Pipeline de tâches

Introduction

Ce devoir maison est évalué et doit être réalisé **en groupe** de 2 à 4 personnes.

Beaucoup de programmes temps réels consistent à faire la même séquence de calculs en boucle mais sur des données différentes, typiquement en utilisant des données issues de capteurs. Il est courant que de tels programmes soient écrits sous la forme d'un **pipeline de tâches**, ce qui a l'avantage de cadrer le programme sous une forme connue dont on peut vérifier certaines propriétés, et de permettre d'exploiter relativement simplement plusieurs ressources de calcul en parallèle.

Le but de ce devoir maison est de voir comment on peut transformer un code séquentiel en un pipeline de tâches exécutable en parallèle. Pour cela, un code séquentiel pouvant facilement être utilisé sur vos machines, proche d'un pipeline de tâches, vous est fourni. Il vous est demandé d'implémenter de différentes manières un pipeline de tâches exécuté en parallèle. Vos implémentations parallèles doivent avoir le même comportement que le code séquentiel fourni. Vous devrez mesurer la performance des implémentations, analyser cette performance et comparer la performance des différentes implémentations entre elles.

Rendu attendu

Vous devez rendre le travail réalisé sous Moodle (1 rendu par groupe).

Votre rendu rendu doit contenir les points suivants.

- Les sources des programmes implémentés.
- Un rapport en **Markdown** qui décrit :
  - la composition de votre groupe (numéro étu, prénom, nom),
  - comment compiler et exécuter vos programmes,
  - une analyse et une comparaison des performances de vos différentes implémentations.

Barème (indicatif)

La note maximale que vous pouvez atteindre dépend du nombre de versions que vous avez implémentées.

- Version 1 : 12
- Versions 1 et 2 : 16
- Versions 1, 2 et 3 : 20

Critères d'évaluation du rapport.

- Clarté et concision de vos explications.
- Qualité de votre analyse (méthodologie utilisée, cohérence de vos conclusions par rapport aux données…).

Critères d'évaluation des codes implémentés.

- Validité des accès mémoire (durée de vie des variables, suffisance des synchronisations).
- Validité de l'API utilisée : types `pthread_t`, `pthread_mutex_t`, `sem_t`.
- Lisibilité du code.
- Minimalisme des synchronisations produites.

Description du programme fourni

Un programme de base séquentiel vous est fourni. Ce programme n'utilise pas directement de données provenant de capteurs, afin de rendre le devoir maison faisable sur n'importe quel ordinateur. Une simulation physique est utilisée à la place, pour faire comme si l'environnement était dynamique.

Cette simulation physique est itérative : les pas de temps sont simulés les uns après les autres.
Pour chaque itération différents calculs sont réalisés en exécutant une suite de tâches.

Liste des tâches du pipeline

Le programme fourni réalise les tâches suivantes les unes après les autres.

1. Simulation d'une itération d'un **système à N corps**.
2. Génération d'images à partir de l'état actuel des corps.
3. Calcul lourd sur l'image générée. Ici, c'est un **flou gaussien**.
4. Enregistrement de l'image floutée sur le système de fichiers. Cette tâche est optionnelle
5. Mise en niveau de gris de l'image.
6. Calcul de statistiques sur l'image en niveau de gris.
7. Enregistrement des statistiques sur le système de fichiers.

Compilation et exécution (CLI et ses arguments)

```
# à la main, tout compiler d'un coup
gcc -o dm-base dm-base.c tasks.c -Wall -Wextra $(pkg-config --libs libpng) -lm

# à la main, compilation séparée
gcc -c -o tasks.o tasks.c -Wall -Wextra $(pkg-config --cflags libpng)
gcc -c -o dm-base.o dm-base.c -Wall -Wextra
gcc -o dm-base dm-base.o tasks.o $(pkg-config --libs libpng) -lm

# via le build system fourni (meson + ninja)
meson setup build
ninja -C build
```

```
usage: ./dm-base <nb-steps> <img-width> <img-height> <save-img>
```

argument	type	sens
`<nb-steps>`	entier	nombre de fois que chaque tâche doit être appelée
`<img-width>`	entier	largeur de l'image à générer, en nombre de pixels
`<img-height>`	entier	hauteur de l'image à générer, en nombre de pixels
`<save-img>`	entier 0 ou 1	active ou non la tâche d'enregistrement de l'image

Cycle de vie des données des tâches

En parcourant le code fourni, vous remarquerez que la plupart des tâches au milieu du pipeline ne modifient pas les données **en place** mais prennent une donnée en entrée et produisent une donnée en sortie.

Cette structure a pour but de vous simplifier grandement la vie en implémentant votre pipeline. Le code de base applique un mécanisme de **double buffering** pour stocker les images avec une empreinte mémoire constante.

Performance et affichage

Les tâches fournies **profilent** elles-mêmes leur performance. Chaque tâche compte le nombre de nanosecondes passées à faire des calculs. À la fin du programme, le temps total passé dans chaque tâche est affiché, ainsi que la proportion du temps passé dans la tâche.

