PG305 TP – Craquage de mot de passe

ENSEIRB-MATMECA, PRCD S9 2014-2015

Qing LIANG

1. Introduction

Ce TP est pour but de construire un programme en l'exécution parallèle à craquer le mot de passe par manière de force-brute. Le programme est implémenté en utilisant MPI et openMP. L'idée principale de l'implémentation est décrit aux sections de la suite, puis on discute quelques points très importants et quelques problèmes s'enlèvent pendant l'implémentation. On termine par une bref analyse de performance de ce programme.

2. Enumération de mot de passe et génération de liste de tâches

2.1 L'intervalle de recherche

Une fois l'alphabet est donné, un mot de passe possible est donc une permutation de letters dans l'alphabet. Pour rechercher de mot de passe par manière force-brute, le thread de travail doit énumérer tous les mots de passe possibles et comparer avec le mot de passe donné. Un problème que on doit considérer est que le nombres de processeurs et threads on peut allouer est très petit par rapport au nombre de toutes les permutations possibles des lettres dans un alphabet donné qui peut facilement être très très grand dans le contexte réel. Pour définir l'intervalle de recherche, il est raisonnable que le processus Master indique uniquement les bornes de permutations de chaque tâche. On a donc besoin d'une stratégie d'énumérer les permutations et la segmentation d'intervalle.

En supposant que l'alphabet est composé de 5 lettres: {a, b, c, b, d} et l'ordre de chaque lettre dans l'alphabet est fixé une fois l'alphabet est initialisé. Donc l'index de lettre correspondant dans l'alphabet est {1, 2, 3, 4, 5}. On fait l'énumération de mot de passe possible sur l'ordre suivante: (vue par column)

a(1)	*aa(6)*	*ba(11)*	*ca(16)*	*da(21)*	*ea(26)*	*aaa(31)*	*aba(36)*	…
b(2)	*ab(7)*	*bb(12)*	*cb(17)*	*db(22)*	*eb(27)*	*aab(32)*	*abb(37)*	…
c(3)	*ac(8)*	*bc(13)*	*cc(18)*	*dc(23)*	*ec(28)*	*aac(33)*	*abc(38)*	…
d(4)	*ad(9)*	*bd(14)*	*cd(19)*	*dd(24)*	*ed(29)*	*aad(34)*	*abd(39)*	…
e(5)	*ae(10)*	*be(15)*	*ce(20)*	*de(25)*	*ee(30)*	*aae(35)*	*abe(40)*	…

En fait l'ordre d'énumération est exactement comme on compte l'entier dans un système de numération positionnel en base du nombre des lettres dans l'alphabet, sauf que l'entier le plus petit est 1 qui correspond la première lettre. Ainsi chaque mot de pass possible on peut le représenter par un unique entier (comme l'entier dans la parenthèse indiqué à l'exemple ci-dessus) dans un telle système de numération positionnel et faire la transformation entre les deux.

L'implementations de l'algorithme **string-to-int** est décrit par le code ci-dessous:

```
unsigned long long str2int(char* str, char* alphabet, int alphab_len) {
    int i = strlen(str);
    int tmp = i;
    unsigned long long sum = 0;
    while(i>0){
        sum += getCharIndex(alphabet, &str[i-1]) *
                (unsigned long long)pow((double)alphab_len, (double)(tmp-i));
        --i;
    }
```

```
    return sum;
}
```

Et l'inverse int-to-string:

```
void int2str(unsigned long long n, char* chr, char* alphabet, int alphab_len) {
    int len = getStrLen(n, alphab_len);
    int rem;
    int i = len - 2;
    while(i>=0){
        rem = n % alphab_len;
        chr[i] = alphabet[(rem + alphab_len - 1) % alphab_len];
        if(rem == 0)
            rem = alphab_len;
        n = (n - rem) / alphab_len;
        --i;
    }
    chr[len-1] = '\0';
}
```

