Université de Bretagne Occidentale

Architectures et Système II

Projet : Produit Matriciel

Auteur :
Jimmy Tournemaine

7 février 2016

Table des matières

1 Configuration

1.1 Installation

La commande `make` ou `make all` permet de compiler tous les exécutable nécessaire au lancement des scripts `run` et `compare`.

La commande `make clean` permet la suppression des fichiers binaires et `make mrproper` supprimera tous les fichiers sauf Makefile et les fichiers sources (vos résultats risquent d'être perdus).

```
$ make
$ make clean
```

1.2 Exécution

La compilation génère trois exécutables :

1. `generator` permettant d'éditer ou de générer aléatoirement des matrices.
2. `product_v1` qui calcul les produits et écrit le résultat dans un fichier.
3. `product_fair` idem, en veillant à répartir les threads équitablement sur les processeurs

Deux scripts bash sont disponible pour lancer simplement les exécutables : `run` et `compare`. Pour ces deux scripts, le canal `stderr` est redirigé vers un fichier de logs pour ne pas surcharger l'affichage sur la console.

1.2.1 run

```
$ bash run nbMult file_out
```

Ce premier script exécute à la suite `generator` puis `product_v1` en gérant automatiquement le fichier de stockage des matrices pour le calcul. Ce fichier sera supprimé à la fin de l'exécution, seul restera le fichier contenant les résultats.

1.2.2 compare

```
$ bash compare nbMult file_out
```

Ce second script exécute `generator` puis `product_v1` et enfin `product_fair`, les deux exécutions de calculs de produits se font sur le même fichier pour en comparer les temps d'exécution (cf *Comparaisons*).

2 Comparaison entre les deux versions

Ces résultats ont été obtenu sur la machine *vador* de l'UBO qui possède 40 processeurs.

Nombre de multiplications	Taille maximum des matrices	TE product_v1	TE product_fair
50	50×50	4 s	6 s
50	100×100	7 s	12 s
200	20×20	2 s	2 s
200	50×50	23 s	25 s
5000	10×10	15 s	15 s
5000	20×20	43 s	48 s
50000	2×2	1 s	2 s
50000	4×4	10 s	12 s
50000	8×8	90 s	82 s
50000	16×16	281 s	299 s
50000	32×32	1741 s	1752 s

D'après les résultats obtenus, il semblerait que le programme qui réparti équitablement les processus sur les différents processeurs d'une machine ait un temps d'exécution plus long que le programme qui ne s'en occupe pas. Tous les résultats concorde sauf un, où les temps d'exécution sont les même, mais à trop petite échelle pour être représentatif.

Ces résultats peuvent être expliquer par le fait que comme chaque thread ne peut s'exécuter que sur un processeur défini pendant l'exécution, si celui si n'est pas disponible, le threads terminera son calcul plus tard. A l'inverse si l'on ne force pas les threads à s'exécuter sur un processeur, l'ordonnanceur les répartira en fonctions des disponibilités des processeurs.

3 Explication du code

Pour des besoins personnels, tous les commentaires et la documentation sont en anglais, pour pouvoir présenter mes différents projets personnels et scolaires sur plusieurs plateformes (site personnel, GitHub, ...). Je vais essayé d'expliquer au maximum le déroulement des programmes dans cette partie.

3.1 La structure `matrix`

La structure `matrix` à été implémentée pour manipuler facilement les matrices. Une matrice est définie par un tableau de tableaux de `double` contenant les valeurs de la matrice. Les champs `rows` and `cols` servent uniquement à garder le nombre de lignes (respectivement de colonnes) de la matrice.

```c
struct matrix {
    size_t rows;
    size_t cols;
    double ** data;
};
```

Le fichier `matrix.h` contient la définition de la structure ainsi que la définition de fonction utilitaires concernant les matrices :

- `matrix_init` Permet d'allouer la mémoire en fonction du nombre de lignes et de colonnes de la matrice.
- `matrix_set` Modifie une valeur d'une matrice à un certain indice.
- `matrix_get` Récupère une valeur d'une matrice à un certain indice.
- `matrix_print` Imprime une matrice sur la sortie standard.
- `matrix_fprint` Imprime une matrice dans un fichier.
- `matrix_destroy` Détruit la matrice en libérant la mémoire.

