Rapport Projet de Programmation Système

1. Présentation

Ce projet à été intégralement réalisé par Lucas RODRIGUEZ et Yann PICKERN, sans aucune assistance extérieur d'autre groupes.

Nous avons implémenté des ordonnanceurs fonctionnant par work-sharing ou work-stealing avec différentes primitives de synchronisation que nous détaillerons dans les parties 2 et 3 de ce rapport. Nous avons 3 démonstrations de disponibles, dont une d'entre elles nécessite une librairie pour les illustrer.

De plus nous avons implémenté nos propres stacks et deque afin d'y stocker les tâches, par la création d'une structure pour encapsuler le pointeur vers la méthode passé à "sched_spawn" ainsi que ses arguments.

2. Les implémentations LIFO

La première implémentation LIFO est composée d'une variable de condition et d'un mutex protégeant l'ensemble des données partagées. La variable de condition rend oisifs les threads en attente de travail.
L'implémentation est simple mais présente un point de contention global avec l'augmentation des threads, un seul mutex protège l'ensemble des données.

La seconde implémentation et la plus simple utilise quant à elle seulement un spinlock construit à l'aide d'une valeur atomique. Lorsque les threads n'ont plus de travail ils s'endorment pendant une durée arbitraire, ici 2ms. Cette implémentation est très efficace et n'est pas beaucoup plus consommatrice en performance que les autres, comme nous le verrons dans la partie 8.

La figure 1 composée des résultats des 8 derniers threads sur un benchmark à 16 threads nous montre bien que l'ordonnanceur LIFO avec spinlock est plus efficace.
Comme évoqué précédemment, le mutex utilisé dans le premier ordonnanceur créé un point de contention avec l'évolution de la quantité de threads, qui n'est pas présent sur l'ordonnanceur spinlock dont les résultats se stabilisent à partir de 8 threads mais n'augmentent pas.

Ordonnanceur LIFO	**Ordonnanceur LIFO spinlock**
9 Threads: 0.135336	9 Threads: 0.121266
10 Threads: 0.134984	10 Threads: 0.119148
11 Threads: 0.136611	11 Threads: 0.118667
12 Threads: 0.135702	12 Threads: 0.119401
13 Threads: 0.138216	13 Threads: 0.116018
14 Threads: 0.138028	14 Threads: 0.116986
15 Threads: 0.138653	15 Threads: 0.117673
16 Threads: 0.142988	16 Threads: 0.116832

Figure 1: ordonnanceur LIFO `lifo_sched` et LIFO avec spinlock `lifo_sched_spin` sur 16 threads.

Une implémentation LIFO avec sémaphore a été tentée mais n'est pas fonctionnelle. Les threads quittent avant la complétion totale du travail et l'ordonnanceur se retrouve avec un seul thread fonctionnel.

3. Les implémentations work-stealing

La première implémentation est l'implémentation basique de cet ordonnanceur, elle nécessite une deque par thread ainsi qu'un mutex par deque. Dans cet ordonnanceur, les threads s'endorment pendant 1ms après un échec de vol de tâches.

En analysant la figure 2 des résultats et statistiques des 4 derniers threads, on remarque une diminution constante du temps d'exécution accompagné d'une croissance du nombre de vol ratés. Le vol de tâches reste une action plutôt rare, cependant elle pourrait poser problème sur des machines ayant encore plus de threads. Effectivement, lors d'un vol de tâche, le mutex de la deque est bloqué le temps de regarder et d'en prendre son contenu, bloquant le thread sur le travail qu'il a à faire.

13 Threads: 0.187703	Steal attempts succeeded: 124465 Steal attempts failed: 181 Tasks completed: 368417
14 Threads: 0.178064	Steal attempts succeeded: 112727 Steal attempts failed: 209 Tasks completed: 368428
15 Threads: 0.175285	Steal attempts succeeded: 109868 Steal attempts failed: 250 Tasks completed: 368423
16 Threads: 0.174812	Steal attempts succeeded: 103703 Steal attempts failed: 272 Tasks completed: 368433

Figure 2: ordonnanceur classique `stealing_sched` sur 16 threads.

Afin d'essayer de résoudre ce problème, deux solutions variant le temps d'attente en fonction des résultats du vol ont été implémentées.
L'une d'elle ne contient qu'une seule valeur d'attente partagée par l'ensemble des threads et protégée par un mutex. L'autre ordonnanceur a pour chaque thread sa propre variable, ne nécessitant pas d'être protégée.

En regardant les résultats des figures 3 et 4, on remarque clairement que les quantités de vols ratés sont significativement plus petites que celles de l'ordonnanceur classique. On remarque aussi que celui avec une seule variable partagée a environ deux fois moins d'échecs que celui avec plusieurs variables et que son temps d'exécution est plus rapide. La contrainte du mutex n'est donc pas assez impactante pour justifier plusieurs variables.
Finalement, on peut voir que les temps d'exécutions ne sont pas plus rapides que sur l'ordonnanceur classique.

