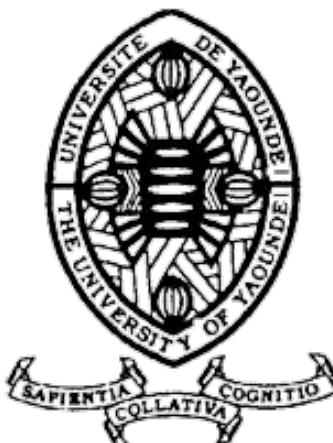


# **Etude de code de parallélisation en langages de programmation c et c++**

**Fiche d'expérimentation et d'analyse**

Supervisé par :  
**Dr. MESSI**



**Faculté des Sciences**

Nom : **DJIEMBOU TIENCHEU VICTOR NICO**  
Devoir : 2  
Lien : <drive.google.com/drive/folders/1YMUEPascSgc4nJK2Cw4BFDG6IaDJ0Eap>  
Reçu le : 2023.11.29  
Auteur : Perçu de Dr. Messi  
Mots clés : Distributed and parallel ML, c, c++, threads  
UE : INF5099  
Matricule : 17T2051  
Département : Informatique  
Niveau - Option : M2 - Sciences des Données

## Table des matières

1	Contexte	3
2	Problème	3
3	motivations	3
4	Parallélisation dans le langage c++	3
5	Parallélisation dans le langage c	3
6	Conclusion	4
7	Annexe	6

# 1 Contexte

À l'université de Yaoundé 1, en classe de Master 2 du cycle de Master les étudiants sont appelés à s'initier à la recherche et à monter en compétence dans des domaines fare des sciences informatiques.

En option sciences des données, nous avons parmi autre pour objectif de pouvoir améliorer les performance de nos processus d'apprentissage. C'est dans cette optique que nous nous focalisation dans la distribution de nos apprentissage en particulier dans des langages de développement c et c++.

L'enseignant pour nous plonger dans ce decor a donc jugé nécessaire de nous faire visiter quelques cas de base de code de parallélisation.

# 2 Problème

Etant appeler à proposer des solution d'apprentissage distribués en langages c et c++, le problème est de savoir comment implementer un code parallélisé ? Quels sont les outils qui dans ses langages serve la cause de la parallélisation de processus ?

# 3 motivations

Il serait intéressant pour nous étudiants de pouvoir à partir de cours de pouvoir expertiser sur

- comment penser et mettre sur pieds un projet avec logique de parallélisation dans les langages c et c++ ?
- quels sont les librairies utilisées dans ses distincts langages pour paralléliser le code ?
- quels sont les contraintes d'implementation dans chaque langage ?

# 4 Parallélisation dans le langage c++

Sur la base des codes **code\_parallel.cpp** et **produit\_scalaire.cpp**, nous pouvons avoir un début de compréhension sur comment se passe la parallélisation sous le langage c++.

Déjà, à la lumière de mon analyse, j'ai décélé 2 grands niveau de fourniture de la parallélisation. Déjà nous devons pouvoir mettre sur pieds un code fonctionnel à logique parallèle puis nous devons pouvoir fournir les ressources nécessaire lors de la compilation du fichier binaire d'exécution. À la Figure 1, nous présentons avec commentaire à l'appui comment est-ce que une code c++ peut être parallélisé moyennant la bibliothèque **thread** qui permet de programmer des solutions qui utilisent plusieurs threads. Ensuite, le code étant conçu de tel sorte à autoriser une exécution parallèle, il faudrait bien encore que lors de la compilation, nous explicitions à compilateur GCC de c++ (g++) qu'il s'agit d'un code qui utilisera une logique multithread (voir Figure 2).

# 5 Parallélisation dans le langage c

L'ajout d'une logique de manipulation multithreading en c nécessite impérativement l'inclusion dans le projet de la bibliothèque **pthread.h**. De plus contrairement à c++, c est plus bas niveau, donc il faut tenir compte de bien plus d'aspect durant la parallélisation tels que le **verrouillage**, **déverrouillage**, **lancement** et **arrêt d'un thread** des ressources avec un service de **mutex** qu'offre la bibliothèque tel que **pthread\_mutex\_init**, **pthread\_mutex\_lock**, **pthread\_mutex\_unlock**, **pthread\_attr\_setdetachstate**, **pthread\_exit** (voir plus de détails dans les Figures 3, 4, 5). En ce qui concerne le Makefile de compilation et d'exécution, il demeure identique à celui de c++ à la seul différence du compilateur GCC de c qui est gcc au lieu de g++.

