UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

RAPPORT APP1

Systèmes d'exploitation
GIF642

Présenté à
Domingo Palao Munoz

Présenté par
Équipe numéro 2
Étienne Beaulieu – beae0601
Emile Bureau – bure1301

Sherbrooke – 8 mai 2024

Table des matières

Identification des problèmes

Avant même d'avoir exécuté le code de base, l'équipe a remarqué qu'il peut y avoir des erreurs qui se glissent dans le code d'une fois à l'autre. Ces erreurs sont liées à des problèmes de concurrence entre les processus. En outre, une queue est utilisée dans deux méthodes différentes. Lorsque l'on utilise plusieurs fils, il est possible d'observer un problème à ce niveau. Si les deux méthodes tentent d'utiliser un même élément de la queue, il se peut qu'un modifie en écriture et que l'autre ne lise pas les changements qui sont faits à l'instant même. Il est donc fort probable que les résultats ne soient pas ceux que l'on croit obtenir à la fin du script. L'utilisation de cette queue est donc une section critique du script. Pour surmonter les erreurs, il est nécessaire d'effectuer des changements au code. Si le code effectué dans les méthodes était atomique, il n'y aurait pas de problème de concurrence à ce niveau.

Il y a aussi un problème d'utilisation de CPU élevé dû à une boucle de type « while(true) » qui n'a aucun mécanisme pour ralentir l'exécution du programme et donc réduire l'utilisation de CPU.

Mise en œuvre des mécanismes de synchronisation

Afin de régler les erreurs de synchronisation, l'équipe a implémenté divers mécanismes, soit des mutexes, des verrous et des variables de condition.

En premier lieu, l'utilisation des mutexes et des verrous permet d'éliminer la concurrence en utilisant plusieurs fils pour exécuter le script. Un mutex est créé, nommé « mutex_ » pour s'assurer du bon fonctionnement entre les méthodes « processQueue » et « parseAndQueue ». L'utilisation de ce mutex avec un verrou de type « unique_lock » permet de bloquer l'exécution du code tant que le verrou n'est pas débloqué par le fil qui l'a instancié. Dans ce cas, « parseAndQueue », bloque avec le verrou lorsque la méthode tente t'ajouter une nouvelle tâche à la queue. De ce fait, « processQueue » va être bloqué et ne pourra pas continuer son exécution. Ce même mutex est utilisé à l'intérieur d'un verrou dans cette méthode pour réaliser le même processus si jamais c'est cette méthode qui s'est exécutée avant.

Maintenant, afin de s'assurer que le fil est débloqué à la suite des modifications faites par une des deux méthodes, l'équipe a ajouté une variable de condition. Celle-ci permet de déverrouiller un verrou d'un fil lorsque l'on reçoit une notification. Le verrou de « processQueue » attend de recevoir la notification comme quoi le « parseAndQueue » a terminé avant de pouvoir continuer. En utilisant « notify_all » sur la variable de condition « condVar », les fils sont tous notifiés que l'exécution est terminée. Ils sont donc en mesure d'enlever le verrou et de continuer leur exécution.

En plus des mutexes pour les méthodes, l'équipe a aussi ajouté un mutex nommé « mutex_end » dans l'exécution du main. Celui-ci est utilisé afin de s'assurer que la queue de tâches est bien vide et que l'exécution de toutes les méthodes est bien terminée. Un verrou créé avec ce mutex fait un « wait » sur la même variable de condition qui est utilisée dans les deux méthodes ci-dessus, soit « condVar ». De cette façon, on s'assure d'attendre la notification de la variable avant de terminer l'exécution du code. Cette notification est envoyée par « processQueue » lorsque l'exécution est finie. Sans ce verrou, il est arrivé que le code n'eût pas le temps de terminer son exécution, résultant ainsi en des images manquantes ou

corrompues. Le processus pouvait aussi simplement ne jamais s'arrêter et donc continuer d'utiliser le CPU.

Pour les détails du code, voir ANNEXE I : Méthode parseAndQueue et ANNEXE II : Méthode processQueue.

Gestion des processus

Pour exécuter le travail, l'équipe utilise le script Bash "run_batch_mp.sh". Ce script invoque "multi_proc.py" avec les fichiers que l'on veut convertir. Pour changer le nombre de processus que l'on souhaite utiliser, il est nécessaire de changer la variable constante "NUM_PROCESS" qui est créée dans le fichier "multi_proc.py". Ce fichier s'occupe d'utiliser le nombre de processus qu'on lui demande pour le script. Pour le nombre de fils à utiliser, l'équipe a fait des changements à "asset_conv.cpp" ainsi qu'à "multi_proc.py".

