HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

HIGH PERFORMANCE CODING
HPC

THREADING BUILDING BLOCKS

João Miguel Domingues Pedrosa
Loïc Haas

5 juin 2016

Table des matières

1 Introduction

L'objectif de ce projet est de comprendre, essayer et exploiter un outil d'optimisation. Dans notre cas, nous avons choisi TBB (Threading Building Blocks). Il s'agit d'une libraire d'Intel fait pour du C++.

Sa caractéristique est de simplifier au maximum l'implémentation de programme parallèle pour des systèmes multicœur. Le développeur pourra ainsi faire un programme portable car c'est la librairie qui va se charger d'utiliser la bonne implémentation de thread (exemple : POSIX pour linux ou les threads Windows). Pour cela, elle met en place différentes fonctions et classes lié à la programmation parallèle et à la gestion de concurrence.

Pour ce projet, nous avons du faire l'installation de la libraire, la procédure sera expliqué plus loin dans le document. Il y aura aussi un code exemple auquel on aura fait des mesures de performances afin de voir les optimisation apporté. Nous finirons par une analyse de notre constats tout au long de nos essaies.

2 Installation

L'installation des libraires a été fait sur des machines possédant un OS Linux Ubuntu. Les installations suivantes ont été fait via ligne de commande.

2.1 Installation via gestionnaire de package

Pour installer sur nos machines, nous avons utilisé le gestionnaire de package `apt-get`. Le package installé est `libtbb-dev et libtbb2`. Nous avons donc la commande suivante pour installer :

```
sudo apt-get install libtbb-dev libtbb2
```

Une fois installé, on peut commencer à programmer avec les librairies pthreads. Pour tester l'installation, on va compiler et exécuter un code exemple afin de voir s'il n'y a pas de problème.(commande pour la compilation `g++ <filename>.cpp -ltbb`)

2.1.1 Code exemple

```
1  //src: https://software.intel.com/articles/1-minute-intro-code-samples
2
3  #include <cmath>
4  #include "tbb/tbb.h"
5
6  double *output;
7  double *input;
8
9
10 int main() {
11     const int size = 20000000;
12     output = new double[size];
13     input = new double[size];
14
15     for(int i = 0; i < size; i++) {
16         input[i] = i;
17     }
18
19     tbb::parallel_for(0, size, 1, [=](int i) {
20
21         output[i] = sqrt(sin(input[i])*sin(input[i]) + cos(input[i])*cos(input[i]));
22
23    });
24     delete[] input;
25     delete[] output;
26     return 0;
27 }
```

2.2 Installation via l'archive

1. Il faut premièrement télécharger. Pour cela, il faut se rendre sur le site d'Intel pour TBB et aller dans la section download [1]. Ensuite, on copie l'adresse sur la release et l'OS que l'on veut (dans notre cas Linux). On entre ensuite la commande suivante :
   `$ wget https://www.threadingbuildingblocks.org/sites/default/files/software_releases/linux/tbb<version>.tgz`
2. Maintenant que l'on a télécharger l'archive, il faut l'extraire pour cela on fait la commande suivante :
   `$ tar zxf tbb<version>.tgz`
3. Un fois extrait, il faut se rendre dans le dossier `bin` et modifier le script `tbbvars.sh`. Il faut changer la ligne suivante :
   `TBBROOT = SUBSTITUTE_INSTALL_DIR_HERE`
   par
   `TBBROOT = <path_tbb_directory>`
   Cela permettra alors de pouvoir utiliser la librairie facilement lors de la compilation.
4. Après, on exécute le script avec la bonne option (ia32 pour une architecture 32 bits ou intel64 pour du 64 bits) et on peut commencer à faire des programmes avec TBB.
5. Pour tester si tout a été bien installé, on compile le code précédent et on l'exécute. Si il n'y a pas eu de problème durant la compilation et l'exécution c'est que tout est bon

