JACQUETTE Pierrick
pierrick.jacquette@gmail.com

TP N°2 - Colimaçon

Temps :

Temps estimé à 10h et fini en 7h15 :

- 30 min pour prendre en compte les remarques du TP1
- 30min pour faire le code de test en temps (clock.c)
- 1h de code pour faire les threads
- 1h pour faire le code des tests automatiques
- 15 min pour améliorer mon code suite à un test négatif
- 1h pour les commentaires et le refactoring
- 30min makefile avec -m32
- 30min pour les librairies
- 2h pour le rapport

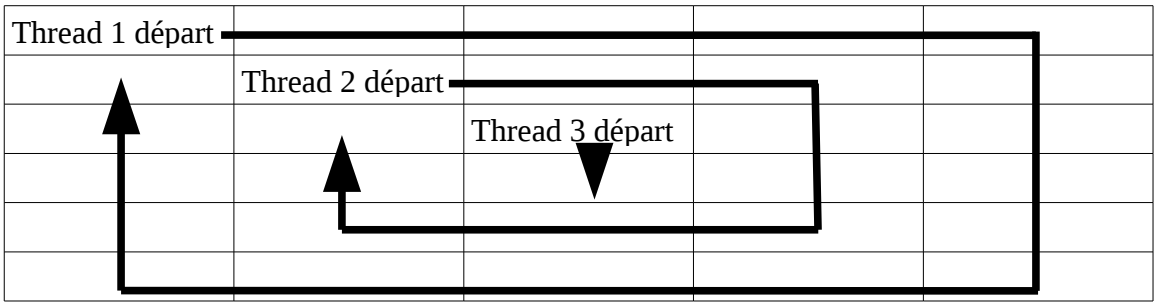
Exécution : $ make

- Pour exécuter le main.c : $ ./colimacon ligne colonne [affichage] [thread]
  Les arguments entre crochet sont facultatifs. Ils permettent de définir respectivement si l'on veut un affichage (1 ou non 0) et lancer l'algorithme avec des thread (1 ou non 0).
  Par exemple ./colimacon 5 4 0 1 : 5 ligne, 4 colonne, pas d'affichage et avec threads.
- Le make génère les deux librairies (dynamique et statique) et les deux exécutables sur test.c
- Le make génère l'exécutable qui calcule le temps d'exécution :
  $ ./clock ligne colonne [thread] [nombre]
  Les arguments entre crochet sont facultatifs. Ils permettent de définir respectivement si l'on veut lancer l'algorithme avec des thread (1 ou non 0) et le nombre de fois où l'on l'éxécute.
- Pour les tests automatiques : il faut avoir la librairie check.h d'installé (pour l'installer sudo apt-get install check)
  $ cd Test puis $ make puis $ ./ check_main
  J'ai préféré faire deux makefile au cas où «check.h» ne serait pas installé comme sur lucien.

Choix & problèmes rencontrés :

J'ai choisi d'utiliser un tableau à une dimension et non à deux. Ce qui me permet d'avoir un seul malloc et pas de fuite de mémoire si $n^{ième}$ malloc ne fonctionne pas. J'ai implémenté une version parallélisée de l'algorithme en plus de l'autre version (avec des appels différents).
int colimacon(int** tab, int ligne, int colonne);
int colimaconThread(int** tab, int ligne, int colonne);
Ceci me permet de faire les calcul sur les thread et le int de retour de renvoyer -1 si une erreur se produit et 0 sinon. Le nombre d'accès au tableau de taille n est n pour les écritures et 0 lecture.

Ma version parallélisée permet d'avoir un thread pour chaque tour du tableau à remplir. Soit min(# ligne, # colonne) / 2 threads avec chacun une case de départ du tour, la valeur minimale et maximale à inscrire. J'ai une fonction me permettant de calculer toutes ces valeurs.
Par exemple :



JACQUETTE Pierrick
pierrick.jacquette@gmail.com

Paralléliser jusqu'à un thread pour une case est plus coûteux en temps et espace.

Test d'exécution : clock.c et valgrind

C'est le temps moyen pour 10 exécutions. Mon pc est sous linux avec i7 et 8go de ram.

	Temps d'exécution (en seconde)				Occupation mémoire			
Taille	Standard		Thread		Standard		Thread	
	lucien	mon pc	lucien	mon pc	en byte	en block	en byte	en block
10 x 10	0.003		0.003		400	1	1 920	11
100 x 100	0.003		0.007	0.005	40 000	1	55 200	101
1 000 x 1 000	0.019	0.005	0.057	0.008	4 000 000	1	4 152 000	1 001
10 000 x 10 000	3.571	0.295	3.156	0.289	400 000 000	1	401 520 000	10 001

Test unitaire : Dossier Test

J'ai effectué un fichier de test en me servant de la librairie « check.h ». Je compare la valeur de chaque case du tableau rempli par la fonction colimaçon avec le résultat attendu.
J'effectue des tests pour des tableaux sur une seule ligne, une seule colonne, une seule case, un carré pair, un carré impair, un rectangle pair/pair, un rectangle pair/impair, un rectangle impair/impair, un rectangle impair/pair, 10x10 et 100x100. (11 tests)
J'effectue cela pour le cas sans thread et le cas avec thread. Enfin comme le code d'initialisation du tableau et de vérification des arguments est commun j'effectue une seule fois les tests : arguments positives, taille du tableau < taille d'un entier et pointeur du tableau est différent de NULL. (3 tests). Soit un total de 25 tests.

Voici l'output que vous devriez obtenir (je l'ai mis au cas où) : une ligne par test, un test par cas :

```
$.check_main
100 %: Checks : 25, Failures : 0, Errors:0
check_colimacon.c : 14 : P : StandardColimacon : colimacon_1_1 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_1_8 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_8_1 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_4_4 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_5_5 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_4_6 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_6_5 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_3_5 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_3_4 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_10_10 : 0 : Passed
check_colimacon.c : 14 : P : StandardColimacon : colimacon_100_100 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_1_1 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_1_8 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_8_1 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_4_4 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_5_5 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_4_6 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_6_5 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_3_5 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_3_4 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_10_10 : 0 : Passed
check_colimacon.c : 14 : P : ThreadColimacon : colimacon_thread_100_100 : 0 : Passed
check_colimacon.c : 195 : P : LimitsColimacon : taillePositive : 0 : Passed
check_colimacon.c : 211 : P : LimitsColimacon : tailleTropGrande : 0 : Passed
check_colimacon.c : 227 : P : LimitsColimacon : null : 0 : Passed
```