Programmation Systeme : Rapport de Projet

FONTAINE Victor - EL FILALY Ahmed Salem - M1 Informatique

13 mai 2018

Dans ce projet, nous avons implémenté un ordonnanceur (dans le langage C) qui utilise deux stratégies différentes, *LIFO* et *Work Stealing*, et nous avons comparé leurs performances. Nous avons également fait des statistiques sur la partie *Work Stealing*. La première stratégie est un **ordonnanceur LIFO** qui ne contient qu'une seule pile de taches et plusieurs *threads* qui viennent prendre des taches dans cette pile. La deuxième stratégie est un **scheduler par *Work Stealing*** qui utilise une *deque* par *thread*. Nous avons comparé leurs performances sur une machine *linux x86-64* qui a 4 cœurs, ainsi que sur la machine *nivose* de l'université qui a 8 cœurs.

Ordonnanceur *LIFO*

Premièrement nous avons implémenté un ordonnanceur *LIFO* qui nous servira de point de comparaison pour évaluer les performances de l'ordonnanceur par *work stealing*. Pour cela nous avons utilisé différentes structures. La première étant l'ordonnanceur :

```
typedef struct scheduler {
  int nthreads; //nombre de threads
  int qlen; //nombres minimum de taches simultanees
  pthread_t * threads; //tableau de threads
  struct Lifo * lifo; //la pile d'execution (lifo)
  pthread_mutex_t mutex; //primitive de synchronisation
  pthread_mutex_t mutex_sleep; //primitive de synchronisation
  pthread_cond_t cond; // condition d'arret pour les mutex
  int nbre_th_sleep; //nombre de thread qui dorment
} scheduler;
```

qui utilise la structure *Lifo* :

```
typedef struct Lifo {
  int taille; //nombre d'elements dans la pile
  Element * dernier; //dernier element de la pile
  pthread_mutex_t mutex; //primitive de synchronisation
} Lifo;
```

qui elle même utilise la structure *Element* qui représente un élément de la pile :

```
typedef struct Element {
  taskfunc t; //la fonction a execute
  void * closure; //les arguments de la fonction
  struct Element * prec; //l'element precedent dans la pile
} Element;
```

Nous avons également implémenté des fonctions nous permettant d'empiler et de dépiler un élément de la pile d'exécution *lifo* :

```
void empiler(struct Lifo * lifo, taskfunc f, void *closure);
Element* depiler(struct Lifo * lifo);
```

Ensuite nous avons initialisé l'ordonnanceur dans la fonction *sched_init* avec les paramètres passés en arguments, et la pile ne contenant que la tache initiale. Les *threads* sont alors lancés avec *pthread_create* executant une fonction qui

suit la stratégie suivante : Lorsqu'un *thread* n'a rien à faire, il dépile une tâche de la pile et l'effectue ; s'il n'y a aucune tâche prête, le *thread* s'endort (avec la fonction *pthread_cond_wait*) en attendant qu'il y en ait une : il est réveillé par *sched_spawn* en utilisant la fonction *pthread_cond_signal*. Lorsqu'une nouvelle tâche est créée, elle est empilée sur la pile. L'ordonnanceur termine lorsque la pile est vide et tous les *threads* sont endormis.

Comme la pile est une structure partagée, nous utilisons des primitives de synchronisation : les *pthread_mutex* que l'on prend avec *pthread_mutex_lock* et que l'on relache avec *pthread_mutex_unlock*.

Ordonnanceur par *Work Stealing*

Pour l'ordonnanceur par *Work Stealing*, chaque *thread* contient sa propre structure de donnée : une *deque* a deux bouts (haut et bas) qui est une liste doublement chaînée et qui a quatre opérations qui permettent d'enfiler et de défiler des taches (*Element*) de la *deque* donnée :

```
void enfilerHaut(struct Deque * deque, taskfunc f, void *closure);
Element* defilerHaut(struct Deque * deque);
void enfilerBas(struct Deque * deque, taskfunc f, void *closure);
Element* defilerBas(struct Deque * deque);
```