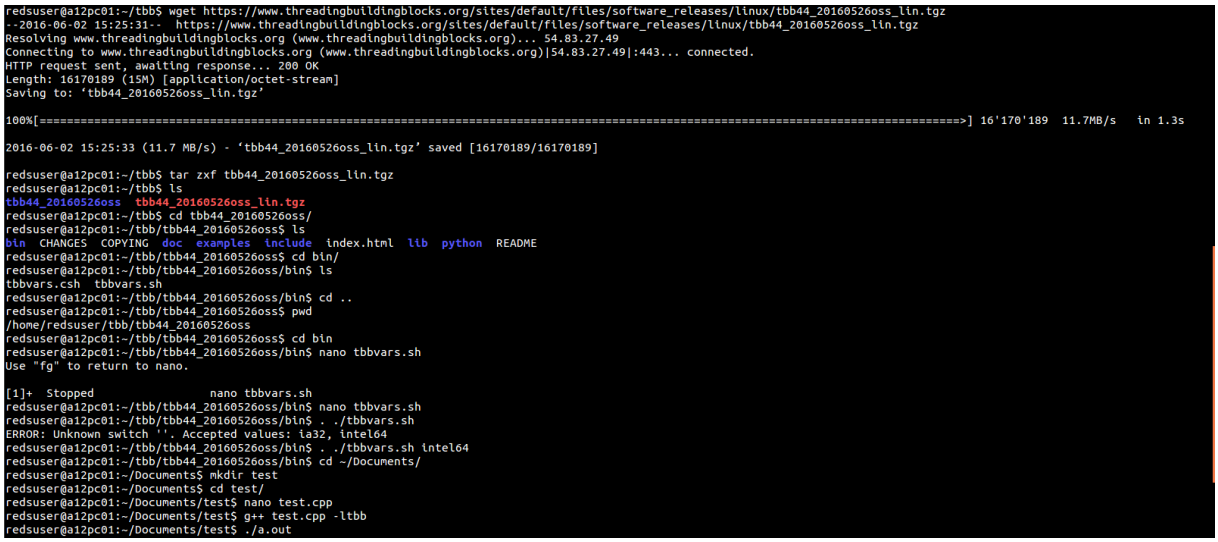

FIGURE 1 – Capture d'écran des commandes utilisées

1. `https://www.threadingbuildingblocks.org/download`

3 Exemple

Le programme utilisé pour l'exemple s'occupe de compter le nombre d'occurrence des caractères de la table ASCII.Nous avons choisi le code suivant car il se prête bien à la parallélisation et que nous avons déjà fait une implémentation parallèle avec les thread POSIX directement et en C. Cela nous permet ainsi de faire des meilleurs critiques au niveau des performances.

Commande pour la compilation : `g++ -O3 -std=c++11 statscount.cpp -o statscount -ltbb`

3.1 Code

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <numeric>

#include <tbb/tbb.h>
#include <cstring>

class StatsCount {
public:
    StatsCount(const std::string& filename, const size_t bufferSize = 512) :
        fileName(filename), BUFFER_SIZE(bufferSize), result(256, 0) { }

    void run() {
        // Thread Local storage for characters counts
        tbb::combinable<std::vector<size_t>> nbChars([](){return std::vector<size_t>(256, 0);});

        // thread local storage for ifstream
        // For some reason gcc 4.8 cant compile if we use referances instated
        tbb::combinable<std::ifstream*> files([this](){
            return new std::ifstream(fileName, std::ios::binary | std::ios::in);
        });

        // Get the file size
        std::ifstream file (fileName, std::ios::ate | std::ios::binary | std::ios::in);
        if (!file.is_open()) {
            std::cerr << "Can not open file : " << fileName << std::endl;
            return;
        }
        const size_t FILE_SIZE = (size_t)file.tellg();
        std::cout << "File size is " << FILE_SIZE << std::endl;