3.2 Génération de matrices

3.2.1 Initialisation

La génération de matrices nécessite deux paramètres à l'exécution : le nombre de produits à générer et le fichier où l'on écrira les produits.

Le programme commence par vérifier les arguments pour éviter des problèmes, puis alloue deux tableaux de matrices, le premier contiendra les matrice M_1 et le second les matrices M_2 tel que :

$$(M_{res})_i = (M_1)_i \times (M_2)_i$$

Ensuite, on ouvre le fichier en écriture avec `fopen` pour récupérer un `FILE *` car on peut facilement écrire dedans avec `fprintf`.

Enfin, deux choix s'offre à nous : créer les matrices ou les laisser être générées aléatoirement.

3.2.2 Création

Une contrainte est commune aux deux cas d'utilisation : pour pouvoir calculer le produit de deux matrices, il est nécessaire que le nombre de colonnes du premier facteur soit identique au nombre de lignes du second.

Création manuelle
Un dialogue s'installe entre le programme et l'utilisateur pour créer les matrices une par une :

1. Nombre de lignes de M_1 ?
2. Nombre de colonnes de M_1 ?
3. Valeurs de chaque $(M_1)_{(j,k)}$
4. Nombre de lignes de M_2 ?
5. Nombre de colonnes de M_2 ?
6. Valeurs de chaque $(M_2)_{(j,k)}$

Création automatique Le programme demande à l'utilisateur la taille maximum des matrices à générer, c'est-à-dire que les matrices générées pourront être de taille 1×1 jusqu'à $n \times n$ où n est le nombre entré par l'utilisateur.

L'édition des tailles et valeurs des matrices se fait de la même façon que pour la création manuelle, avec des tirages aléatoires d'entiers pour les tailles des matrices et des tirages de `double` pour les valeurs des $M_{(j,k)}$.

3.2.3 Enregistrement des matrices

Pour finir, le programme écrit les données qu'il à généré dans le fichier qu'il avait ouvert au préalable puis termine en nettoyant toutes les données stockées dans la mémoire.

3.3 Produit Matriciel

3.3.1 La structure `Calculation`

Cette structure contient tout ce qui est nécessaire au bon traitement des calculs.

- `state` représente l'état du programme :
  - `STATE_WAIT` : Le programme est en attente du premier calcul.
  - `STATE_CALC` : Le programme est en phase de calculs.
  - `STATE_PRINT` : Les calculs sont terminés, nous pouvons imprimer les résultats.
- `mutex` est `cond` sont le sémaphore et sa condition.

- `nbMult` est le nombre de produit à effectuer.
- `size` est le nombre de valeurs que contient la plus grosse matrice à traiter.
- `pendingCoefs` représente les calculs en attente.
- `m1` est le premier facteur d'un produit.
- `m2` est le second facteur d'un produit.
- `result` est la matrice résultat obtenue.

3.3.2 Calcul du nombre de threads

Le produit d'une $(m \times n)$-matrice et d'une $(n \times p)$-matrice donne une $(m \times p)$-matrice, il nous faudra donc $\max(m_i \times p_i)$ threads pour le calcul de la plus grosse matrice, c'est le rôle de la fonction `nbThread` de calculer ce nombre.

3.3.3 Lectures dans le fichier

`mmap` nous retourne un pointeur sur le premier caractère du fichier, on utilise donc ce pointeur pour lire chaque nombre qui compose le fichier. Cette fonction demande un `char **`, car ce pointeur servira également de curseur : il sera avancé en même temps que l'on lira les caractères.

La fonction utilise les valeurs ASCII des caractères pour parser un nombre. La fonction agit en deux étapes :

1. On avance le curseur jusqu'au début d'un nombre.
2. On lit le nombre.

Un nombre est composé de chiffres mais peut également être composé d'un "−" et/ou d'un ".". Pour savoir si un caractère appartient à un nombre ou pas, il suffit donc de vérifier si sa valeur ASCII vaut 45 pour le point, 46 pour le moins ou une valeur entre 48 et 57 inclus pour les chiffres.

