Anne-Elisabeth Lelièvre 16 084 130

Olivier Beauséjour 16 099 883

Devoir en Processus concurrents et parallélisme

Devoir #3

Rapport de travail

Dans le cadre du cours IFT630

Présenté à :

Gilles Brunet

Université de Sherbrooke

3 août 2018

**Table des matières**

[1. Démarche et recherches 2](#_Toc520406068)

[2. Description du mode de déploiement 3](#_Toc520406069)

[3. Références 5](#_Toc520406070)

# Démarche et recherches

Pour notre démarche, nous avons bien perfectionné nos connaissances sur les MPI, *OpenCL* à l’aide des Power Points.

Pour les recherches au niveau du MPI, nous avons été inspirés par un code sur internet (voir les références). Or ce code ne fait que des multiplications matricielles pour des matrices carrées. Il a donc fallu modifier le code pour permettre la multiplication de matrice pour des matrices non-carrées. Après avoir bien analysé le code pris sur Internet, nous avons fait la remarque d’un processus maître dans le code. En effet, on remarque que le *taskid* = 0 fait référence au travail fait par le maître et lorsque le *taskid* est plus grand que 0, cela correspond au travail fait les travailleurs. En bref, le processus maître initialise les matrices et lance les travailleurs (*workers*). Les travailleurs eux font le calcul de la multiplication matricielle, puis le processus maître collecte les résultats de ceux-ci.

Pour nos recherches au niveau du problème 2, nous avons pris le code en annexe et modifié à notre guise. Pour faire la solution de plusieurs processus, nous avons dû aller sur internet pour apprendre comment lancer plusieurs « threads » en c++. De plus, il a fallu bonifier nos connaissances sur les fonctions *lambda* en c++ et sur le fonctionnement d’un *mutex* en c++, étant donné que nous avons seulement utilisé le *mutex* en c# dans le dernier devoir. Pour *OpenCL*, nous avons recherché des exemples d’utilisations de la librairie. Particulièrement, nous sommes partis d’une base de code effectuant une multiplication matricielle en utilisant la librairie et l’avons modifié incrémentalement. Nous avons aussi utilisé un *CMakeLists* trouvé en ligne pour l’utilisation de la librairie.

# Description du mode de déploiement

Tout d’abord, pour faire l’exécution du programme du problème 1, vous devez rajouté des variables d’environnements : NP, M, N et K en initialisant leur valeur. Vous devez bien respecter les majuscules de ces variables et les noms de ceux-ci, puisqu’on y fait référence dans le code. NP correspond au nombre de processus qui sera exécutés, les variables M, N et K correspondent aux dimensions des matrices telle qu’une multiplication matricielle du type : MxK \* KxN.

Premièrement pour bien comprendre notre solution au problème 1, nous avons créé un fichier script.sh contenant la commande "mpirun -np ${NP} ./cmake-build-debug/main". Ce script permet d’exécuter une application MPI en spécifiant le nombre de processus (la valeur NP. Pour ce problème nous avons décidé arbitrairement d’exécuter la multiplication matricielle en exécutant 4 processus de manière parallèle. Il est donc possible de modifier cette valeur à votre guise. Puis, le "./cmake-build-debug/main" est le chemin relatif de la solution de l’exécutable générée.

Pour exécuter le programme du problème 2, vous devez appeler l’exécutable nommé « bruteForceAttack ». Cet exécutable prend 2 ou 3 paramètre. Le premier paramètre est la chaîne en clair que nous devons effectuer une attaque dessus. Ce paramètre est optionnel. S’il n’est pas spécifié, la chaîne abaaaaaa sera utilisé. L’exécutable encodera la chaîne choisie, puis effectuera l’attaque et finalement retournera la chaîne décodée. Le troisième paramètre est le type d’attaque à employer. Ce paramètre est obligatoire et peut prendre 3 valeurs. La première valeur permise est « -ST ». Ceci exécutera une attaque simple fil. La deuxième valeur permise est « -MT ». Cette valeur exécutera l’attaque multi fils. Enfin, la troisième et dernière valeur est « -CL ». Cette valeur exécutera l’attaque en utilisant la librairie *OpenCL*. Il est important de noter que pour exécuter l’exécutable, le fichier bruteForceAttack.cl doit être au même niveau que l’appel. Voici un exemple d’appel :  
« ./cmake-build-debug /bruteForceAttack zaaaaaa –ST »

Deuxièmement, pour résoudre le problème 2 nous avons créé les fichiers *bruteForceDecoder.h/bruteForceDecoder.cpp* contenant les fonctions permettant d’exécuter le décodage du mot de passe par les 3 versions par la force brute. Il y a donc la fonction qui exécute le mot de passe de façon séquentielle (*singleThreadHostAttack*), une version exécutée de manière parallèle sur les cœurs d’un ordinateur normal, puis finalement une dernière version qui exécute notre code en *OpenCL*. Les fichiers *encoder.h/encoder.cpp* qui contiennent la signature des fonctions et le système d’encodage pris en annexe dans l’énoncé du travail. Ainsi, on y retrouve donc les fonctions *add, xor, decale*, etc. Le fichier *bruteForceMain.cpp* contient l’exécution des versions du système de codage. Le fichier bruteForceAttack.cl contient le code *kernel* pour l’attaque *OpenCL*.

Pour la solution séquentielle du problème 2, nous avons créé une chaîne de caractère arbitraire en appelant la fonction *encode* pour faire comme si c’était un mot de passe. Par la suite, nous appelons notre fonction *singleThreadHostAttack*, qui elle, prend notre mot encodé. La fonction *getNext* permet de trouver la prochaine chaîne et ainsi cette fonction explore toutes les possibilités possibles (d’où le principe de la force brute).

Pour la solution multi fils, nous devions paralléliser notre méthodologie séquentielle. Nous avons donc créé une fonction effectuant le processus pour une chaine particulière et avons créé des fils effectuant ce processus en boucle. Selon cette approche, nous devions avoir une chaîne partagée entre les threads. Cette chaîne représente la chaîne courante à être évaluée. Ainsi chaque fil y accédait et l’incrémentait. En conséquence, nous devions aussi restreindre son accès de sorte que les différents threads ne traitent pas plusieurs la même chaîne. Pour résoudre ce problème, nous avons utilisé un *mutex* pour limiter l’accès à la chaîne partagée.

Pour la solution OpenCL, nous devions transposer notre logique multi fil en code kernel. Nous avons donc traduit toutes les fonctions d’encodage et avons adapté la logique de sorte que nous puissions obtenir la chaîne actuelle à être évaluée en fonction d’une chaîne de base et d’un nombre entier issu du résultat la fonction get\_global\_id(0).

# Références

<https://bousk.developpez.com/cours/reseau-c++/TCP/05-envoi-reception-serveur/>

<https://fr.wikipedia.org/wiki/Signal_(informatique)>

<https://forgetcode.com/c/1201-file-transfer-using-tcp>

<https://stackoverflow.com/questions/1543466/how-do-i-change-a-tcp-socket-to-be-non-blocking>

<https://en.cppreference.com/w/cpp/utility/program/signal>

<http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>

<ftp://www.cs.uregina.ca/pub/class/330/Fork/fork.html>

<https://stackoverflow.com/questions/1002513/non-blocking-version-of-system>