13 Threads: 0.19677	Steal attempts succeeded: 120345 Steal attempts failed: 43 Tasks completed: 368420
14 Threads: 0.192715	Steal attempts succeeded: 118267 Steal attempts failed: 46 Tasks completed: 368428
15 Threads: 0.188102	Steal attempts succeeded: 113665 Steal attempts failed: 49 Tasks completed: 368423
16 Threads: 0.186267	Steal attempts succeeded: 107779 Steal attempts failed: 52 Tasks completed: 368431

Figure 3: ordonnanceur avec une variable partagée `stealing_sched_opt`.

13 Threads: 0.204092	Steal attempts succeeded: 125548 Steal attempts failed: 88 Tasks completed: 368418
14 Threads: 0.200361	Steal attempts succeeded: 116802 Steal attempts failed: 103 Tasks completed: 368429
15 Threads: 0.190247	Steal attempts succeeded: 109664 Steal attempts failed: 107 Tasks completed: 368428
16 Threads: 0.192223	Steal attempts succeeded: 110320 Steal attempts failed: 115 Tasks completed: 368430

Figure 4: ordonnanceur avec une variable par thread `stealing_sched_opt_multiple`.

Une dernière implémentation a été d'utiliser une variable de condition pour endormir les threads, similaire à l'implémentation de l'ordonnanceur LIFO.

On remarque sur la figure 5 que les tentatives de vol sont inférieures à celles de l'ordonnanceur classique, notamment car les threads ne sont réveillés que lorsque du travail a été remis dans la

deque. Ils sont toutefois plus élevés que les deux ordonnanceurs avec variation du temps d'attente, mais obtiennent des résultats plus efficaces.
Il s'agit de l'ordonnanceur par work-stealing le plus performant implémenté.

13 Threads: 0.180424	Steal attempts succeeded: 121887 Steal attempts failed: 117 Tasks completed: 368420
14 Threads: 0.174492	Steal attempts succeeded: 113106 Steal attempts failed: 156 Tasks completed: 368428
15 Threads: 0.171848	Steal attempts succeeded: 111630 Steal attempts failed: 171 Tasks completed: 368428
16 Threads: 0.170607	Steal attempts succeeded: 105807 Steal attempts failed: 196 Tasks completed: 368431

Figure 5: ordonnanceur avec une variable de condition `stealing_sched_cond`.

4. Work-stealing contre LIFO

Le code utilisé pour générer le graphique décrivant le temps d'exécution des différentes implémentations d'ordonnanceurs est le script python `benchmark/bench.py` qui pourra aussi régénérer les csv liés aux statistiques.

Dans ce projet nous avons été relativement surpris par l'efficacité du simple ordonnanceur LIFO. Néanmoins, ce graphique ainsi que les statistiques récoltées par notre benchmark nous permettent de nous rendre compte que la vitesse du simple LIFO ne s'améliore que très peu au delà de 8 threads et commence déjà à ralentir avec 16 threads (la machine la plus puissante à notre disposition possède un processeur avec 8 coeurs). Au contraire de l'ordonnanceur par work-stealing qui lui ne cesse de gagner en vitesse avec l'augmentation du nombre de threads.

Nous pouvons donc émettre l'hypothèse qu'avec un passage à l'échelle suffisant, un écart encore plus grand se creuserait entre l'ordonnanceur LIFO et work-stealing.