Pour éviter de créer deux fonctions quasi-similaires, une pour les entiers et une pour les valeurs des matrices, j'ai choisi de caster en entier les nombres en sortie de la fonction au besoin.

3.3.4 La fonction des threads

La fonction des threads effectue le calcul d'un coefficient. Chaque thread reçoit en paramètre l'indice du coefficient duquel il devra s'occuper.

Cette fonction est composée d'une boucle `for` pour effectuer le calcul pour chaque produit à calculer.

Pour commencer, le thread est mis en attente de calcul. Pour qu'il soit débloqué, il est nécessaire que l'état de l'automate soit à `STATE_CALC` et que le `pendingCoef` de son indice soit à 1.

Lorsque le thread est débloqué, on commence par vérifier s'il a un calcul à effectuer en comparant son indice et les dimensions de la matrice résultat. Si celui-ci à un calcul a faire, on commence par transformer son indice en deux indice grâce à une astuce très simple : la division entière de l'indice par le nombre de colonnes de la matrice donne l'indice de la ligne concernée et le reste nous donne l'indice de la colonne.

Remarque : Il est nécessaire de recalculer les indices pour chaque produit car suivant les dimensions des matrices, un thread peut exécuter des calculs pour des coefficients différents à chaque itération.

Maintenant que le thread sais quel coefficient il doit calculer, il effectue le calcul :

$$M_{(i,j)} = \sum_{k=1}^{j} M1_{(i,k)} \times M1_{(k,j)}$$

La dernière partie de la fonction permet de bloqué les threads dont le calcul est terminé et de passer à l'affichage du résultat lorsque le dernier thread à fini son calcul. La valeur de retour n'est pas utilisée.

3.3.5 La fonction `main`

Le première instruction de la fonction main affecte le *timestamp* courant pour chronométrer le temps d'exécution du programme. Après, on vérifie les arguments ce qui permet d'éviter des problèmes, des erreurs de segmentation entre autre si des arguments sont mal renseignés.

On initialise le sémaphore et la condition puis on récupère la taille du fichier contenant les produits à effectuer pour ensuite le mapper en mémoire. Après cela, on récupère le nombre de produits pour allouer les tableaux de matrices.

Le seul moyen de calculer le nombre maximum de threads nécessaires à l'exécution du programme est de calculer les dimensions de la plus grande matrice résultat. C'est pour cette raison que j'ai choisis d'hydrater deux tableaux de matrices, plutôt que de sauvegarder uniquement les offset car tant qu'à parcourir tout le contenu du fichier, autant récupérer directement les données dont nous aurons besoin.

Ensuite, on alloue et initialise tout ce qui ne l'a pas encore été grâce aux données que l'on vient de récupérer.

Pour chaque calcul, on ré-initialise les matrices de calculs et de résultat, les calculs en attente ... On donne la main au threads puis on attend qu'ils aient tous terminés. On affiche le résultat dans le fichier choisi par l'utilisateur.

Quand tous les calculs sont terminés, on attend la terminaison de tous les threads puis on libère les ressources.

Pour finir, on calcul le temps d'exécution et on l'affiche sur la sortie standard.

3.3.6 Répartition forcée sur les processeurs

Le second programme se déroule comme l'autre à une différence près : chaque thread s'exécute sur un seul et unique processeur pendant l'exécution du programme.

On observera donc cet ajout dans la fonction de calcul :

```
    int cpu;
    cpu_set_t ensemble;
    cpu = index % nbhocks;
    CPU_ZERO(&ensemble);
    CPU_SET(cpu, &ensemble);
    sched_setaffinity(0, sizeof(cpu_set_t), &ensemble);
```

Sachant que `nbhocks` est déclaré global pour être utilisé par tous les threads et est affecté dans la fonction `main` par :

```
nbhocks = (int) sysconf(_SC_NPROCESSORS_ONLN);
```

On à observé à l'exécution (se référer à Comparaison) que ce programme met toujours plus de temps à s'exécuter que celui ci-dessus.