        // Calculate the nomber of iteration required
        const size_t NB_ITERATIONS = FILE_SIZE / BUFFER_SIZE +
            ((FILE_SIZE % BUFFER_SIZE) != 0 ? 1 : 0);


        // Process the counting algo in parallel
        tbb::parallel_for(tbb::blocked_range<size_t>(0, NB_ITERATIONS, 2),
            [&nbChars, &files, this](const tbb::blocked_range<size_t> &r) {
                std::ifstream* file = files.local();
                if (!file || !file->is_open()) {
                    return;
                }

                // Place the read cursor to the good position
                file->seekg(r.begin() * BUFFER_SIZE, std::ios::beg);

                // Get a ref to the localstorage for this thread
                std::vector<size_t>& chars = nbChars.local();

                // Count all characters
                uint8_t* buffer = new uint8_t[BUFFER_SIZE];

                for (size_t i = r.begin(); i != r.end(); i+=r.step()) {
                    std::streamsize readed = file->readsome((char*)buffer, BUFFER_SIZE);
                    while(readed--) {
                        chars[buffer[readed]]++;
                    }
                }

                delete[] buffer;
            }
        );

        // Combine the result of all thread in the result
        nbChars.combine_each([this](const std::vector<size_t> &v){
```

```
69              for (size_t i = 0; i < result.size(); ++i) {
70                  result[i] += v[i];
71              }
72          });
73
74          // Close all opened files
75          files.combine_each([this](std::ifstream* file){
76              file->close();
77              delete file;
78          });
79      }
80
81      void print() {
82          for (size_t i = 0; i <= result.size(); ++i) {
83              if (isprint((int)i)) {
84                  std::cout << "'" << (char)i << "' : " << result[i] << std::endl;
85              }
86          }
87
88          size_t total = std::accumulate(result.begin(), result.end(), 0);
89          std::cout << std::endl << "Total char in file : " << total << std::endl;
90      }
91
92  private:
93      const std::string fileName;
94      const size_t BUFFER_SIZE;
95      std::vector<size_t> result;
96  };
97
98  int main(int argc, char *argv[]) {
99      if (argc != 2) {
100         std::cerr << argv[0] << " filename" << std::endl;
101         return -1;
102     }
103     StatsCount stats(argv[1]);
104     stats.run();
105     stats.print();
106     return 0;
107 }
```

3.1.1 Explication

La libraire TBB possède beaucoup de fonction permettant le parallélisation. Pour expliquer le fonctionnement, nous allons montrer comment fonctionne une d'elle, le `paralle_for`.

Cette fonction possède plusieurs entête différents qui varie selon ce qu'on a besoin. Ici, on utilise une qui demande une classe template `blocked_range` et une expression lambda. L'expression lambda correspondra à la fonction que devront effectuer chaque thread lancé par le programme. Ici, il s'agit de compter le nombre d'occurrences de chaque caractère dans un fichier.
La classe `blocked_range` permet la répartition de l'exécution du programme dans tout les cœurs du système. Il faut lui indiquer dans le constructeur la range d'itérations qu'il y a maximum à faire, 0 à n-1. Cela permettra dans les boucles for interne, ou autre calcul nécessitant la connaissance des index, d'utiliser le `begin` et `end` afin de pouvoir bien répartir les données. Cela est utile quand certaine partie prenne plus de temps que d'autre. Ainsi, lorsqu'un thread se termine on lance directement dessus le jeux de données suivant. Cela abstrait la répartition des données manuel dans chaque thread et l'optimisant sachant que l'on attend sur le thread le plus rapide. Dans notre exemple, on utilise se procédé pour mettre le pointeur de fichier au bonne endroit selon les threads :

```
1                   return;
2               }
```

On l'utilise aussi dans la boucle for qui va lire le fichier :

