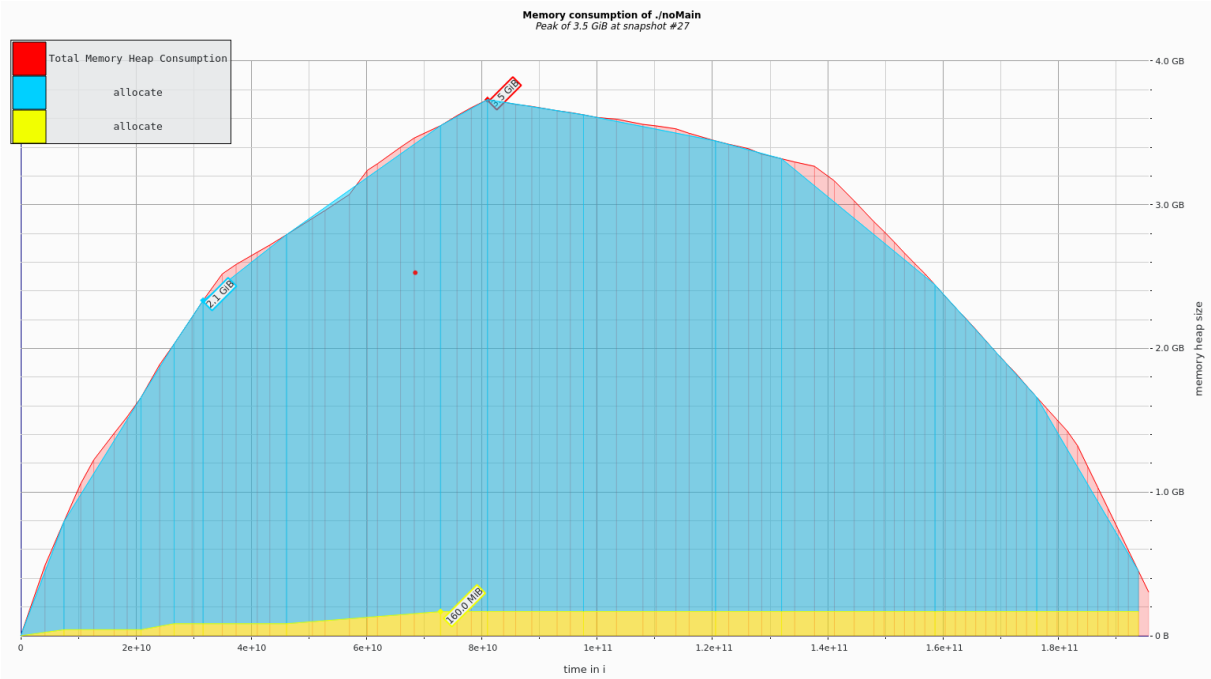




On peut voir que l'accélération s'estompe rapidement en fonction du nombre de threads. On peut supposer que cela peut être dû au fait que le problème, se résolvant en 5 secondes, ne peut être réduit davantage, mais si on augmente la fréquence, on peut observer des résultats similaires. À une fréquence de 15, nous ne présentons pas le résultat, car il peut s'agir d'un manque de mémoire, puisque la consommation mémoire est doublée. En effet, voici la consommation mémoire de cette exécution.

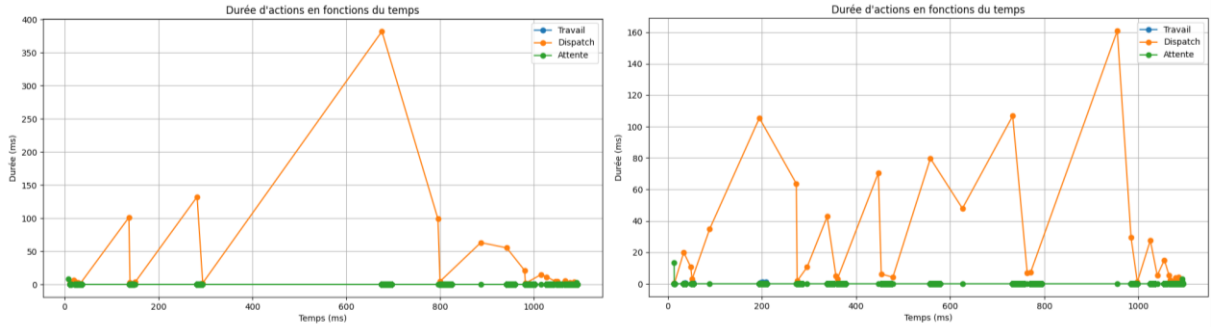


Voici la consommation pour la version séquentielle.

