UNIVERSITÉ DE SHERBROOKE

Application d'une méthode pour montrer qu'une parallélisation est correcte

PRÉSENTÉ À :

Gabriel Girard, professeur

PAR

Gabriel Blais Bourget
10 132 000

27 avril 2012

Contenu

Introduction

Au cours des dernières années, j'ai eu la chance de travailler sur plusieurs projets d'envergures. Bien que différends à de nombreux niveaux, ceux-ci consistaient, généralement, à faire la refonte d'un système déjà en place. Il semblerait que ce genre de projets est de plus en plus commun. En effet, il y a de cela pas très longtemps, notre monde a connu une effervescence de l'informatique incroyable. Les entreprises, pour suivre la vague, en ont fait une course à l'armement. De ce fait, une multitude de systèmes ont vu le jour. Cela fait en sorte qu'aujourd'hui, beaucoup d'entreprises trainent de vieux programmes et ceux-ci ont besoin d'être mis à jour pour pouvoir continuer à être compétitifs et fonctionnels. Il est donc courant, de nos jours, de devoir réécrire des programmes au complet. J'ai été surpris d'apprendre que le défi pour ce genre de projet n'est pas la réécriture elle-même, mais plutôt de m'assurer que les comportements soient identiques.

La méthode la plus utilisée est de simplement tester les programmes. Par contre, comme l'a dit Dijkstra : « Tester un programme peut démontrer la présence de bugs, jamais leur absence. ». Cela fait en sorte qu'il est presque impossible de montrer une équivalence à l'aide de tests. Sur ce, pendant un cours de processus concurrents et parallélisme, je me suis questionné sur la difficulté de démontrer qu'un programme séquentiel est pareil à sa parallélisation. J'ai découvert qu'il est possible de prouver avec exactitude cette équivalence de manière non exhaustive. C'est ainsi qu'en tant que néophyte en matière de système de preuve appliqué à l'informatique, j'ai exploré une méthode qui me semblait applicable dans un laps de temps raisonnable.

Cet ouvrage s'attarde principalement à une méthode qui a été développée dans la thèse de Raphaël Couturier, à l'époque étudiant au doctorat en informatique. Dans un premier temps, je ferai une description de cette méthode. Ensuite, je décrirai un programme séquentiel, servant à faire une sommation, que j'ai écrite dans le but de servir d'exemple d'application. Je décrirai aussi comment je me suis pris pour faire sa

parallélisation. Un second programme pour briser des mots de passe sera aussi décrit de la même façon, mais je n'ai pas écrit le code étant donné sa trivialité. Après, je commencerai les exemples d'applications en élaborant les post-conditions nécessaires. Pour terminer, j'appliquerai la preuve au programme servant à briser les mots de passe.

Méthode pour montrer qu'une parallélisation est correcte

La méthode proposée par M. Couturier s'adresse aux « parallélisations utilisant l'archétype de décomposition de domaine ». Cela signifie que le programme ou l'algorithme utilise pratiquement les mêmes calculs que leur version séquentielle, mais appliquée à des sous-ensembles. De cette manière, nous devons démontrer que les calculs sont les mêmes et donnent les mêmes résultats. Par contre, il est aisé de s'imaginer des cas où cela peut être très difficile. Par conséquent, il nous est plutôt proposé d'établir des post-conditions pour chacune des versions. Cela consiste en fait à établir, par abstraction, une condition qui devra être vérifiée après l'exécution de l'algorithme. Cette étape devrait être la plus complexe à réaliser puisque cela oblige à posséder une très bonne compréhension de ce que fait le code. En établissant une post-condition trop abstraite, on ne prouvera rien et si elle ne l'est pas suffisamment, on peut avoir à prouver le code en entier. Il faut donc, premièrement, choisir une post-condition pour la version séquentielle. Deuxièmement, il faut choisir une post-condition pour chaque partie parallèle. Finalement, choisir une post-condition que devra respecter la « colle » entre nos parties parallèles. Celle-ci étant la partie séquentielle de notre algorithme, ou programme, parallèle qui s'occupe de distribuer le domaine et de compiler les résultats.