```
1               // Count all characters
2               uint8_t* buffer = new uint8_t[BUFFER_SIZE];
3
4               for (size_t i = r.begin(); i != r.end(); i+=r.step()) {
5                   std::streamsize readed = file->readsome((char*)buffer, BUFFER_SIZE);
6                   while(readed--) {
```

Une autre variante de `parallel_for` exige comme paramètre un min, un max, un step et une fonction (ou expression lambfa). Celle-ci diffère de l'autre par le paramètre step. Il s'agit au fait du nombre de pas qui sépare chaque itérations de thread. Par exemple, on veut itère de 0 à n-1 mais on veut le faire que sur les nombres pair, donc un sur deux. Il suffit de mettre un step de 2 et on aura la réaction voulu. La répartition des tâches sera la même que le `parallel_for` discuté avant mais on aura moins d'itérations et l'index fournit sera calculer selon ce paramètre. Exemple :
Série :

```
for(int i=0; i < N-1; i+=2){
   tabPair[i/2] = i;
}
```

Parallèle TBB :

```
parallel_for(0, N-1, 2, [tabPair](const int& i) {
   tabPair[i/2] = i;
});
```

Ensuite, pour récolter le nombre d'occurrences dans chaque de caractères que possède chaque thread, on passe par une classe template combine. Cette classe demande une expression lambda au constructeur :

```
        tbb::combinable<std::vector<size_t>> nbChars([](){return std::vector<size_t>(256, 0);});
```

Cela va générer un objet du type voulu lors du lancement du thread. On pourra le récupérer à chaque fois que l'on appelle la méthode `local` de la classe, on renvoi un vecteur d'une certaine taille dans notre cas :

```
                std::vector<size_t>& chars = nbChars.local();
```

Il va s'agir ici de Thread Local Storage. Cela veut dire que tout les threads possède le même objet qui se trouve à la même plage d'adresse mais qui est indépendant pour chaque thread et donc que son contenu est différent. Dans notre cas, on aimerait qu'à la fin, on est un seul tableau et non plusieurs contenant le nombre total d'occurrences pour chaque caractère. Pour cela, une fois l'exécution du `parallel_for` terminé, on appelle la méthode `combine_each` et on lui passe une expression lambda récupérant le nombre d'occurrences rencontré pour chaque thread dans un seul tableau :

```
        // Combine the result of all thread in the result
        nbChars.combine_each([this](const std::vector<size_t> &v){
            for (size_t i = 0; i < result.size(); ++i) {
                result[i] += v[i];
            }
        });
```

La même chose est fait pour la répartition du fichier. On applique un `close` en fin d'exécution :