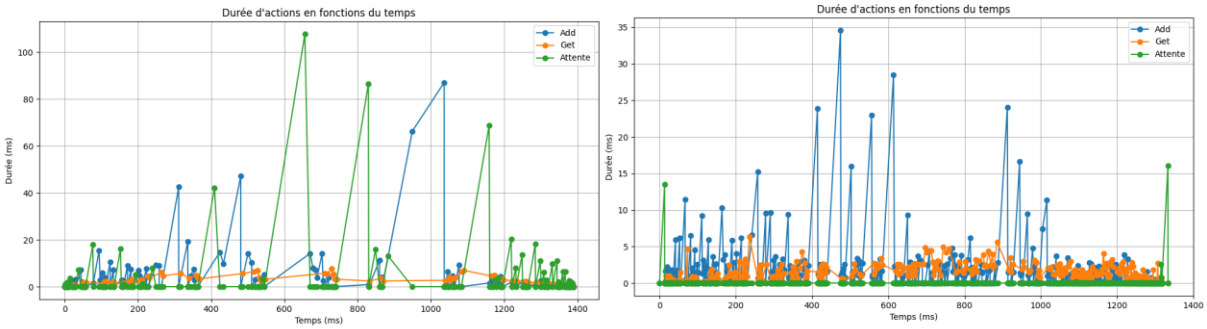


On peut voir que celle en parallèle est plus aplatie au sommet, ce qui peut être dû au parallélisme et au fait que les threads atteignent plus rapidement leur consommation maximale de mémoire.

La faible accélération peut cependant s'expliquer par l'accès concurrent au Dispatcher, comme le montrent ces graphiques, qui illustrent respectivement le cas d'un seul Worker comparé à un total de 7 et 32 Workers.



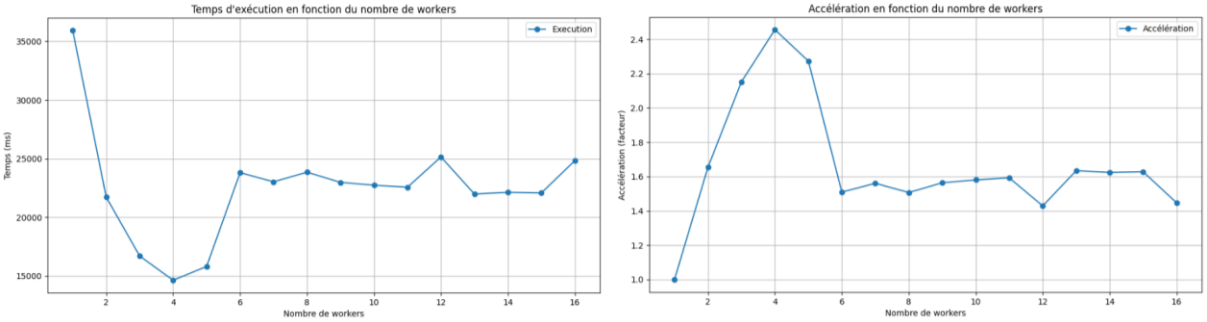
Ceci est catastrophique, les Workers passent le plus de temps dans des actions (« get » et « add ») liées au Dispatcher. Par ailleurs, ces graphiques du Dispatcher pour un total respectif de 7 et 32 Workers montrent qu'ils sont saturés.