Pour cela, nous avons créé une structure *Mythread*, et l'ordonnanceur (*scheduler*) contient maintenant un tableau de *Mythread*. De plus, chaque *Mythread* contient une *deque* :

```
typedef struct scheduler {
  int nthreads; //nombre de threads
  int qlen; //nombres minimum de taches simultanees
  MyThread * mythreads; //les threads
  pthread_mutex_t mutex; //primitive de synchronisation
  int nthreads_sleep; //le nombre de threads endormis
} scheduler;

typedef struct MyThread {
  pthread_t thread;
  struct Deque * deque;
  int id;
  int sleep; //1 si il dors, 0 sinon
  int nb_t_eff; //nombre de taches effectuees
  int ws_success; //nombre d'etapes de work stealing reussies
  int ws_fail; //nombre d'etapes de work stealing echouees
} MyThread;

typedef struct Deque {
  int taille; //nombre d'elements dans la deque
  Element * premier;
  Element * dernier;
  pthread_mutex_t mutex; //primitive de synchronisation
} Deque;
```

```
typedef struct Element {
  taskfunc t; //la fonction a execute
  void * closure; //les arguments de la fonction
  struct Element * prec; //l'element precedent dans la pile
  struct Element * suivant; //l'element suivant dans la pile
} Element;
```

Ensuite nous initialisons l'ordonnanceur dans la fonction *sched_init* : toutes les *deques* sont vides sauf une, qui contient la tâche initiale. Lorsqu'un *thread* a fini d'exécuter une tâche, il défile la tâche qui est en bas de la *deque* qui lui est associée ; si cette *deque* est vide, il effectue une étape de *work stealing*. S'il n'a toujours pas réussi à trouver du travail, il s'endort pendant 1 ms, puis tente une étape de *work stealing* de nouveau :

```
while(ws == 0){//tant que le work stealing echoue (ie return 0)
  ws = workStealing(args->scheduler,args->id);
  if(ws == 0){
    args->scheduler->mythreads[id].ws_fail ++;
    usleep(1000); //dors 1ms avant de recommencer une etape de WS
  }else{
    args->scheduler->mythreads[id].ws_success ++;
  }
}
```

Lorsqu'une nouvelle tâche est créée, elle est enfilée en bas de la *deque* associée au *thread* qui a fait l'appel à *sched_spawn* (la création de tâches est locale). L'ordonnanceur termine lorsque tous les *threads* sont en train de dormir après avoir échoué à voler du travail.

Dans cette stratégie nous avons aussi utilisé des *pthread_mutex_t* pour nos primitives de synchronisation, pour protéger les *deques* lors des étapes d'enfilage, de défilage et de *work stealing*. Pourtant, pour le *work stealing*, notre programme ne se comporte pas toujours parfaitement : environ 1 fois sur 10 pour 4 cœurs, le programme s'arrête à cause d'une *Segmentation Fault*. Nous avons alors cherché la source du problème, notamment grâce à l'outil de débogage *C debugger (gdb)*, et nous avons remarqué que quelques fois, un *Element* d'une *deque* était corrompu : son *suivant* ne pointait plus sur rien. Cela est sûrement dû à un *thread* qui effectue une étape de *work stealing* en même temps qu'une étape d'enfilement ou de défilement à lieu par un autre *thread*, et qui, à cause probablement d'une mauvaise utilisation d'un *pthread_mutex_t* à ce moment la, modifie le même *Element*, ce qui amène à une *Segmentation Fault*. Malgré de nombreux efforts pour localiser et corriger le problème, nous ne sommes pas parvenus à totalement le corriger dans les temps impartis.

Cependant notre programme fonctionne normalement la plupart du temps, et donne des résultats cohérents avec nos attentes, ce qui nous renforce dans l'idée que le programme fonctionne bien, et fait ce que l'on attend de lui lorsqu'il termine son exécution. Nous avons donc pu utiliser les résultats obtenus pour les comparer à ceux de l'ordonnanceur *Lifo*.