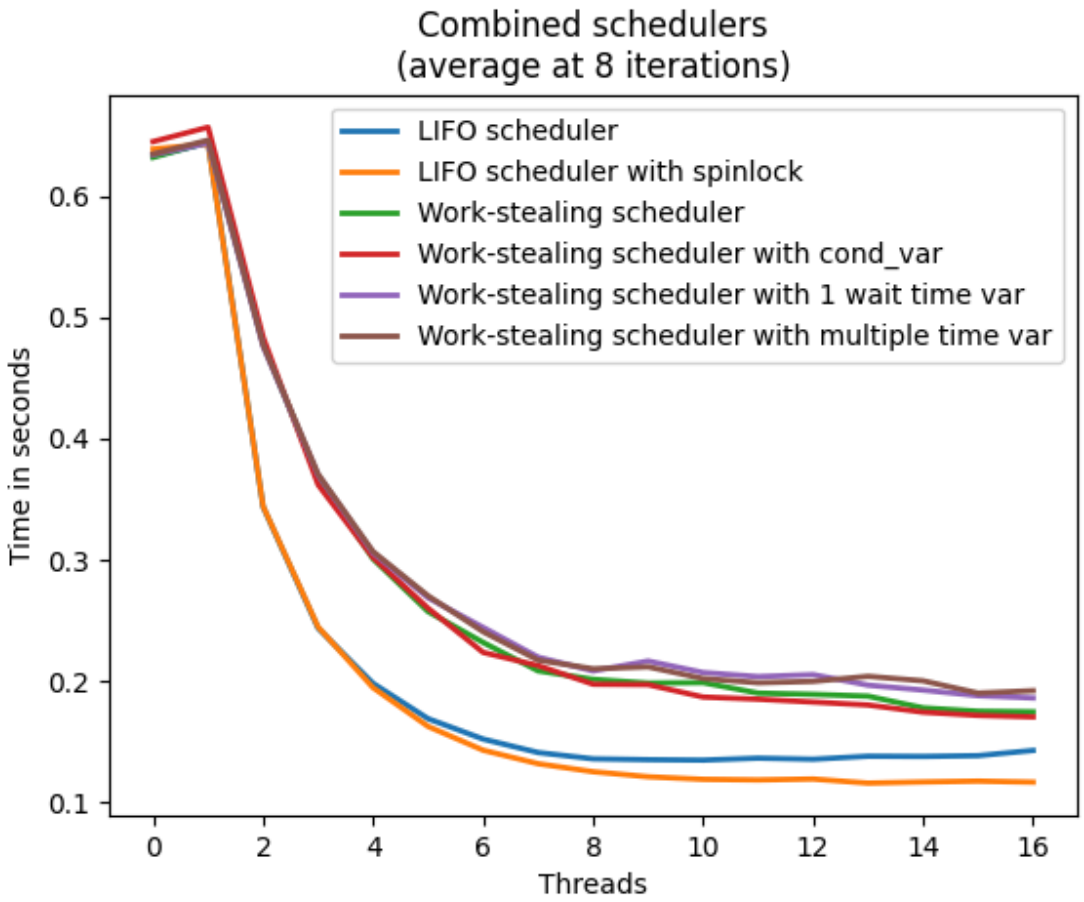

Figure 6: benchmarks des ordonnanceurs sur quicksort avec 16 threads

5. Les démonstrations

Comme première démonstration à vous proposer, nous avons adapté le code fournit au TP10 permettant d'explorer une approximation de l'ensemble de Mandelbrot avec nos scheduler.

Nous nous sommes ensuite intéressés au mathématicien russe Gueorgui Voronoï pour ses diagrammes qui sont plus communément appelés "diagrammes de Voronoï" ou encore "décomposition de Voronoï". Un diagramme de Voronoï est un pavage du plan, 2D ou 3D en cellule adjacentes à partir d'un ensemble de points appelés "germes". Ce pavage est construit de sorte à ce que chaque cellule abrite un nombre de germes strictement égal à 1 et forme l'ensemble des points plus proches de ce germes que d'aucun d'autres.

Il existe déjà un algorithme prouvés optimal pour la construction d'un tel diagramme mais l'optique de ce projet est d'avoir une librairie capable d'améliorer les performances d'un programme en le parallélisant et non pas en le codant efficacement.

Donc nous avons une démonstration sans librairie externe permettant de générer un fichier texte codant la couleur de chaque pixel d'une image représentant un diagramme de Voronoï avec deux algorithmes différents. Un premier très semblable à celui vu en TP pour mandelbrot et un second qui scinde l'espace à calculer en 4 jusqu' à une certaine dimension avant de calculer le découpage des cellules. Le second algorithme peut bénéficier d'encore bien des améliorations mais il est fonctionnel.

En dernière démo nous nous sommes permis d'utiliser une librairie externe pour gérer une fenêtre et ses rafraîchissements. Dans cette démonstration on reprend le diagramme de Voronoï mais en effectuant une translation sur les germes tous les x temps pour voir les cellules croître et décroître en fonction de leur position.

Comme illustré par le benchmark, la différence entre nos différentes adaptations d'ordonnanceurs ne se fait pas grande mais une nette amélioration est constatée entre le séquentiel et le parallèle, ce qui aurait été inquiétant dans le cas contraire.

6. Benchmark sur différents matériels

Il peut être important de voir comment se comportent les ordonnanceurs avec moins de cœurs et moins de threads.

En comparant les benchmarks de la figure 6 et 7, la plus grosse différence notable est l'écart de performance au nombre maximum de threads entre l'ordonnanceur LIFO et work-stealing.
Pour le processeur à 6 threads, on note 65% d'écart de performance entre les deux meilleurs courbes de types d'ordonnanceur, contre 35% pour celui à 16.
On imagine donc que l'écart de performance pourrait se rapprocher en augmentant le nombre de threads, comme évoqué dans la partie 4.