Ayant un telle stratégie d'énumération, la segmentation de l'intervalle va être très simple. L'intervalle de recherche envoyé vers le processus slave par le Master est toujours un fragment de permutation avec un entier indiquant le début et un entier indiquant le nombre de permutations dans cet intervalle. Il est défini par un struct ci-dessous:

```
typedef struct {
    unsigned long long start;       // start position number of the permutation
    unsigned long long num_perms;   // number of permutations in the intervalle
} Intervalle;
```

Le processus Master possède un curseur unsigned long long start qui stocke toujours l'entier qui présente le début de permutations restant. Si il y'a pas de permutations restant, start est mis à 0. Chaque fois quand un processus slave demande un intervalle, le Master va générer un intervalle à partir des permutations restant et mettre à jour le curseur start, si il y a encore des permutations à distribuer. La longueur d'intervalle est défini à

```
Intervalle_LENGTH = num_AllPerms / (num_procs*2);
```

2.2 Génération de liste de tâches

Le thread de communication dans le processus Slave garde une chaîne de tâches. La tâche est défini par un struct ci-dessous:

```
typedef struct Task {
        unsigned long long start;  // start position number of the permutation
        unsigned long long end;    // end position number of the permutation
        struct Task* next;         // pointer to next task in the list
} Task;
```

Chaque fois quand le processus Slave reçoit un intervalle de recherche à partir du Master, il invoque la méthode void updateTaskList(Intervalle* itv) qui va découper l'intervalle reçu dans certaines fragments de permutations que nous appelons les tâches pour le thread de travail et les ajouter à partir de la fin de la chaîne de tâches. Par contre, le thread de travail traiter la tâche de recherche une par une en manière d'itération. Au début de chaque itération, il va récupérer une

tâche à partir de la chaîne de tâches en invoquant la méthode Task* nextTask(). La longueur de tâche est présentée par TASK_SIZE qui est prédéfini à 10000 dans mes codes sources. Deux éléments listHeader et listTail de type Task* sont définis pour stocker l'adresse de première tâche et la dernière respectivement, qui sont utilisé pour la mis à jour de la chaîne et sa libération de mémoire.

---

3. Communication entre Master et Slave

Sauf que la partie de recherche de mot de passe en parallel par les threads de travail générés par openMP dans les processus Slaves, le thread de communication de chaque processus Slave est chargé de communiquer avec le processus Master. Cette communication est fait principalement par les méthodes MPI_Send et MPI_Recv avec 7 types de messages représentés par 7 tags dans la fichier mpi_utils.h décrits ci-dessous:

```
#define TAG_PASSWORD          0   // Slave à Master
#define TAG_INTERVAL          1   // Master à Slave
#define TAG_INTERVAL_REQUEST  2   // Slave à Master
#define TAG_KILL_SELF         3   // Master à Slave
#define TAG_NO_MORE_INTERVAL  4   // Master à Slave
#define TAG_PWD_NOT_FOUND     5   // Slave à Master
#define TAG_CONFIRM_EXIT      6   // Slave à Master
```

Pour savoir le détail de protocole de communication entre les deux parties, veuillez voir les codes sources.

---

4. Quelques points à faire attention

4.1 La largeur d'entier

Comme on a parlé dans la section 2.1, le nombre de permutations possibles défini le code par l'entier num_AllPerms peut facilement être très très grand. On doit le définir du type unsigned long long, qui a une largeur de 8 bits avec la valeur de 0 à 18,446,744,073,709,551,615. Mais si le nombre de lettres dans l'alphabet (par exemple 256) est assez grand et la longueur maximale de mot de passe à chercher est assez grand (par exemple 12), la valeur de num_AllPerms va dépasser la borne de unsigned long long ainsi le programme va marcher pas. C'est un bug dans ce programme.