Comparaison des performances entre Ordonnanceur *Lifo* et Ordonnanceur par *Work Stealing*

Nous avons évalué les performances des deux ordonnanceurs différents sur une machine qui a 4 cœurs. Nous avons lancé le programme une vingtaine de fois pour chaque stratégies et pour différents nombres de *threads* utilisés : 1,2,3 ou 4. Nous avons ensuite fait la moyenne de ces résultats et nous les avons représenté sous la forme d'un graphique comportant deux courbes (une pour l'ordonnanceur *Lifo* et une pour l'ordonnanceur par *work stealing*) qui sont les temps moyens d'exécution en secondes en fonction du nombre de *threads* utilisés :

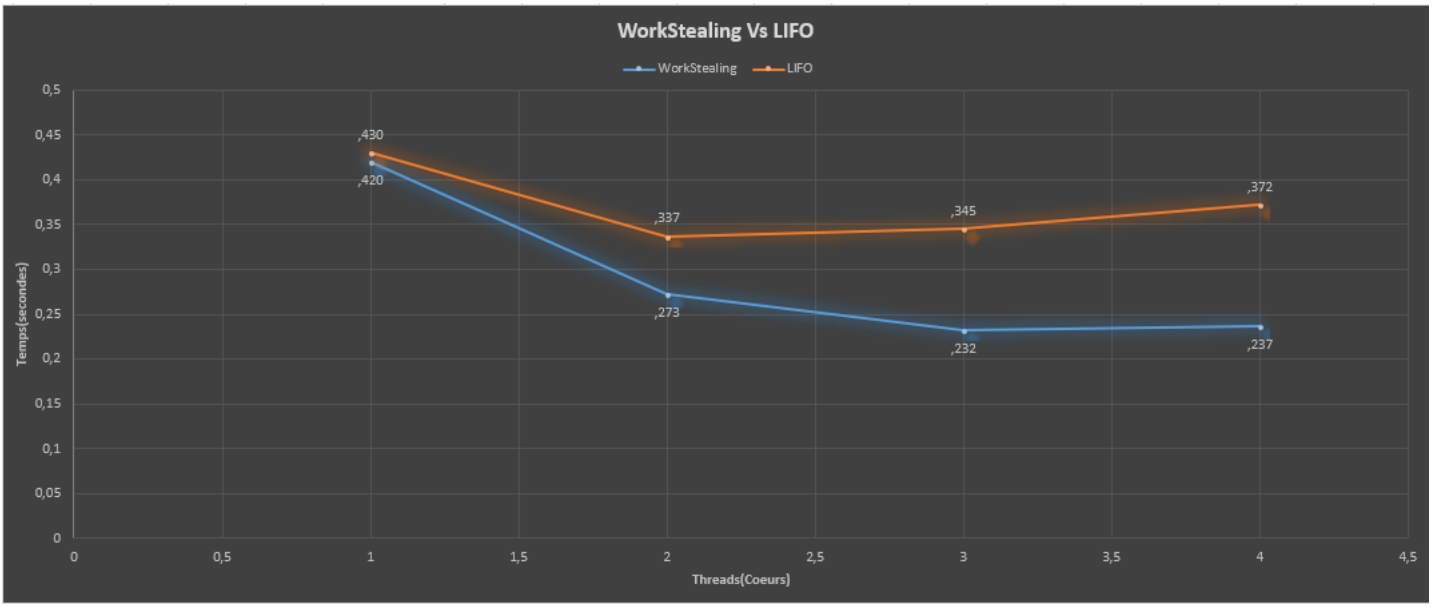

FIGURE 1 – Graphique de comparaisons des performances de *Lifo* et *Work Stealing* sur une machine 4 cœurs

On remarque alors que pour 1 *thread*, les deux stratégies ont à peu près les mêmes performances. Cela est cohérent car un ordonnanceur par *work stealing* qui n'a qu'un *thread* à une *deque* se comporte de la même manière que l'ordonnanceur *Lifo*. Ensuite, plus le nombre de cœurs augmente et plus les différences de performances entre les deux stratégie sont évidentes : dans le cas de l'ordonnanceur par *work stealing*, le temps d'exécution décroit quand le nombre de cœurs augmente, contrairement à l'ordonnanceur *Lifo* pour lequel les performances restent plus ou moins les mêmes. Cela s'explique par le fait que dans l'ordonnanceur par *work stealing*, chaque *thread* a sa propre *deque* ce qui lui permet de créer les taches localement (optimisation de la localité : meilleure utilisation du cache).