## 6 Conclusion

La programmation parallèle est un concept très répandu dans le domaine d'apprentissage automatique et est accessible dans deux nombreux langages tels que c et c++. Dans ses derniers, il est toujours important de codé avec les outils mise en oeuvre pour la programmation multithread tels **thread** et **pthread.h** respectivement en c++ et c, et d'associer lors de la compilation de la version binaire, les ressources permettant cette exécution parallèle grâce à des flag qui on été préalablement mise en place dans le compilateur gcc qui sont entre autres **-lpthread -fopenmp**.

## Références

- [Bre09] Clay Breshears, *The art of concurrency : A thread monkey's guide to writing parallel applications*, O'Reilly Media, 2009.
- [KKI03] George Em Karniadakis and Robert M. Kirby II, *Parallel scientific computing in c++ and mpi : A seamless approach to parallel algorithms and their implementation*, Cambridge University Press, 2003.
- [Pac11] Peter Pacheco, *Parallel programming in c++ and mpi*, Morgan Kaufmann, 2011.
- [Qui03] Michael J. Quinn, *Parallel programming in c with mpi and openmp*, McGraw-Hill Education, 2003.
- [WA04] Barry Wilkinson and Michael Allen, *Parallel programming : Techniques and applications using networked workstations and parallel computers*, 2nd ed., Prentice Hall, 2004.
- [Wil12] Anthony Williams, *C++ concurrency in action : Practical multithreading*, Manning Publications, 2012.
- [Wil19] \_\_\_\_\_, *C++ concurrency in action : Practical multithreading*, Manning Publications, 2019.

## 7 Annexe

```
1 // C++ program to demonstrate
2 // multithreading using three
3 // different callables.
4 #include <iostream>
5 #include <thread>           module de programmation multithread
6 #include <vector>
7
8 using namespace std;
9
10 // A dummy function
11 void foo(int z)
12 {
13     cout << "I am thread " << z << endl;
14 }
15
16 // A callable object
17 class thread_obj {
18 public:
19     void operator()(int x)
20     {
21         for (int i = 0; i < x; i++)
22             cout << "Thread using function object as callable\n";
23     }
24 };
25
26 // Driver code
27 int main()
28 {
29     int n = 5;
30     std::vector<std::thread> threads(n);      création du vecteur de thread threads
31     // spawn n threads:
32
33     for(int i = 0; i < n; i++) {               création de n threads pour lancer la méthode foo sus-définie avec l'argument i
34         threads[i] = thread(foo, i);          patienter la fin d'exécution d'un thread avant de passer au suivant
35     }
36
37     // Wait for thread t1 to finish
38     for (auto& th: threads) {
39         th.join();
40     }
41
42     return 0;
43 }
44
45
```

FIGURE 1: Explication des parties du code parallèle

```

1  exec produit_scalaire:produit_scalaire.cpp ← a:b, service à dépendant de l'existant de la ressource b
2  g++ --std=c++11 -o produit_scalaire produit_scalaire.cpp -g -O0 -lm -lpthread
3  #           ↗ g++ --std=c++11 -o produit_scalaire produit_scalaire.cpp -g -O0 -lm -lpthread -fopenmp
4  gcc          ↗
5  posr run_produit_scalaire:produit_scalaire génération du binaire
6  c++ ./produit_scalaire 10000 2 autoriser le débogage
7                                     ↗ désactiver l'optimisation de code
8  clean: rm produit_scalaire fourniture de la bibliothèque mathématique
9                                     ↗ fourniture des fonctionnalités pour la
10                                     ↗ programmation multithread sur c++
11                                     ↗ liaison du support OpenMP
12                                     ↗
13                                     ↗ spécification de la version de c++    supprimer le binaire    exécution avec 2 paramètres
14                                     ↗