4.2 Assurer la réception de message

En sachant que le processus Master et Slave écoute les messages vers sa propre part par une boucle de vérification de flag à la méthode non-bloquant MPI_Iprobe, if faut assurer que chaque fois un message est arrivé, il faut invoquer la méthode MPI_Recv pour recevoir ce message et mettre a jour du flag à 0. Sinon le flag va rester toujours à 1. Ainsi avant un autre message arrive, le processus va penser que il y a toujours un message est arrivé dans chaque itération avec le même type de message reçu précédemment. C'est qui va bloquer ou provoquer les problèmes très bizarres pendant l'exécution de programme.

4.3 Protection de données à accéder en concurrence dans openMP

Puisque la chaîne de tâches dans le processus Slave va probablement être accédé en occurrence par plusieurs threads de travail et le thread de communication, il faut assurer que elle est accédé exclusivement. Pour le faire, les méthodes updateTaskList et nextTask doivent être protégé par un bloc critique respectivement en déclarant #pragma omp critical {…}.

4.4 L'ordre de termination de processus Master et Slave

C'est un problème qui m'a encombré. Dans ce programme, l'exécution du processus Slave se termine uniquement quand le Slave reçoit le message avec tag TAG_KILL_SELF à partir du Master, n'importe qu'il a trouvé le mot de passe ou tous ses threads de travail s'arrêtent sans trouvant le mot de passe. Puis le slave envoie un message de TAG_CONFIRM_EXIT au Master pour déclarer sa fin de l'exécution.

4.5 Evider de réitérer à envoyer le message sans avoir de réponse

Cette situation concerne principalement le message qui va être envoyé dans une certaine condition. Par exemple le message avec tag TAG_INTERVAL_REQUEST va être envoyé par le Slave à Master quand la condition que le nombre de tâches dans la liste est inférieur à le nombre de thread de travail est satisfait. En attente de la réponse de Master sur cette requête, le slave va continuer à renvoyer le même requête au Master puisque la condition est toujours satisfait. Cette réitération d'envoyer de message va probablement provoquer le débordement de buffer de réception dans Master et le Master va prendre beaucoup de temps à traiter cette suite de messages quand même, probablement est bloqué. Pour évider cette situation, je mets un flag send initialisé à 0 et puis j'en mets à 1 après l'envoi pour chaque ce type de message. Après la réception de réponse de ce message, le flag est mis à 0. Le message va être envoyé uniquement quand la condition est satisfait et le flag est 0.

5. Performance

Le teste de performance de ce programme est fait sur PlaFRIM. Le nombre de lettres dan l'alphabet est 26 et la longueur maximal de mot de passe à tester est 8. Les résultats du temps passé en lancement du programme au nombre de slaves de 1 à 8 et le nombre de threads de travail dans chaque slave de 1 à 12 indique que, le temps passé minimal se trouve souvent à un faible nombre de processus slave de 2 à 3, avec le nombre de threads de travail de 10 à 20 environnement. Quand le nombre de processus Slave augmente, il va prendre plus de temps parce que le nombre de messages passés entre les Slaves et le Master va augmenter également, ainsi le coût de communication va être beaucoup plus cher.

En fait la performance dépend également à la taille de intervalle et la taille de tâche pour chaque thread de travail. Une trop petite taille de l'intervalle ou la tâche par rapport au nombre de permutations à essayer implique une augmentation de messages transférés entre les Slaves et le Master dont le coût de communication est beaucoup plus sur les mémoires distribuées.

A part du nombre de Slaves et du nombre de threads de travail, le temps passé dépend largement à la position de correct mot de passe dans tous les intervalles. Parce que dans cette réalisation de programme, les intervalles à distribuer sont découpés en linéaire. Si le mot de passe correct est dans le dernière intervalle, ça va être le cas au pire.

6. Conclusion

Le programme marche très bien sur PlaFRIM et un ordinateur local avec l'environnement de MPI et openMP installés. Une idée pour élever la performance est peut être trouver une stratégie de découper et distribuer les intervalles aléatoirement pour élever la possibilité de distribuer l'intervalle qui contient le correct mot de passe le plus tôt possible à un processus Slave.