



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)



## HIGH PERFORMANCE CODING HPC

---

# THREADING BUILDING BLOCKS

---

João Miguel Domingues Pedrosa  
Loïc Haas

5 juin 2016

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
2.1	Installation via gestionnaire de package . . . . .	3
2.1.1	Code exemple . . . . .	3
2.2	Installation via l'archive . . . . .	4
<b>3</b>	<b>Exemple</b>	<b>5</b>
3.1	Code . . . . .	5
3.1.1	Explication . . . . .	6
3.2	Équivalent C . . . . .	8
3.3	Remarque . . . . .	10
3.4	Mesures . . . . .	10
3.4.1	TBB . . . . .	10
3.4.2	POSIX . . . . .	10
<b>4</b>	<b>Analyse</b>	<b>11</b>
<b>5</b>	<b>Conclusion</b>	<b>11</b>
<b>6</b>	<b>Bibliographie</b>	<b>11</b>

# 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 librairie 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 librairie, 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 essais.

## 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 `libtbb2`. Nous avons donc la commande suivante pour installer :

```
sudo apt-get install libtbb2
```

Une fois installé, on peut commencer à programmer avec les librairies `pthread`. 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<sup>1</sup>. 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 téléchargé l'archive, il faut l'extraire pour cela on fait la commande suivante :

```
$ tar xzf 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
```

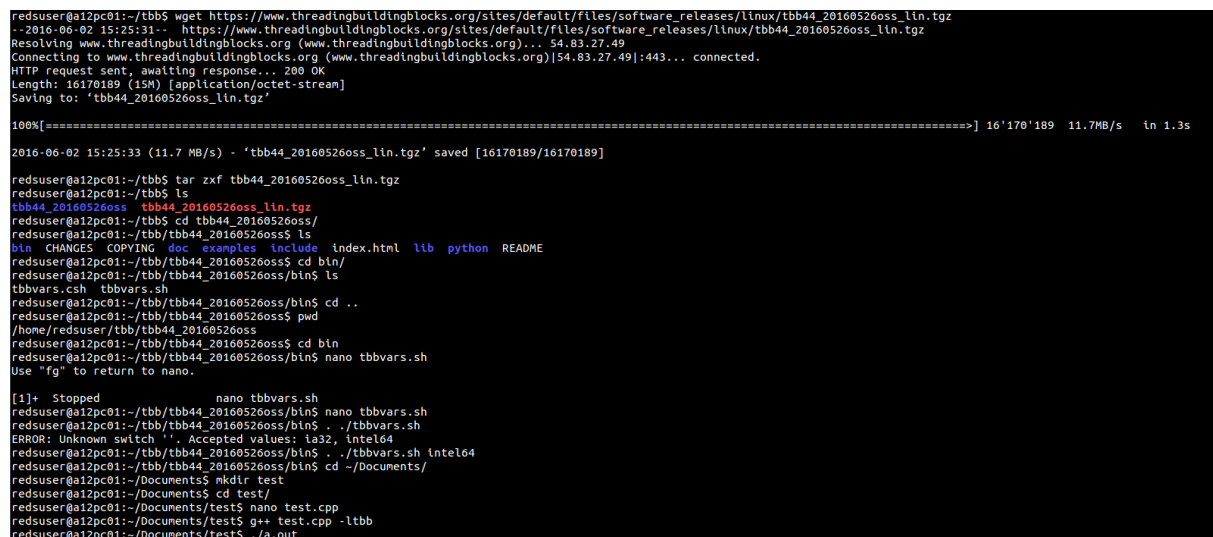
par

```
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