Afin de changer facilement le nombre de fils, l'équipe a choisi d'opter pour un argument en ligne de commande à "asset_conv.cpp". Puisqu'il y a déjà une variable nommée "NUM_THREADS" dans le fichier, on doit seulement changer cette valeur à l'aide d'un argument. Si aucun argument n'est spécifié, alors la valeur par défaut d'un thread est utilisée. En ajoutant la ligne suivante et en changeant la valeur de "NUM_THREADS" pour cette nouvelle variable créée :

```
1  int threadCount = NUM_THREADS;
2      std::mutex mutex_end;
3      std::condition_variable condVar_end;
4
5      if (argc >= 2)
6      {
7          threadCount = std::atoi(argv[1]);
8      }
```

Puisque l'on permet maintenant l'ajout d'un argument en ligne de commande pour le nombre de fils à utiliser, on peut également faire usage de cette modification dans "multi_proc.py" où l'on invoque le fichier "asset_conv.cpp". On ajoute le nombre de fils (3 fils pour le cas suivant) à la suite de l'appel du fichier « asset_conv » :

```
1  NUM_PROCESSES = 4
2
3  def run_process(task_desc):
4      p = subprocess.Popen(["./asset_conv", "3"], stdin=subprocess.PIPE)
```

Dans l'extrait de code ci-dessus, le script sera donc exécuté à l'aide de quatre processus à trois fils chacun.

Gestion des requêtes répétitives

Afin d'optimiser le temps de calcul dans les cas où nous recevons certaines requêtes en double ou plus, nous avons implémenté une cache des requêtes traitées. Il y avait déjà un début d'implémentation de cache dans le code de base, cependant, on y voyait un problème de performance puisqu'il fallait cache les PNG résultant ce qui implique donc de devoir attendre la fin de la conversion d'image avant de pouvoir remplir la cache. Nous avons donc décidé d'implémenter une cache des requêtes, cela nous permet de gagner en temps d'exécution parce que nous évitons d'ajouter une requête à la file d'attente si celle-ci a déjà été traitée. Voir le code à ANNEXE I : Méthode parseAndQueue.

Analyse de la performance

Pour bien comparer les temps d'exécution, l'équipe a décidé de lancer le script trois fois et de faire une moyenne de chacun de temps. De cette façon, elle s'assure que les valeurs retournées ont moins de chances d'être des valeurs aberrantes. Par exemple, on note le temps réel, le temps utilisateur et le temps système de trois exécutions distinctes pour un processeur et un fil. Ensuite on fait la moyenne de chaque valeur pour arriver aux résultats dans le tableau ci-dessous. Les chiffres inscrits dans la première colonne représentent le nombre de processus, tandis que ceux inscrits en entête représentent le nombre de fils :

NB Processus/Fils	Type temps	1	2	3	4
1	Réel	3,63	1,92	2,19	2,58
	Utilisateur	3,30	3,49	3,40	3,40
	Système	0,06	0,05	0,05	0,05
2	Réel	1,90	1,13	1,68	1,57
	Utilisateur	3,42	3,95	3,69	3,69
	Système	0,06	0,04	0,05	0,05
3	Réel	1,35	0,81	1,72	1,30
	Utilisateur	3,61	4,10	3,85	3,85
	Système	0,06	0,09	0,08	0,08
4	Réel	1,11	0,70	1,94	1,25
	Utilisateur	3,77	4,60	4,18	4,18
	Système	0,08	0,10	0,09	0,09

En se référant aux valeurs du tableau ci-dessus, on peut conclure qu'en général, si on augmente le nombre de fils et de processus, le temps d'exécution réel est réduit. Cependant, ce n'est pas toujours le cas. Si on augmente seulement le nombre de processus à 4, on voit une diminution du temps. Lorsque l'on augmente le nombre de fils, il est possible que l'on perçoive une augmentation du temps d'exécution.

Faisons une comparaison sur les tests à 2 processus. Avec deux fils, le temps moyen est de 1,13 seconde, tandis qu'avec quatre fils, on obtient un temps de 1,57 seconde. Donc l'ajout de fils n'améliore pas nécessairement la vitesse d'exécution, et c'est le cas pour chacun des tests. On remarque plutôt le meilleur temps à deux fils, peu importe le nombre de processus utilisés. Le code n'est donc pas nécessairement optimisé pour bien fonctionner sur deux fils.

Pour ce qui est des temps utilisateur et système, on remarque une augmentation du temps plus le nombre de processus et fils est haut, sans avoir une très grande différence avec le test initial d'un processus et un fil. La cause du problème est que puisque l'on augmente le nombre de fils, le processeur doit changer de contexte plus souvent pour naviguer entre les processus et leurs fils. Ces « context switch » demande au processeur de stocker en mémoire le statut du thread et/ou processus et charger le statut du prochain thread/processus à partir de la mémoire. Cela impacte donc bien évidemment le temps de système puisque ça cause beaucoup d'appels système.

En conclusion, la meilleure configuration pour ce travail est, dans le cas de l'équipe, le cas où sont utilisés 4 processus avec 2 fils chacun. Dépendamment des performances de l'ordinateur qui exécute le script et du code lui-même, il est possible que d'autres configurations aient de meilleurs résultats. Il sera donc nécessaire de faire cette même évaluation pour un script totalement différent si l'on souhaite augmenter les performances. Le fait d'augmenter le nombre de conversions d'images à faire pourrait aussi faire en sorte de modifier le résultat obtenu en favorisant peut-être un nombre de thread/processus plus élevé.

