

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 TIENTCHEU 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 faire 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 décelé 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 mutlithread 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>
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<<" \n";
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);
31     // spawn n threads:
32     for(int i = 0; i < n; i++) {
33         threads[i] = thread(foo, i);
34     }
35     // Wait for thread t1 to finish
36     for (auto& th: threads) {
37         th.join();
38     }
39     return 0;
40 }
```

module de programmation multithread

classe dont les instances sont peuvent être exécuté comme des fonctions

création du vecteur de thread threads

création de n threads pour lancer la méthode foo sus-définie avec l'argument i

patienter la fin d'exécution d'un thread avant de passer au suivant

FIGURE 1: Explication des parties du code parallèle

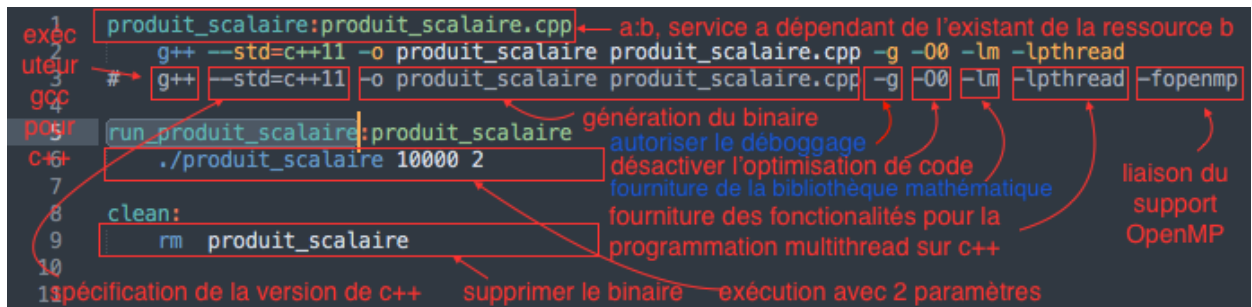


FIGURE 2: Explication des ressources nécessaire pour la compilation d'un code parallèle

```

1 #include <pthread.h>
2 #include <stdlib.h>
3 #include <stdio.h>
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;

```

Annotation: **charger les fonctionnalités de programmation multithread avec c** (pointing to `#include <pthread.h>`)

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;
145     int rc, n;
146     float somme, somme_parallele;
147     long t;
148     void *status;
149
150     srand(time(NULL));
151
152     float *x, *y;
153
154     //printf("Entrez la taille des vecteurs\n");
155     //scanf("%d",&n);
156
157     if(argc < 3)
158     {
159         printf("Nombre d'argument insuffisant");
160         exit(-1);
161     }
162     else
163     {
164         n = atoi(argv[1]);
165         NUM_THREADS = atoi(argv[2]);
166
167         thread = (pthread_t *) malloc(sizeof(pthread_t) * NUM_THREADS);
168
169         x = creer_vecteur(n);
170         y = creer_vecteur(n);
171
172         printf("\nnb_thread = %d\n", NUM_THREADS);
173         data.x = x;
174         data.y = y;
175         data.sum = 0.0;
176         data.length = n;
177
178         pthread_mutex_init(&mutex_sum, NULL);
179
180     }
181
182     top1();
183     somme = produit_scalaire(x,y,n);
184     top2();
185
186     unsigned long temps = cpu_time();
187     printf("\ntime seq = %ld.%03lds\n", temps/1000, temps%1000);
188
189     /* Initialize and set thread detached attribute */
190     pthread_attr_t attr;
191     pthread_attr_t attr;
192     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
193
194     top1();
195     for(t=0; t<NUM_THREADS; t++) {
196         //printf("Main: creating thread %ld\n", t);
197         rc = pthread_create(&thread[t], &attr, produit_scalaire_parallele_1, (void *)t);
198         if (rc) {
199             printf("ERROR: return code from pthread_create() is %d\n", rc);
200             exit(-1);
201         }
202     }
203 }

```

déclarer de variables devant accueillir les threads et leur attribut à chaque appel

redéfinir la dimension du vecteur de threads à la taille dde la valeur du deuxième paramètre d'exécution

initialiser une ressource mutable par les différents threads

initialiser l'attribut de thread et l'associer au thread via à la ressource PTHREAD_CREATE_JOINABLE

création d'un thread

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

```

52
53 data_t data;
54 pthread_mutex_t mutex_sum;
55
56 void *produit_scalaire_parallele_1(void *arg)
57 {
58     /* Define and use local variables for convenience */
59     int i, taille;
60     long debut, fin;
61     long indice;
62     float som_locale, *x, *y;
63     indice = (long) arg;
64
65     taille = data.length;
66     debut = indice * (taille / NUM_THREADS);
67
68     if(indice == (NUM_THREADS - 1))
69         fin = data.length;
70     else
71         fin = debut + (taille / NUM_THREADS);
72
73     x = data.x;
74     y = data.y;
75
76     /* Perform the dot product and assign result to the appropriate variable in the structure. */
77     som_locale = 0;
78     for (i = debut; i < fin; i++)
79     {
80         som_locale += (x[i] * y[i]);
81     }
82     //printf("Thread %ld debut = %ld fin = %ld\n", indice, debut, fin);
83
84     /* Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating. */
85     pthread_mutex_lock(&mutex_sum);
86     data.sum += som_locale;
87     pthread_mutex_unlock(&mutex_sum);
88
89     pthread_exit((void *) 0);
90 }
91
92
93

```

déclaration du label de variable du mutex

verrouiller un mutex pour édition, puis éditer et enfin déverrouiller

libérer le thread

FIGURE 5: Verrou transactionnel entre threads