Comme il a été dit plus tôt, le choix des post-conditions peut-être difficile. Sur ce, voici ce que l'auteur de la méthode propose comme critères important à respecter :

« - L'abstraction du programme séquentiel, de la colle et des parties parallèles doit obligatoirement faire apparaitre les données qui contiennent le résultat du programme

et les données qui sont distribuées. » … « vérifier que les calculs sont distribués correctement et les données contenant un résultat vont nous permettre de vérifier que le résultat du programme est correct. »

« - L'abstraction du programme séquentiel doit modéliser une partie du calcul entre toutes les données. » … « on abstrait ou non une bonne partie des calculs afin de garder l'essentiel uniquement. »

« - L'abstraction des parties parallèles doit si possible être similaire à l'abstraction du programme séquentiel. » … « l'abstraction des parties parallèles doit faire apparaitre que certains calculs sont effectués localement sur un processeur » … « La différentiation entre les calculs utilisant des données locales à un processeur et les calculs utilisant des données distantes à un processeur est primordiale. »

« L'abstraction de la colle doit modéliser le fait que les calculs faisant intervenir des données disposées sur plusieurs processeurs doivent être effectués. »

Une fois les post-conditions choisis, nous devons prouver que si l'ensemble de nos parties parallèle est vérifié et que notre « colle » l'est aussi, cela implique la post-condition de la partie séquentielle. La méthode veut que pour ceci nous utilisions un prouveur de théorèmes.

- Soit n parties parallèles, PP_i une partie parallèle tel que, $i \in [1 \dots n]$. PS est le programme séquentiel, C est la colle et $post(P)$la post-condition du code P. Il faut prouver que :

$$\bigwedge_{i=1}^{n} post(PP_i) \wedge post(C) \Rightarrow post(PS)$$

Description des programmes choisis

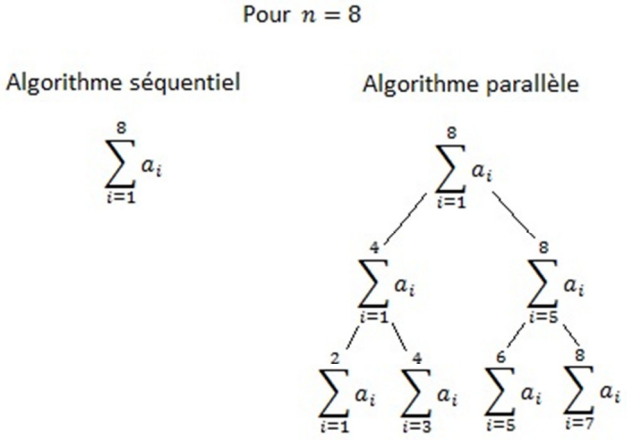
Programme de sommation (code Annexe A)

Le programme a été écrit dans le langage C et utilise les librairies MPI pour les parties parallèles. En fait, il simule deux comportements distincts qu'on pourrait voir comme 2 programmes. Si on exécute le programme avec un processus, seule la partie séquentielle sera exécutée. Si on a plus d'un processus, la partie parallèle sera exécutée.

L'algorithme séquentiel de sommation ne pourrait être plus simple. On boucle sur les valeurs du vecteur et on les additionne pour arriver à un total. Il a une complexité algorithmique de O(n).

L'algorithme parallèle se déroule en $\log_2 n$ étapes. On isole chacune des additions pour en fabriquer un vecteur de résultats partiels. On recommence à partir du nouveau vecteur jusqu'à ce qu'on obtient qu'une seule valeur. La première étape fera alors $n/2^1$ sommations, la deuxième $n/2^2$ et ainsi de suite jusqu'à ce qu'on est qu'une seule sommation. La complexité algorithmique est O($n \log_2 n$), mais sera en réalité en O($\log_2 n$) étant donné la parallélisation. L'illustration suivante montre le fonctionnement des deux algorithmes :

Fig. 1.1 – *Description des algorithmes séquentielle et parallèle*



Programme d'attaque force brutes

Prenons en exemple un programme dont le but est de briser un mot de passe de 5 chiffres en tentant toutes les possibilités. L'algorithme séquentiel est, encore une fois, fort simple. On commence à partir de la première possibilité, on compare celle-ci avec le mot de passe et s'ils ne sont pas égaux, alors on essaie la deuxième possibilité. On continue de la sorte jusqu'à ce que toutes les possibilités aient été explorées ou jusqu'à ce que le mot de passe ait été trouvé. Il a une complexité algorithmique de O(n^m), m étant le nombre de chiffres du mot de passe.