```

reduser@ai2pc01:~/tbb$ wget https://www.threadingbuildingblocks.org/sites/default/files/software_releases/linux/tbb44_20160526oss_lin.tgz
--2016-06-02 15:25:31-- https://www.threadingbuildingblocks.org/sites/default/files/software_releases/linux/tbb44_20160526oss_lin.tgz
Resolving www.threadingbuildingblocks.org (www.threadingbuildingblocks.org)... 54.83.27.49
Connecting to www.threadingbuildingblocks.org (www.threadingbuildingblocks.org)|54.83.27.49|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 16170189 (15M) [application/octet-stream]
Saving to: 'tbb44_20160526oss_lin.tgz'

100%[=====] 16'170'189 11.7MB/s in 1.3s

2016-06-02 15:25:33 (11.7 MB/s) - 'tbb44_20160526oss_lin.tgz' saved [16170189/16170189]

reduser@ai2pc01:~/tbb$ tar xzf tbb44_20160526oss_lin.tgz
reduser@ai2pc01:~/tbb$ ls
tbb44_20160526oss  tbb44_20160526oss_lin.tgz
reduser@ai2pc01:~/tbb$ cd tbb44_20160526oss/
reduser@ai2pc01:~/tbb/tbb44_20160526oss$ ls
bin  CHANGES  COPYING  doc  examples  include  index.html  lib  python  README
reduser@ai2pc01:~/tbb/tbb44_20160526oss$ cd bin/
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ ls
tbbvars.csh  tbbvars.sh
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ cd ..
reduser@ai2pc01:~/tbb/tbb44_20160526oss$ pwd
/home/reduser/tbb/tbb44_20160526oss
reduser@ai2pc01:~/tbb/tbb44_20160526oss$ cd bin
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ nano tbbvars.sh
Use "fg" to return to nano.

[1]+  Stopped                  nano tbbvars.sh
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ nano tbbvars.sh
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ . ./tbbvars.sh
ERROR: Unknown switch '-'. Accepted values: ia32, intel64
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ . ./tbbvars.sh intel64
reduser@ai2pc01:~/tbb/tbb44_20160526oss/bin$ cd ~/Documents/
reduser@ai2pc01:~/Documents$ mkdir test
reduser@ai2pc01:~/Documents$ cd test/
reduser@ai2pc01:~/Documents/test$ nano test.cpp
reduser@ai2pc01:~/Documents/test$ g++ test.cpp -ltbb
reduser@ai2pc01:~/Documents/test$ ./a.out

```

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

```

1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <vector>
5  #include <numeric>
6
7  #include <tbb/tbb.h>
8  #include <cstring>
9
10 class StatsCount {
11 public:
12     StatsCount(const std::string& filename, const size_t bufferSize = 512) :
13         fileName(filename), BUFFER_SIZE(bufferSize), result(256, 0) { }
14
15     void run() {
16         // Thread Local storage for characters counts
17         tbb::combinable<std::vector<size_t>> nbChars([]() { return std::vector<size_t>(256, 0); });
18
19         // thread local storage for ifstream
20         // For some reason gcc 4.8 cant compile if we use references instated
21         tbb::combinable<std::ifstream*> files([this]() {
22             return new std::ifstream(fileName, std::ios::binary | std::ios::in);
23         });
24
25         // Get the file size
26         std::ifstream file(fileName, std::ios::ate | std::ios::binary | std::ios::in);
27         if (!file.is_open()) {
28             std::cerr << "Can not open file : " << fileName << std::endl;
29             return;
30         }
31         const size_t FILE_SIZE = (size_t)file.tellg();
32         std::cout << "File size is " << FILE_SIZE << std::endl;
33
34         // Calculate the number of iteration required
35         const size_t NB_ITERATIONS = FILE_SIZE / BUFFER_SIZE +
36             ((FILE_SIZE % BUFFER_SIZE) != 0 ? 1 : 0);
37
38         // Process the counting algo in parallel
39         tbb::parallel_for(tbb::blocked_range<size_t>(0, NB_ITERATIONS),
40             [&nbChars, &files, this](const tbb::blocked_range<size_t> &r) {
41                 std::ifstream* file = files.local();
42                 if (!file || !file->is_open()) {
43                     return;
44                 }
45
46                 // Place the read cursor to the good position
47                 file->seekg(r.begin() * BUFFER_SIZE, std::ios::beg);
48
49                 // Get a ref to the localstorage for this thread
50                 std::vector<size_t>& chars = nbChars.local();
51
52                 // Count all characters
53                 uint8_t* buffer = new uint8_t[BUFFER_SIZE];
54
55                 for (size_t i = r.begin(); i != r.end(); ++i) {
56                     std::streamsize readed = file->readsome((char*)buffer, BUFFER_SIZE);
57                     while (readed-- > 0) {
58                         chars[buffer[readed]]++;
59                     }
60                 }
61
62                 delete[] buffer;
63             }
64         );
65
66         // Combine the result of all thread in the result
67         nbChars.combine_each([this](const std::vector<size_t> &v) {
68

```