```
        // Close all opened files
        files.combine_each([this](std::ifstream* file){
            file->close();
            delete file;
        });
```

3.2 Équivalent C

Compilé avec la commande suivante : `gcc -std=c11 -O3 statscount.c -o statscountposix -lpthread`

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#include <assert.h>

#define NB_ASCII_CHARS (256) /* Taille de la table ASCII (facilite l'indexage)*/
#define NB_OCC_CHARS    62 /* Nombre de caractère dont l'occurence doit être vérifié*/

#define MAX_BUFFER 256
#define NB_UPPER_CASE 26
#define NB_LOWER_CASE 26
#define NB_NUMBERS    10

#define NB_THREAD 4

struct char_stats {
    char *path;
    uint32_t size;
    char tab_occ[NB_ASCII_CHARS];
};

struct thread_data{
    char *path;
    uint32_t start; /* position de départ du bloc à traiter*/
    size_t size;
};

void* char_count(void* args) {

    size_t nb_read;
    uint8_t buffer[MAX_BUFFER];
    uint8_t *occ;
    struct thread_data *data = (struct thread_data *) args;

    FILE *fp = fopen(data->path, "r");
    if (fp == NULL) {
        pthread_exit(NULL);
    }

    occ = calloc(NB_ASCII_CHARS, sizeof *occ);

    fseek(fp, data->start, SEEK_SET);

    // Trouve le minimum entre buffer et taille à lire (bits trick)
    size_t size_to_read = data->size ^ ((MAX_BUFFER ^ data->size) & -(MAX_BUFFER < data->size));

    /* Traite le bloc de mémoire assigné */
    while((nb_read = fread(buffer, 1, size_to_read, fp)) != 0){
        for(size_t i=0; i < nb_read; i++){
            occ[buffer[i]]++;
        }

        // Trouve le minimum entre buffer et taille restent à lire (bits trick)
        // Si on arrive à la fin, size_to_read vaudra 0
        data->size -= size_to_read;
        size_to_read = data->size ^ ((MAX_BUFFER ^ data->size) & -(MAX_BUFFER < data->size));
    }

    fclose(fp);

    pthread_exit(occ);
}

void print_occurences(struct char_stats *stats){
    for(int i = 0; i < NB_UPPER_CASE; i++){
        printf("%c : %u\n", 'a' + i, stats->tab_occ['a' + i]);
        printf("%c : %u\n", 'A' + i, stats->tab_occ['A' + i]);
    }

    for(int i = 0; i < NB_NUMBERS; i++){
        printf("%c : %u\n", '0' + i, stats->tab_occ['0' + i]);
    }
}

struct char_stats *stats_init(const char *path)
{
```

```
80      FILE *fp = fopen(path, "r");
81
82      /* Vérifie que le fichier existe */
83      if(fp == NULL){
84          return NULL;
85      }
86
87      /* Retrouve la taille du fichier */
88      fseek (fp , 0 , SEEK_END);
89      size_t size = ftell (fp);
90
91      fclose(fp);
92
93      struct char_stats *stats;
94
95      /* Alloue de l'espace pour la structure à rendre*/
96      stats = calloc(1,sizeof *stats);
97      /* Vérifie que la mémoire a bien été alloué*/
98      if(stats == NULL){
99          return NULL;
100     }
101
102     stats->size = size;
103     size_t lenght = strlen(path) + 1;
104     stats->path = malloc(lenght);
105     memcpy(stats->path, path, lenght);
106
107     return stats;
108
109 }
110
111 size_t stats_count(struct char_stats *stats)
112 {
113     assert(stats != NULL);
114     size_t size_per_thread;
115     size_t rest;
116
117     pthread_t threads[NB_THREAD];
118     struct thread_data datas[NB_THREAD];
119     char *occ_thread;
120
121     memset(stats->tab_occ, 0, NB_ASCII_CHARS);
122
123     size_per_thread = stats->size / NB_THREAD;
124     rest = stats->size % NB_THREAD;
125
126     /* Prépare les threads avec les bloc mémoire à traiter */
127     datas[NB_THREAD-1].path = stats->path;
128     datas[NB_THREAD-1].start = (NB_THREAD-1) * size_per_thread + rest;
129     datas[NB_THREAD-1].size = size_per_thread;
130     pthread_create(&threads[NB_THREAD-1], NULL, char_count, &datas[NB_THREAD-1]);
131
132     for(int i = NB_THREAD-2; i >= 0; i--){
133         datas[i].path = stats->path;
134         datas[i].start = i * size_per_thread;
135         datas[i].size = size_per_thread;
136         pthread_create(&threads[i], NULL, char_count, &datas[i]);
137     }
138
139     for(int i = 0; i < NB_THREAD; i++){
140         pthread_join(threads[i], (void**)&occ_thread);
141
142         // Vérifie que le pointeur retourné est non null
143         if(occ_thread == NULL) continue;
144
145         // Met à jour le tableau des occurences
146         for(unsigned char j = 'a'; j <= 'z'; j++){
147             stats->tab_occ[j] += occ_thread[j];
148         }
149
150         for(unsigned char j = 'A'; j<= 'Z'; j++){
151             stats->tab_occ[j] += occ_thread[j];
152         }
153
154         for(unsigned char j = '0'; j <= '9'; j++){
155             stats->tab_occ[j] += occ_thread[j];
156         }
157         free(occ_thread);
158     }
159     return NB_OCC_CHARS;
160 }
161
162 void stats_clear(struct char_stats *stats)
163 {
164     free(stats->path);
```

```
165     free(stats);
166 }
167
168 int main(int argc, const char* argv[]){
169
170     struct char_stats *stats;
171     uint32_t nb_count;
172
173     if(argc < 2){
174         printf("parameter: <filepath>\n");
175         exit(1);
176     }
177
178     if((stats = stats_init(argv[1])) == NULL){
179         printf("Fail at initialisation\n");
180         exit(1);
181     }
182
183     stats_count(stats);
184
185     print_occurences(stats);
186
187     stats_clear(stats);
188
189     return 0;
190 }
```