L'algorithme parallèle fonctionne presque de la même manière. Par contre, on divise le nombre total de possibilités par le nombre de processus pour créer des sous-ensembles. Puis, on partage ceux-ci entre les processus. Chaque partie parallèle travaillera exactement comme le fait l'algorithme séquentiel. Quand le mot de passe aura été trouvé par l'un d'entre eux, celui-ci retournera la réponse au processus maitre. La complexité algorithmique sera alors de O(n^m/p), p étant le nombre de processus.

Choix des post-conditions

Programme de sommation

Étant donné la simplicité de l'algorithme séquentiel, il est difficile d'appliquer un niveau d'abstraction pour le choix de la post-condition. Cependant, cela dépend de la spécification donnée par le client. Il est possible que celui-ci ne désire que faire la sommation de vecteur de taille 2^n. Prouver que la sommation fonctionne pour n'importe quelle taille de vecteur serait alors inutile. Notre post-condition est que le résultat de la sommation soit la sommation des nombres que contient un vecteur quelconque de taille 2^n.

Comme il l'a été décrit plus tôt, le but d'une partie parallèle est de faire la sommation d'un vecteur de deux nombres. Notre post-condition sera alors que le résultat de la sommation est égal à la somme des deux nombres en entrée.

En ce qui concerne la « colle », la post-condition consisterait à ce que le programme envoie les $n \log_2 n$ calculs à faire aux processus et que celui-ci cumule les résultats partiels. En débutant avec $n/2$ sommations, puis $n/4$ et ainsi de suite jusqu'à avoir qu'une seule sommation qui donnera le résultat final.

Finalement pour faire la preuve que le programme est équivalent à sa version séquentielle : Si tous les résultats des sommations de deux nombres sont vrais et que la « colle » à bien partagé et compilé les résultats pour obtenir un vecteur d'un résultat, cela implique que le résultat de la sommation soit la sommation des nombres d'un vecteur de taille 2^n.

On remarque que si la post-condition de la partie séquentielle avait inclus des vecteurs de taille impaire, la colle, telle qu'on l'a défini, n'aurait pas suffi.

Programme d'attaque force brutes

Il a été dit précédemment que le but du code séquentiel fût de trouver le mot de passe ou d'avoir tenté toutes les possibilités, mais est-il possible d'avoir tout tenté et de n'avoir rien trouvé? Notre post-condition ne s'intéressera alors qu'au cas où l'algorithme trouve le mot de passe.

Dans le cas des parties parallèles, il est normal qu'elles ne trouvent pas tout le mot de passe. Notre post-condition sera alors que le mot de passe a été trouvé ou que toutes les possibilités ont été tentées pour un ensemble donné.

La post-condition de la « colle » sera que la somme des possibilités des sous-ensembles est égale au nombre total de possibilités.

Finalement, pour prouver que le programme parallèle produit les mêmes résultats que sa version séquentielle on a : Si chaque partie parallèle tente de trouver le mot de passe et que la somme des sous-ensembles de possibilités est égale au nombre total de possibilités, cela implique que le mot de passe a été trouvé.

Preuve à l'aide d'un prouveur de théorème

Programme d'attaque force brutes

Le prouveur de théorème, nommé Z3, qui a été utilisé a été développé par Microsoft Research. Pour faire la preuve, je me suis servi de l'interface Python disponible sur le Web. Le code est disponible à l'annexe B. J'y ai mis un nombre fixe de processeurs, car le prouveur ne me permettait pas de cas général. Il a été écrit pour prouver que la parallélisation du programme séquentiel est correcte. Voici les résultats obtenus à l'exécution :

y = Mot de passe a trouver
x = Mot de passe trouve
s = Nombre total de possibilités
d = Debut du sous-ensemble
f = Fin du sous-ensemble
p = Nombre de processeurs

Post-condition partie sequentiel
$y \geq 0 \wedge y < s \Rightarrow x = y$

Post-condition d'une partie parallèle
$d \leq y \wedge y < f \Rightarrow x = y \vee x = f$

Post-condition de la colle
$(s/6)\cdot 1 - (s/6)\cdot 0 + (s/6)\cdot 2 - (s/6)\cdot 1 + (s/6)\cdot 3 - (s/6)\cdot 2 + (s/6)\cdot 4 - (s/6)\cdot 3 + (s/6)\cdot 5 - (s/6)\cdot 4 + s/6)\cdot 6 - (s/6)\cdot 5 = s$