```

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 librairie 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. Par exemple, si on a 16 itérations à faire mais qu'on est dans un système 4 coeurs, on aura la répartition suivantes dans chaque :

1. 0 à 3
2. 4 à 7
3. 8 à 11
4. 12 à 15

Cela abstrait la répartition des données manuel dans chaque thread, la librairie le faisant pour nous. 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) {
5 |             std::streamsize readed = file->readsome((char*)buffer, BUFFER_SIZE);
6 |             while(readed-->0) {

```

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 :

```
1 || 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 :

```
1 || file->seekg(r.begin() * BUFFER_SIZE, std::ios::beg);
```

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 :

```
1 ||     }
2 ||     );
3 ||
4 ||     // Combine the result of all thread in the result
5 ||     nbChars.combine_each([this](const std::vector<size_t> &v){
6 ||         for (size_t i = 0; i < result.size(); ++i) {
```

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

```
1 ||     });
2 ||
3 ||     // Close all opened files
4 ||     files.combine_each([this](std::ifstream* file){
```



## 3.2 Équivalent C

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <stdint.h>
4  #include <stdbool.h>
5  #include <string.h>
6  #include <pthread.h>
7  #include <assert.h>
8
9  #define NB_ASCII_CHARS (256) /* Taille de la table ASCII (facilite l'indexage)*/
10 #define NB_OCC_CHARS 62 /* Nombre de caractère dont l'occurrence doit être vérifié*/
11
12 #define MAX_BUFFER 256
13 #define NB_UPPER_CASE 26
14 #define NB_LOWER_CASE 26
15 #define NB_NUMBERS 10
16
17 #define NB_THREAD 4
18
19 struct char_stats {
20     char *path;
21     uint32_t size;
22     char tab_occ[NB_ASCII_CHARS];
23 };
24
25 struct thread_data{
26     char *path;
27     uint32_t start; /* position de départ du bloc à traiter*/
28     size_t size;
29 };
30
31 void* char_count(void* args) {
32
33     size_t nb_read;
34     uint8_t buffer[MAX_BUFFER];
35     uint8_t *occ;
36     struct thread_data *data = (struct thread_data *) args;
37
38     FILE *fp = fopen(data->path, "r");
39     if (fp == NULL) {
40         pthread_exit(NULL);
41     }
42
43     occ = calloc(NB_ASCII_CHARS, sizeof *occ);
44
45     fseek(fp, data->start, SEEK_SET);
46
47     // Trouve le minimum entre buffer et taille à lire (bits trick)
48     size_t size_to_read = data->size ^ ((MAX_BUFFER ^ data->size) & -(MAX_BUFFER < data->size));
49
50     /* Traite le bloc de mémoire assigné */
51     while((nb_read = fread(buffer, 1, size_to_read, fp)) != 0){
52         for(size_t i=0; i < nb_read; i++){
53             occ[buffer[i]]++;
54         }
55
56         // Trouve le minimum entre buffer et taille restant à lire (bits trick)
57         // Si on arrive à la fin, size_to_read vaudra 0
58         data->size -= size_to_read;
59         size_to_read = data->size ^ ((MAX_BUFFER ^ data->size) & -(MAX_BUFFER < data->size));
60     }
61
62     fclose(fp);
63
64     pthread_exit(occ);
65 }
66
67 void print_occurrences(struct char_stats *stats){
68     for(int i = 0; i < NB_UPPER_CASE; i++){
69         printf("%c : %u\n", 'a' + i, stats->tab_occ['a' + i]);