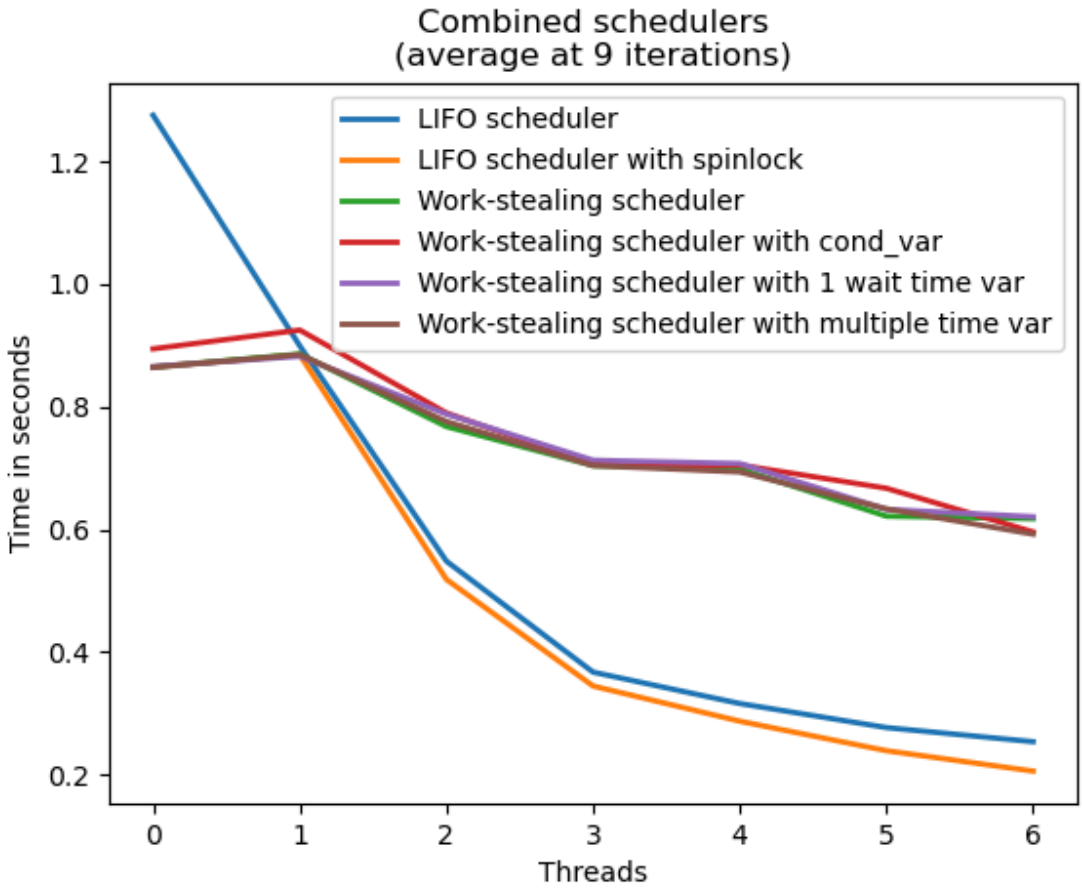

Figure 7: benchmarks des ordonnanceurs sur quicksort avec 6 threads

7. Consommation matérielle

Une autre information pertinente liée au benchmark de ces ordonnanceurs est de regarder la consommation CPU moyenne lors de l'exécution avec quicksort.

On s'aperçoit que les deux ordonnanceurs avec variation de temps ont des consommations très proches, bien que leurs résultats soient différents, le mutex n'est pas plus coûteux dans cette situation.
On peut remarquer que les variables de conditions sont gourmandes en ressources, le LIFO et le work-stealing étant supérieur à leur implémentation concurrentes.
Pour finir, le LIFO spinlock est l'ordonnanceur le plus efficace dans la globalité et consomme 10% de moins que le second moins coûteux.

LIFO	70%
LIFO spinlock	62%
Work-stealing	71%
Work-stealing variable de condition	74%
Work-stealing avec 1 variable de temps	67%
Work-stealing avec n variables de temps	68%

Figure 8: consommation CPU moyenne des ordonnanceurs, benchmark quicksort sur 16 threads

8. Conclusion

Pour conclure, la performance et consommation du LIFO spinlock nous ferait comprendre que la simplicité du code sur un nombre de cœurs et threads raisonnables serait plus pertinent qu'une implémentation en work-stealing.
La consommation CPU étant relativement élevée en comparaison des performances du LIFO spinlock, il serait intéressant d'étudier son évolution sur une machine disposant d'un plus grand nombre de threads, afin de déterminer jusqu'à quel point elle demeure concurrentielle.

Yann PICKERN 22012875
Lucas RODRIGUEZ 22002335