Preuve finale
$((y \geq 0 \wedge y < s/6 \Rightarrow x = y \vee x = s/6) \wedge (s/6 \leq y \wedge y < (s/6)\cdot 2 \Rightarrow x = y \vee x = (s/6)\cdot 2) \wedge ((s/6)\cdot 2 \leq y \wedge y < (s/6)\cdot 3 \Rightarrow x = y \vee x = (s/6)\cdot 3) \wedge ((s/6)\cdot 3 \leq y \wedge y < (s/6)\cdot 4 \Rightarrow x = y \vee x = (s/6)\cdot 4) \wedge ((s/6)\cdot 4 \leq y \wedge y < (s/6)\cdot 5 \Rightarrow x = y \vee x = (s/6)\cdot 5) \wedge ((s/6)\cdot 5 \leq y \wedge y < (s/6)\cdot 6 \Rightarrow x = y \vee x = (s/6)\cdot 6) \Rightarrow x = y) \wedge (s/6)\cdot 1 - (s/6)\cdot 0 + (s/6)\cdot 2 - (s/6)\cdot 1 + (s/6)\cdot 3 - (s/6)\cdot 2 + (s/6)\cdot 4 - (s/6)\cdot 3 + (s/6)\cdot 5 - (s/6)\cdot 4 + (s/6)\cdot 6 - (s/6)\cdot 5 = s \Rightarrow y \geq 0 \wedge y < s \Rightarrow x = y$

Resolution...
Prouve

Conclusion

Cette méthode présente plusieurs aspects très intéressants. Premièrement, elle permet de prouver qu'une parallélisation est correcte sans avoir à prouver le code en entier. Deuxièmement, le fait qu'elle utilise un prouveur de théorème permet de développer des preuves qui pourraient être très complexes. Par contre, l'utilisation de tels logiciels n'est pas simple. Cela peut demander beaucoup de temps à maitriser. La méthode que propose M. Couturier présente aussi quelques lacunes. Il y a une étape, que j'ai omise, qui consiste à réécrire le code qui affecte les post-conditions et puis tester celui-ci à plusieurs reprises afin de comparer les résultats. Cela me semble une tâche très longue et qui semble enlever tous les bienfaits de la méthode. De plus, il y a certains algorithmes qui sont très longs à exécuter, comme le force brute. Les tester de nombreuses fois semble absurde. En résumé, je crois que cette façon de prouver une parallélisation ne s'applique qu'à un nombre très restreint d'algorithmes ou de programmes, mais elle apporte aussi certaines notions qui méritent d'être étudiées davantage.

Annexe A – Programme de sommation

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0              // Identifiant du processus maître
#define TABLE_LENGTH 8        // La taille doit être de 2^n
#define DOWORK 1              // Indique à l'esclave qu'il doit travailler
#define DONOTWORK 0           // Indique à l'esclave que son travail est terminer
#define NB_TO_ADD 3           // Taille du tampon de travail à envoyer
```

```
/**
	@desc  S'il n'a aucun esclave à sa charge, le maître fait la sommation
			lui-même. Sinon, il distribue les sommes à faire à ses esclaves
	@param	A : Vecteur de nombre à sommer
	@return	Résultat de la sommation
 */
int master(int[]);

/**
	@desc  Calcul la somme d'un vecteur de deux nombres
 */
void slave();

/**
	@desc  Initialise un vecteur avec des nombres de 1 à TABLE_LENGTH
	@param	A : Vecteur à initialiser
 */
void initTable(int[]);

/**
	@desc  Algorithme de sommation séquentiel
	@param	A : Vecteur de nombre à sommer
	@return	Résultat de la sommation
 */
int addSequential(int[]);

int main(int argc, char **argv)
{
	int A[TABLE_LENGTH];		// Vecteur contenant les nombres à sommer
	int total;			// Somme des nombres du vecteur
	int rank;			// Identifiant du processus

	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);

	initTable(A);

	// Division du travail entre maître et esclaves
	if(rank == MASTER)
	{
		total = master(A);
		printf("Le total est de %d\n", total);
	}
	else
		slave();

	MPI_Finalize();
```

```
	return 0;
}

int master(int A[TABLE_LENGTH])
{
	int* total;			// Tableau de sommes
	int size;			// Nombre d'esclaves
	int work[3];			// Nombres à calculer par l'esclave
	int result[1];			// Résultat obtenue par l'esclave
	int esclave;			// Numéro d'esclave
	int i;				// Compteur
	int h;				// Taille du tableau de résultats

	MPI_Status status;
	MPI_Comm_size(MPI_COMM_WORLD, &size);