Nous avons aussi lancer une vingtaine de fois le programme pour les deux stratégies sur la machine *nivose* de l'université qui a 8 cœurs. Les résultats observés nous amènent aux mêmes conclusions que précédemment :

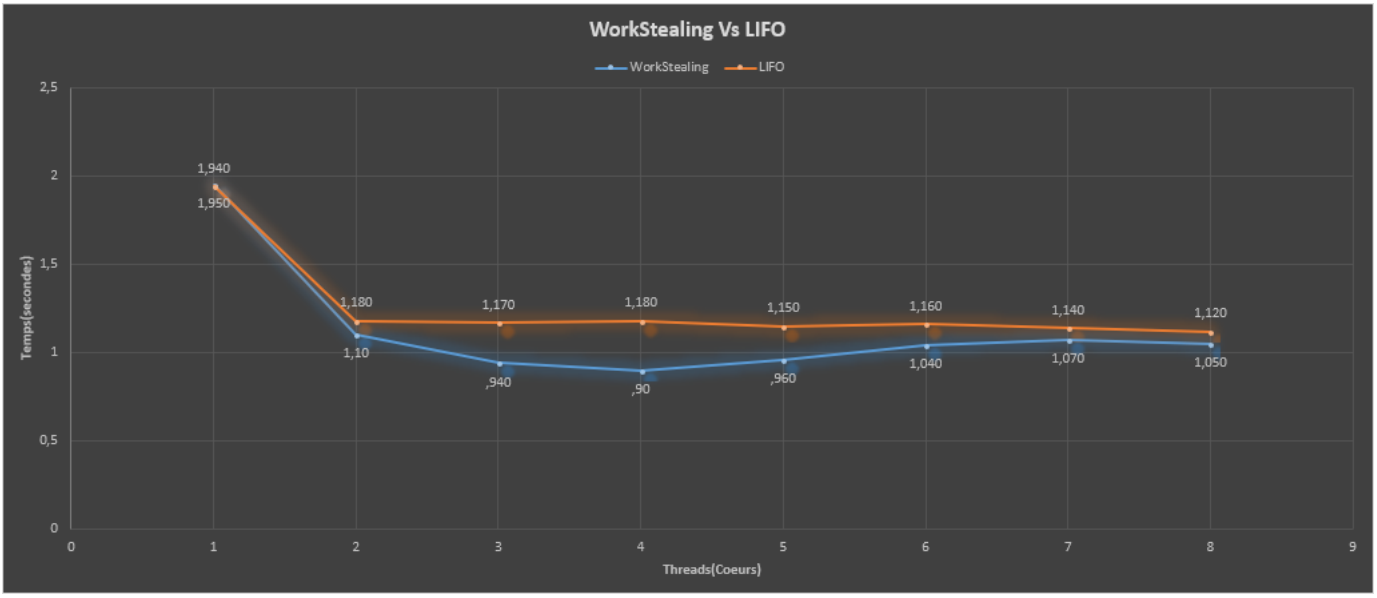

FIGURE 2 – Graphique de comparaison des performances de *Lifo* et *Work Stealing* sur la machine ***Nivose*** de l'université qui a 8 cœurs

Nous avons également fait quelques statistiques sur la stratégie *work stealing* pour savoir pour chaque *thread* combien de taches il avait effectué, combien d'étapes de *work stealing* ont échouées et combien ont réussies. Nous avons fait la moyenne d'un certain nombre de résultats (environ 20) pour le *work stealing* sur une machine a 4 cœurs et qui utilise 4 *threads* :

	Thread n-0	Thread n-1	Thread n-2	Thread n-3
Nombre de taches effectuées	119488	111170	111290	116789
Nombre d'étapes de Work Stealing réussies	10	13	13	11
Nombre d'étapes de Work Stealing échouées	1	15	15	16
Pourcentages d'étapes de Work Stealing réussies	90,90%	46,42%	46,42%	40,74%

FIGURE 3 – Statistiques de l'ordonnanceur par *work stealing* sur une machine 4 cœurs

Nous remarquons que le premier *thread* effectue en moyenne plus de taches que les autres, cela est dû au fait que c'est dans sa *deque* qu'est initialement enfilée la première tache. De plus on remarque que très peu d'étapes de *work stealing* ont lieu, et ont une pourcentage de réussite plutôt élevé, ce qui implique un bon fonctionnement de l'ordonnanceur et une bonne localité de création des taches.