Version 1

Iimplémentez dans un fichier `dm-v1.c` une version parallèle du code séquentiel fourni. Ce programme doit accepter les mêmes entrées que le code séquentiel, réaliser exactement les mêmes affichages, et produire exactement les mêmes fichiers de sortie. Vous pouvez utiliser `diff` pour vérifier que les fichiers produits sont identiques. Il est recommandé de commencer ce code par une copie de `dm-base.c`.

Comportement attendu du thread principal.

1. Similaire à la base : lecture des entrées, allocation et initialisation des données, prise d'une mesure du temps courant.
2. Création d'un thread pour chaque type de tâche qui doit s'exécuter.
3. Attente de la fin de tous les threads.
4. Similaire à la base : prise d'une mesure du temps courant, affichage de statistiques de performance, nettoyage mémoire.

Comportement attendu de chaque thread qui gère un type de tâche.

```
Faire <nb-steps> fois
  Exécuter la tâche
S'arrêter
```

Vous devez synchroniser les exécutions pour être sûr que chaque exécution de tâche soit valide. Il est attendu que vous implémentiez une fonction C différente pour chaque type de tâche. Par exemple une fonction `f_simu` qui gère toutes les exécutions de la tâche de simulation des N corps, une fonction `f_img_gen` qui gère toutes les exécutions de la tâche de génération d'image à partir de la position des N corps…

Temps	0	1	2	3	4	5	6	7
Thread principal	init							exit
Thread `simu`		step 0	step 1	step 2	step 3			
Thread `img_gen`			step 0	step 1	step 2	step 3		
Thread `img_blur`				step 0	step 1	step 2	step 3	

Table 1: Illustration de l'exécution du pipeline de tâches, en supposant que `<nb-steps>` vaut 3 et que seuls les trois premiers types de tâches sont exécutés (simulation, génération d'image et floutage de l'image). Par exemple, au temps 2, le thread principal et le thread `img_blur` ne font rien, le thread `simu` calcule le step 1 de la simulation des N corps, le thread `img_gen` génère l'image correspondant au step 1 de la simulation.

Version 2

Iimplémentez dans un fichier `dm-v2.c` une version parallèle du code séquentiel fourni. Ce programme doit accepter les mêmes entrées que le code séquentiel, réaliser exactement les mêmes affichages, et produire exactement les mêmes fichiers de sortie. Vous pouvez utiliser `diff` pour vérifier que les fichiers produits sont identiques. Il est recommandé de commencer ce code par une copie de `dm-v1.c`.

Cette version est très similaire à la précédente, vous devez implémenter en parallèle l'exécution du pipeline de tâches. La différence est que l'attribution des tâches aux threads ne doit pas être statique mais dynamique. Vous devez pour cela utiliser une file de messages synchronisée, dans laquelle les différents threads vont produire ou consommer des requêtes de calcul de tâches. Vous pouvez réutiliser la file de messages synchronisée que vous avez implémenté au TP précédent (le TP noté).

Le comportment attendu du thread principal est le même qu'en version 1, sauf qu'entre la création de tous les threads et l'attente de leur terminaison, le thread principal doit aussi produire une demande d'exécution d'une tâche de type `simu` pour le step 0.

Comportement attendu des threads (sauf thread principal).

```
En boucle :
  t = consommer une tâche
  exécuter(t)
  produire les tâches de sortie nécessaires
```

Les tâches de sortie à produire dépendent du type de tâche réalisée. Par exemple, l'exécution d'une tâche de simulation au step 0 devrait libérer la demande d'exécution d'une tâche de génération d'image au step 0, ainsi que d'une tâche de simulation au step 1.

Vous devez faire en sorte que votre programme s'arrête correctement quand toutes les tâches du pipeline ont été calculées. Votre thread principal doit reprendre la main à la fin pour faire ce qu'il fait dans toutes les versions : afficher des statistiques sur le temps passé dans chaque tâche et faire des nettoyages mémoire si nécessaire.

Version 3

Iimplémentez dans un fichier `dm-v3.c` une version parallèle du code séquentiel fourni. Ce programme doit accepter les mêmes entrées que le code séquentiel, réaliser exactement les mêmes affichages, et produire exactement les mêmes fichiers de sortie. Vous pouvez utiliser `diff` pour vérifier que les fichiers produits sont identiques. Il est recommandé de commencer ce code par une copie de `dm-v2.c`. Si vous avez besoin de modifier le module qui implémente les tâches, créez, modifiez et utilisez des fichiers `tasks-v3.h` et `tasks-v3.c`. **Rendez également ces fichiers si vous les avez utilisés.**

Cette version, comme la version 2, demande d'attribuer dynamiquement les tâches aux threads. Cette version devrait cependant permettre plus d'optimisation.

La vitesse d'exécution d'un pipeline est limitée par la tâche la plus lente du pipeline.

Grâce aux mesures du temps passé dans chaque tâche, déterminez quelle tâche ralentit votre programme quand vous l'exécutez sans écrire l'image sur votre système de fichiers, c'est-à-dire quand `<save-img>` vaut 0.

Découpez cette tâche en plusieurs sous-tâches afin de l'intégrer dans votre pipeline de calcul. Vous devrez probablement légèrement modifier le code des tâches elles-mêmes pour rendre cela possible. Vous devrez probablement modifier votre code de synchronisation.