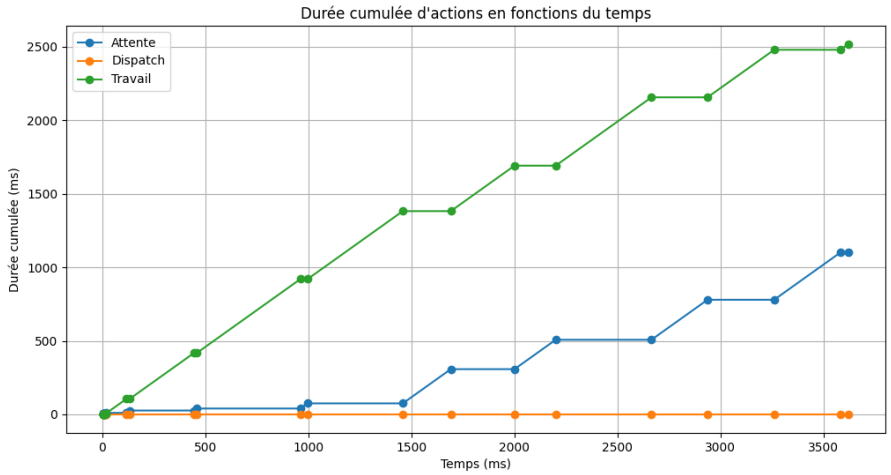
Le Dispatcher devrait passer le plus de temps en attente, alors que là, il passe le plus de son temps à ajouter des tâches. Un axe d'amélioration serait que les Workers ajoutent plutôt de gros blocs de tâches pour réduire le nombre d'appels. Néanmoins, cette solution repousse seulement le problème.

D. Algorithme par exécution de tâches synchronisées avec OpenMP

Cette stratégie est un échec : la vitesse n'y est pas, et ajouter plus de threads ralentit le programme, comme le montre ce graphique.



Cela est dû à la récurrence de points de synchronisation : plus on augmente le nombre total de Workers, plus il y a de chances qu'il faille attendre un Worker qui n'ait pas fini. Cela peut être observé par ce graphique, qui montre que le Worker, pour 32 threads, attend plus qu'il ne travaille.

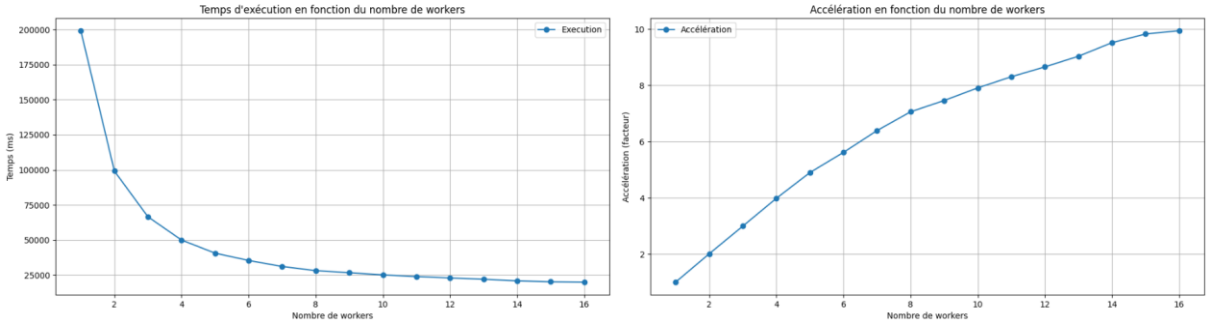


Le seul point positif est qu'il n'y a pas de consommation mémoire supérieure par rapport à l'algorithme séquentiel. En effet, tous deux ont un pic d'utilisation vers les 3,5 Go.

E. Algorithme par travail hybride statique avec OpenMP

Le premier résultat observable pour cette stratégie est que son exécution ne nécessite que quelques Go de mémoire, même pour une fréquence élevée comme 15. Cela permet donc de pouvoir l'exécuter sur toutes les fréquences sans grosses contraintes de mémoire. Cela est dû au fait que l'on explore l'arbre en deux étapes, avec la première donnant les Root_tree, puis la deuxième où on les explore. Cela est également dû au fait que, quand on explore un arbre, on l'explore en largeur dans notre algorithme, ce qui donne la contrainte que, quand on arrive aux feuilles d'un arbre, on doit toutes les stocker en mémoire. Tandis que, grâce à notre exploration en deux étapes, on découpe l'arbre principal en plus petits arbres.