70         printf("%c : %u\n", 'A' + i, stats->tab_occ['A' + i]);
71     }
72
73     for(int i = 0; i < NB_NUMBERS; i++){
74         printf("%c : %u\n", '0' + i, stats->tab_occ['0' + i]);
75     }
76 }
77
78 struct char_stats *stats_init(const char *path)
79 {
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 allouée*/
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 size_t stats_count(struct char_stats *stats)
111 {
112     assert(stats != NULL);
113     size_t size_per_thread;
114     size_t rest;
115
116     pthread_t threads[NB_THREAD];
117     struct thread_data datas[NB_THREAD];
118     char *occ_thread;
119
120     memset(stats->tab_occ, 0, NB_ASCII_CHARS);
121
122     size_per_thread = stats->size / NB_THREAD;
123     rest = stats->size % NB_THREAD;
124
125     /* Prépare les threads avec les bloc mémoire à traiter */
126     datas[NB_THREAD-1].path = stats->path;
127     datas[NB_THREAD-1].start = (NB_THREAD-1) * size_per_thread + rest;
128     datas[NB_THREAD-1].size = size_per_thread;
129     pthread_create(&threads[NB_THREAD-1], NULL, char_count, &datas[NB_THREAD-1]);
130
131     for(int i = NB_THREAD-2; i >= 0; i--){
132         datas[i].path = stats->path;
133         datas[i].start = i * size_per_thread;
134         datas[i].size = size_per_thread;
135         pthread_create(&threads[i], NULL, char_count, &datas[i]);
136     }
137
138     for(int i = 0; i < NB_THREAD; i++){
139         pthread_join(threads[i], (void**)&occ_thread);
140
141         // Vérifie que le pointeur retourné est non null
142         if(occ_thread == NULL) continue;
143
144         // Met à jour le tableau des occurrences
145         for(unsigned char j = 'a'; j <= 'z'; j++){
146             stats->tab_occ[j] += occ_thread[j];
147         }
148
149         for(unsigned char j = 'A'; j <= 'Z'; j++){
150             stats->tab_occ[j] += occ_thread[j];
151         }
152
153         for(unsigned char j = '0'; j <= '9'; j++){
154             stats->tab_occ[j] += occ_thread[j];
155         }
156         free(occ_thread);
157     }
158     return NB_OCC_CHARS;
159 }
160
161 void stats_clear(struct char_stats *stats)
162 {
163     free(stats->path);
164     free(stats);
165 }
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 effectuée 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 utilisés dans les deux cas. On a plus que 3 CPU utilisés durant l'exécution du programme. On peut donc dire que TBB fait bien son travail et parallélise notre implémentation.

## 5 Conclusion

Au final, on voit que la librairie TBB fait bien son travail. Elle répartit la tâche dans plusieurs threads et dans tous les cœurs sans même avoir à demander combien de thread on veut lancer. Il exploite ainsi tout le potentiel du processeur sans que le développeur ait à se soucier combien de thread il doit lancer. On a donc une bonne abstraction de la part de la librairie avec des performances satisfaisantes qui permettent au développeur de faire du code simple, compréhensible et générique, sachant que la librairie s'adapte tout seule à l'implémentation des threads selon l'OS ou l'architecture utilisée.

## 6 Bibliographie

- Tutoriel d'Intel : <https://www.threadingbuildingblocks.org/intel-tbb-tutorial>
- Documentation d'Intel : <https://software.intel.com/en-us/tbb-documentation>