3.3 Remarque

On voit que la version C++ TBB est plus élégante que la version C POSIX. Dans la version C, il faut préparer tout les threads leur définir la taille de données, où sa commence et où il faut s'arrêter. De plus, c'est nous qui devons choisir nous même le nombre de thread. Alors que TBB, lui, choisi tout seul le nombre de thread optimum à lancé.

3.4 Mesures

Les mesures ont été effectué sur un fichier d'environ 250 MB. On a utilisé l'utilitaire `perf` afin de relever des données intéressante sur les performances de notre implémentation TBB.
Commande utilisé : `perf stat -e task-clock,cycles,instructions,cache-references,cache-misses <exec> <filepath>`

3.4.1 TBB

```
Performance counter stats for './statscount ./text.txt':

        459.565934      task-clock (msec)         #    3.458 CPUs utilized
    1'395'613'235       cycles                    #    3.037 GHz
    1'576'580'726       instructions              #    1.13  insns per cycle
        3'095'897       cache-references          #    6.737 M/sec
        1'210'620       cache-misses              #   39.104 % of all cache refs

      0.132899179 seconds time elapsed
```

3.4.2 POSIX

```
Performance counter stats for './statscountposix ./text.txt':

        488.637225      task-clock (msec)         #    3.239 CPUs utilized
    1'511'952'760       cycles                    #    3.094 GHz
    1'709'169'704       instructions              #    1.13  insns per cycle
        4'169'408       cache-references          #    8.533 M/sec
        1'215'851       cache-misses              #   29.161 % of all cache refs

      0.150841692 seconds time elapsed
```

4 Analyse

On peut qu'entre la version TBB et POSIX, la différence est moindre. On a juste quelque problème au niveau de la cache avec TBB mais on arrive pas comme ça à identifier d'où vient le problème. Par contre, on voit bien que les CPUs sont bien utilisé dans les deux cas. On a plus que 3 CPU utilisé durant l'exécution du programme. On peut donc dire que TBB fait bien sont travaille et parallélise notre implémentation.

5 Conclusion

Au final, on voit que la librairie TBB fait bien sont travaille. Elle repartit la tâche dans plusieurs threads et dans tout les coeurs sans même avoir à demander combien de thread on veut lancer. Il exploite ainsi tout le potentiel de processeur sans que le développeur est à se soucier combien de thread il doit lancer. On a donc une bonne abstraction de la part de la librairie avec des performances satisfaisante qui permet au développeur de faire du code simple, compréhensible et générique, sachant que la libraire s'adapte tout seul à l'implémentation des threads selon l'OS ou l'architecture utilisé.

6 Bibliographie

— Tutoriel d'Intel : `https://www.threadingbuildingblocks.org/intel-tbb-tutorial`
— Documentation d'Intel : `https://software.intel.com/en-us/tbb-documentation`
— Présentation d'Intel : `http://www.cs.cmu.edu/afs/cs/academic/class/15499-s09/www/handouts/TBB-HPCC07.pdf`