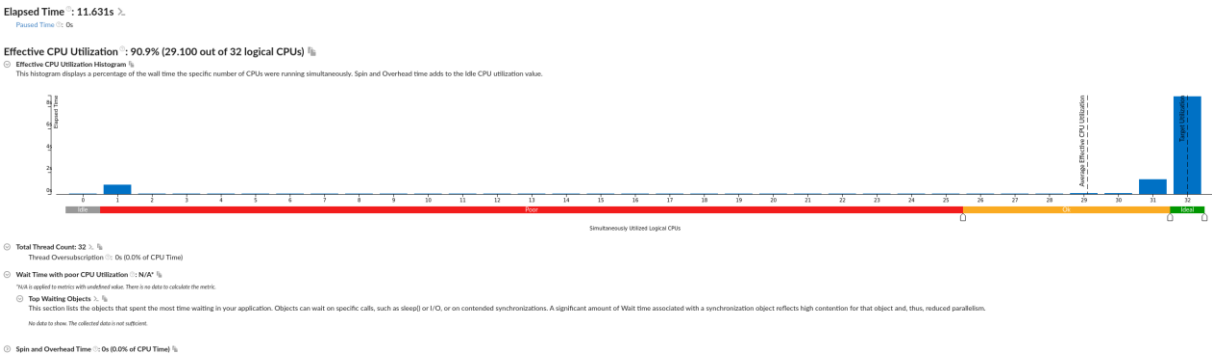
Grâce à une meilleure économie de la mémoire, nous pouvons présenter l'accélération pour une fréquence de 15.



Comme on peut le voir, notre stratégie permet de baisser le temps d'exécution. Mais on se retrouve tout de même avec une accélération qui tend à converger en fonction du nombre de Workers. En se basant sur nos données et notre expérience, nous n'arrivons pas à expliquer cette convergence. Enfin, comme énoncé précédemment, l'exploration d'un Root_tree peut être faite en parallèle en imposant en dur dans le code un nombre de Workers pour chacune de ces explorations. Il s'agit d'une fonctionnalité qui nous semblait pouvoir améliorer le tout, mais il s'avère que l'algorithme est plus rapide sans une exploration parallèle

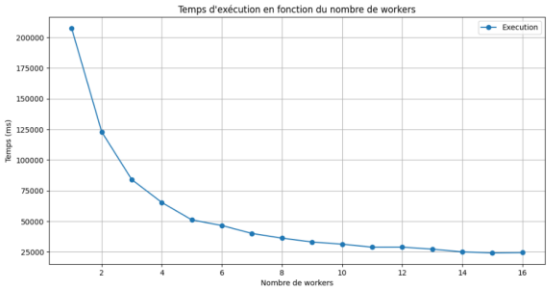
d'un Root_tree. On pensait que cela aurait permis d'avoir un meilleur équilibrage dans le traitement parallèle des différents Root_tree.

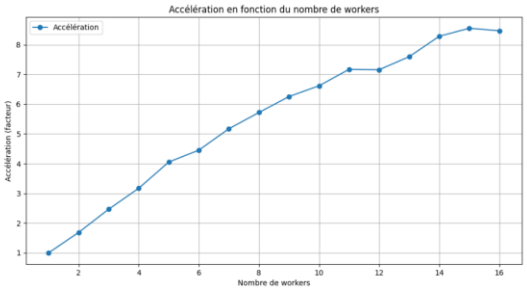
Enfin, pour traiter la boucle for, on peut opter pour un partage dynamique ou statique. Il s'avère, après test, que les deux ont des performances similaires, indiquant que tantôt il serait bon de faire du dynamique, et que quelquefois du statique serait mieux. Cela est d'ailleurs confirmé par VTune Profiler, donnant une utilisation d'environ 90 % des 32 cœurs logiques pour les deux options. Voici un exemple des résultats retournés par le profiler.



F. Algorithme par travail hybride statique avec MPI

L'implémentation MPI de notre stratégie par travail hybride statique a des résultats similaires à celle implémentée par OpenMP. Ceci était attendu, d'autant plus qu'ils sont légèrement inférieurs, ce qui est logique par rapport à la lourdeur de communication imposée entre les processus. Voici les schémas de rapidité d'exécution obtenus pour une fréquence de 15.





Par simplification d'implémentation, le partage des Root_tree se fait par blocs plutôt qu'un partage cyclique avec OpenMP. Cela se voit puisque le processus maître enregistre le début de son étape « merge » (réduction) quand il a terminé de traiter ses tâches. Pour une fréquence de 15, cela a pour conséquence d'avoir l'étape de la boucle principale durant 13 secondes contre 12 secondes pour la réduction. Rassembler les solutions ne prend évidemment pas 12 secondes. Cela est simplement dû au fait que le maître est contraint d'attendre les autres processus, démontrant un mauvais partage des Root_tree. Il faudrait donc faire un partage cyclique ou éventuellement un partage dynamique et non statique.