Simulateur de la propagation d'ondes

Afin de paralléliser le code des méthodes « curl », nous avons fait une implémentation de ces méthodes en C++. Nous avons ensuite utilisé le script Python pour démarrer un processus C++ et ouvrir un fichier en mémoire partagé par « memory mapping » afin que les deux processus aient accès à la mémoire à tour de rôle. Étant donné que les deux langages utilisent des encodages différents pour les structures de données, nous avons opté pour un encodage brut des données de la matrice en 4 dimensions vers une simple liste d'octets.

Du côté C++, lorsqu'on reçoit la commande de faire du calcul, on commence par lire la mémoire partagée afin de reconstituer la matrice 4d, ensuite on parallélise de calcul de « curl » en démarrant 3 fils pour calculer parallèlement les valeurs dans les 3 axes. Chacun de ces fils démarre ensuite 2 fils afin de séparer les 1 000 000 itérations en 2 fils de 500 000 itérations. On peut voir cette stratégie dans ANNEXE III : Extrait thread pour méthode curl.

Pour valider la preuve de concept, nous avons décidé d'exécuter notre version parallélisée du code en même temps que le code de référence fourni en Python pour comparer nos résultats. Nous calculons donc la nouvelle valeur de « E » avec notre code puis avec le code de référence et comparons les résultats grâce à la méthode « allclose » de la librairie « numpy ».

Nous avons remarqué que notre implémentation est plus lente que le code de référence même si elle donne les mêmes résultats. Nous expliquons ce résultat par le fait que la librairie « numpy » qui est utilisée par le code de référence a été optimisée par une équipe de développeurs expérimentés. D'ailleurs elle

utilise aussi, comme nous, du code en C++ afin d'effectuer des calculs sur des matrices plus rapidement. La différence de temps d'exécution est donc facilement expliquée par la quantité d'optimisation qui a été faite par l'équipe de « numpy ».

ANNEXE I : Méthode parseAndQueue

```
1  void parseAndQueue(const std::string& line_org)
2      {
3          std::queue<TaskDef> queue;
4          TaskDef def;
5          if (parse(line_org, def)) {
6              std::cerr << "Queueing task '" << line_org << "'." <<
   std::endl;
7              std::unique_lock<std::mutex> lock(mutex_);
8
9              if (taskCache_.find(line_org) == taskCache_.end())
10             {
11                 taskCache_.insert({line_org, true});
12                 task_queue_.push(def);
13                 condVar.notify_all();
14             }
15             else
16             {
17                 std::cerr << "Found in cache, not adding to queue." <<
   std::endl;
18             }
19
20         }
21     }
```

ANNEXE II : Méthode processQueue

```
1  void processQueue()
2      {
3          while (should_run_) {
4              std::unique_lock<std::mutex> lock(mutex_);
5              condVar.wait(lock, [&]{return !task_queue_.empty();});
6
7              TaskDef task_def = task_queue_.front();
8              task_queue_.pop();
9
10             lock.unlock();
11
12             if(task_def.fname_in == "finished")
13             {
14                 should_run_ = false;
15                 break;
16             }
17
18             TaskRunner runner(task_def);
19             runner();
20         }
21         condVar.notify_all();
22     }
```

ANNEXE III : Extrait thread pour méthode curl

```
auto threadX = [](Matrix& curl, const Matrix& mtx)
    {
        auto threadX_1 = [](Matrix& curl, const Matrix& mtx)
        {
            for (int i = 0; i < MATRIX_SIZE/2 -1; ++i) {
                for (int j = 0; j < MATRIX_SIZE; ++j) {
                    for (int k = 0; k < MATRIX_SIZE; ++k) {
                        if (j < MATRIX_SIZE -1)
                            curl[i][j+isH][k][0] += mtx[i][j+1][k][2] -
mtx[i][j][k][2];

                        if (k < MATRIX_SIZE -1)
                            curl[i][j][k+isH][0] -= mtx[i][j][k+1][1] -
mtx[i][j][k][1];
                    }
                }
            }
        };

        auto threadX_2 = [](Matrix& curl, const Matrix& mtx)
        {
            for (int i = MATRIX_SIZE/2; i < MATRIX_SIZE; ++i) {
                for (int j = 0; j < MATRIX_SIZE; ++j) {
                    for (int k = 0; k < MATRIX_SIZE; ++k) {
                        if (j < MATRIX_SIZE -1)
                            curl[i][j+isH][k][0] += mtx[i][j+1][k][2] -
mtx[i][j][k][2];

                        if (k < MATRIX_SIZE -1)
                            curl[i][j][k+isH][0] -= mtx[i][j][k+1][1] -
mtx[i][j][k][1];
                    }
                }
            }
        };
```