```

FIGURE 2: Explication des ressources nécessaires pour la compilation d'un code parallèle

```

1  #include <pthread.h>                    charger les fonctionnalités
2  #include <stdlib.h>                     de programmation
3  #include <stdio.h>                      multithread avec c
4  #include <math.h>
5  #include <time.h>
6  #include <sys/time.h>
7  #include <limits.h>
8
9  typedef struct timezone timezone_t;
10  typedef struct timeval timeval_t;
11
12  timeval_t t1, t2;
13  timezone_t tz;
14
15
16  static struct timeval _t1, _t2;
17  static struct timezone _tz;
18  timeval_t t1, t2;
19  timezone_t tz;
20
21  static unsigned long _temps_residuel = 0;
22  #define top1() gettimeofday(&_t1, &_tz)
23  #define top2() gettimeofday(&_t2, &_tz)
24
25  void init_cpu_time(void)
26  {
27      top1(); top2();
28      _temps_residuel = 1000000L * _t2.tv_sec + _t2.tv_usec -
29      (1000000L * _t1.tv_sec + _t1.tv_usec );
30  }
31
32  unsigned long cpu_time(void) /* retourne des microsecondes */
33  {
34      return 1000000L * _t2.tv_sec + _t2.tv_usec -
35      (1000000L * _t1.tv_sec + _t1.tv_usec ) - _temps_residuel;
36  }
37
38
39
40
41  // #define NUM_THREADS 1
42  int NUM_THREADS = 1;

```

FIGURE 3: Importation de la bibliothèque pour la programmation parallèle en c

```

140
141     int main(int argc, char *argv[])
142     {
143         pthread_t*thread;
144         pthread_attr_t attr; ← déclarer de variables devant accueillir
145         //les threads et leur attribut à chaque
146         //appel
147         int rc, n;
148         float somme, somme_parallel;
149         long t;
150         void *status;
151
152         srand(time(NULL));
153
154         float* x, *y;
155
156         //printf("Entrez la taille des vecteurs\n");
157         //scanf("%d", &n);
158
159         if(argc < 3)
160         {
161             printf("Nombre d'argument insuffisant");
162             exit(-1);
163         }
164         else
165         {
166             n = atoi(argv[1]);
167             NUM_THREADS = atoi(argv[2]);
168         }
169         thread = (pthread_t*)malloc(sizeof(pthread_t)*NUM_THREADS); ← redéfinir la dimension du vecteur de
170
171         x= creer_vecteur(n);
172         y= creer_vecteur(n);
173
174         printf("\nnb_thread = %d\n", NUM_THREADS);
175         data.x = x;
176         data.y = y;
177         data.sum = 0.0;
178         data.length= n;
179
180         pthread_mutex_init(&mutex_sum, NULL); ← initialiser une ressource mutable par
181
182         somme = produit_scalaire(x,y,n); ← initialiser l'attribut de thread et l'associé
183         top2(); ← au thread via à la ressource
184
185         unsigned long temps = cpu_time();
186         printf("\nTime seq = %ld.%03ldms\n", temps/1000, temps%1000);
187
188         /* Initialize and set thread detached attribute */
189         pthread_attr_init(&attr);
190         pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE); ← création d'un thread
191
192         top1();
193         for(t=0; t<NUM_THREADS; ++t) {
194             //printf("Main: creating thread %d\n", t);
195             rc = pthread_create(&thread[t], &attr, produit_scalaire_parallel_1, (void *)t); ←
196             if (rc) {
197                 printf("ERROR: return code from pthread_create() is %d\n", rc);
198                 exit(-1);
199             }
200         }
201

```

FIGURE 4: Mise en oeuvre de la programmation parallèle en c

```

52
53     data_t data;
54     pthread_mutex_t mutex_sum; ← déclaration du label de variable du
55
56     void *produit_scalaire_parallel_1(void *arg) ← mutex
57     {
58         /* Define and use local variables for convenience */
59         int i, taille;
60         long debut, fin;
61         long indice;
62         float somLocale, *x, *y;
63         indice = (long)arg;
64
65         taille = data.length;
66
67         debut = indice*(taille/NUM_THREADS);
68
69         if(indice == (NUM_THREADS - 1))
70             fin = data.length;
71         else
72             fin = debut + (taille/NUM_THREADS);
73
74         x = data.x;
75         y = data.y;
76
77         /* Perform the dot product and assign result to the appropriate variable in the structure. */
78         somLocale = 0;
79         for(i=debut; i<fin ; i++)
80         {
81             somLocale += (x[i] * y[i]);
82         }
83         //printf("Thread %d debut = %ld fin = %ld\n", indice, debut, fin);
84
85         /* Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating. */
86         pthread_mutex_lock (&mutex_sum); ← verrouiller un mutex pour édition,
87         data.sum += somLocale; ← puis éditer et enfin déverrouiller
88         pthread_mutex_unlock (&mutex_sum);
89
90
91         pthread_exit((void*) 0); ← libérer le thread
92     }
93

```

FIGURE 5: Verrou transactionnel entre threads