Pour ce qui est de la consommation mémoire, nous obtenons des résultats également similaires, là aussi elle n'est pas élevée.

V. Retour d'expérience et analyse critique

Dans cette section, nous allons proposer un retour constructif sur les différents aspects sur lesquels nous avons travaillé.

A. Modélisation

Nous sommes globalement satisfaits de l'approche de modélisation, puisqu'elle est construite, réfléchie et cohérente. En revanche, son problème majeur est la modélisation UML, puisque celle-ci est assez lourde. En ouvrant le fichier texte, on ne comprend pas d'un coup d'œil la modélisation. Elle reste compréhensible uniquement pour ceux qui l'ont faite. De plus, la modélisation a nécessité la création du fichier helper.hpp, qui simplifie les interactions avec les templates, mais dont la compréhension reste complexe sans une étude approfondie.

Pour un prochain projet, il faudra réfléchir à une approche peut-être plus simple. Néanmoins, à l'heure actuelle, si nous devions refaire, nous ne saurions pas comment faire mieux. Par conséquent, il faut que l'on reste ouvert à l'apprentissage d'autres outils et techniques.

Par ailleurs, le choix réalisé de collecter des données et de les post-traiter en Python pour en sortir des graphiques est intéressant. Mais, nous pensons qu'il manquait un fichier général Excel que l'on viendrait modifier pour avoir un ensemble des résultats au même endroit. Ici, les différents résultats sont « éparpillés ». D'autre part, enregistrer des données directement dans le programme peut ralentir l'exécution de celui-ci ; à l'avenir, il serait plus sage d'apprendre à utiliser des outils spécialisés pour cela.

B. Différences entre OpenMP et MPI

Le choix entre OpenMP et MPI est une décision importante, et ce projet nous a permis d'en comprendre les enjeux. En effet, d'un point de vue personnel, nous trouvons que choisir MPI rend la programmation plus simple, puisque les Workers ne partagent pas la même mémoire, ce qui permet d'allouer de la mémoire dynamiquement sans créer de goulots d'étranglement. Par ailleurs, si nous avons besoin d'avoir beaucoup de communication entre les Workers, alors OpenMP est sans doute le meilleur choix grâce à sa légèreté.

C. La meilleure stratégie

Notre meilleure stratégie est sans aucun doute celle par travail hybride statique. Ses implémentations sont les plus rapides et les plus économes en mémoire. Néanmoins, sa création est le fruit d'autres stratégies qui nous ont permis d'élargir nos horizons en testant et parfois en échouant. Dans cette stratégie, il y a la mise en place du concept des Root_tree, qui pourrait être également intégré à l'algorithme séquentiel de référence, puisqu'il permet l'économie de mémoire sans ralentissement. Le seul changement serait le parcours de l'arbre.

Pour comprendre et analyser l'efficacité de nos stratégies, nous avons découvert VTune Profiler d'Intel, qui nous a permis d'améliorer nos algorithmes. Toutefois, nous ne l'avons pas utilisé à son plein potentiel faute de connaissances. Par conséquent, il faut que l'on étudie davantage ce logiciel pour pouvoir parfaire son utilisation.

VI. Conclusion

Ce projet nous a permis d'approfondir nos connaissances en programmation parallèle et en gestion de la mémoire. Grâce à une approche itérative combinant réflexion théorique et expérimentations pratiques, nous avons pu élaborer et comparer différentes stratégies de résolution du problème de Langford, chacune présentant ses avantages et ses limites. Parmi elles, la stratégie par travail hybride statique s'est avérée la plus performante, tant en termes de rapidité que d'économie de mémoire.

Nous avons également exploré les outils de profilage tels que VTune Profiler et Valgrind, qui se sont révélés essentiels pour comprendre les goulots d'étranglement et optimiser nos algorithmes. Bien que notre utilisation de ces outils soit encore perfectible, cette expérience nous a introduit vers une meilleure maîtrise des techniques d'optimisation des performances.

En somme, ce projet nous a offert une excellente opportunité d'apprendre, d'expérimenter et de repousser nos limites en termes de modélisation, d'optimisation et d'approche méthodologique. Les enseignements tirés nous serviront sans aucun doute dans nos futurs travaux et projets.