	// S'il n'y a qu'un seul processus,
	// alors le maitre fait le calcul de manière séquenciel.
	if(size == 1)
		return addSequential(A);

	// Sinon, il le fait faire par ses esclaves en parallèle
	h = TABLE_LENGTH;
	total = malloc(h * sizeof(int));
	for(i = 0;i < h;++i)
		total[i] = A[i];

	while(h != 1)
	{
		esclave = 1;
		// Envoie aux esclaves les nombres à additioner
		for(i = 0;i < h - 1;++i)
		{
			work[0] = DOWORK;
			work[1] = total[i];
			work[2] = total[++i];

			MPI_Send(work, NB_TO_ADD, MPI_INT, esclave, 0,
				MPI_COMM_WORLD);
			if(++esclave == size) esclave = 1;
		}

		// Réception des totaux
		esclave = 1;

		// Création d'un nouveau vecteur de nombres avec les résultats partiels
		free(total);
```

```
                h = (h / 2);
                total = malloc(h * sizeof(int));

                // Place les résultats dans le nouveau vecteur
                for(i = 0;i < h;++i)
                {
                        MPI_Recv(result, 1, MPI_INT, esclave, 0, MPI_COMM_WORLD,
                                &status);

                        total[i] = result[0];
                        if(++esclave == size) esclave = 1;
                }
        }

        // Arrêt du travail des esclaves
        for(i = 1;i < size;++i)
        {
                work[0] = DONOTWORK;
                MPI_Send(work, NB_TO_ADD, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        return total[0];
}

void slave()
{
        int work[NB_TO_ADD];            // Travail à faire
        int total[1];                   // Résultat du travail

        MPI_Status status;

        while(1)
        {
                MPI_Recv(work, NB_TO_ADD, MPI_INT, MASTER, 0,
                        MPI_COMM_WORLD, &status);

                // Arrête de travailler s'il en reçoit l'ordre
                if(work[0] == 0)
                        break;

                total[0] = work[1] + work[2];
                MPI_Send(total, 1, MPI_INT, MASTER, 0, MPI_COMM_WORLD);
        }
}

void initTable(int A[TABLE_LENGTH])
{
        int i;
```

```
	for(i = 1;i <= TABLE_LENGTH;++i)
		A[i - 1] = i;
}

int addSequential(int A[TABLE_LENGTH])
{
	int total = 0;
	int i;

	for(i = 0;i < TABLE_LENGTH;++i)
		total += A[i];

	return total;
}
```

Annexe B – Preuve à l'aide d'une prouveur

```
y = Int('y') #Mot de passe à trouver
x = Int('x') #Mot de passe trouvé
s = Int('s') #Nombre total de possibilités
d = Int('d') #Début du sous-ensemble
f = Int('f') #Fin du sous-ensemble
p = 6	#Nombre de processeurs

#Algorithme séquentiel
def PS(z):
```

```
   return Implies(And(z >= 0,z < s), x == z)

#Algorithme parallèle
def PP(a, b, z):
   return Implies(And(a <= y,y < b), Or(x == z, x == b))

#Colle
def C():
   return Sum([(s / p) * (i + 1) - (s / p) * i for i in range(p)]) == s

#Ensemble des parties parallèles
def PPs(z):
   glue = And(PP(0, s / p, z), PP(s / p, (s / p)*2, z))
   for i in range(p - 2):
      glue = And(glue, PP((s / p)*(i + 2), (s / p)*(i + 3), z))
   return Implies(glue, x == y)

#Résolution de la preuve
def resoudre(c):
   preuve = Solver()
   preuve.add(Not(c))
   if preuve.check() == sat:
      print "Prouve"
   else:
      print "Non prouve"

print "y = Mot de passe a trouver"
print "x = Mot de passe trouve"
print "s = Nombre total de possibilités"
print "d = Debut du sous-ensemble"
print "f = Fin du sous-ensemble"
print "p = Nombre de processeurs"
print ""
print "Post-condition partie sequentiel"
print PS(y)
print ""
print "Post-condition d'une partie parallèle"
print PP(d, f, y)
print ""
print "Post-condition de la colle"
print C()
print ""
correcte = Implies(And(PPs(y), C()), PS(y))
print "Preuve finale"
print correcte
print ""
print "Resolution..."
resoudre(correcte)
```

Bibliographie

Raphaël Couturier*, Utilisation des méthodes formelles pour le développement de programmes parallèles*, université Henri Poincaré, 2000