Rapport 2^{eme} Période

Jean-Didier Pailleux - Maxence Joulin - Damien Thenot - Romain Robert - Robin Feron

Test de primalité

https://github.com/CHPS-M1-PRIME-NUMBERS/Prime_numbers

01/05/2018

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Projet de Programmation numérique

Table des matières

1 Introduction

Ce document est le compte-rendu de notre travail qui s'inscrit dans le cadre de la deuxième période du module *Projet de Programmation numérique*du master *Calcul Haute Performance Simulation* de l'***UVSQ*** proposé par notre encadrant Sébastien Gougeaud.

Durant la première période de ce projet, il nous a été demandé d'implémenter plusieurs tests de primalité en séquentiel dans le but de déterminer si un grand nombre donné est premier ou non. On rappel qu'un nombre premier est un entier qui admet uniquement deux diviseurs distincts et positifs 1 et lui même. Et qu'il existe deux types de tests de primalité, les déterministes qui permettent d'établir avec certitude le résultat et les tests probabilistes qui émettent un résultat non fiable avec une certaine probabilité d'erreur mais possèdent de meilleures performances que les tests déterministes.

Ce projet est une application de plusieurs algorithmes (AKS, Miller Rabin, Pocklington, Euclide et Eratostène) pour tester la primalité d'un nombre. Après une étude des tests effectués durant la première période sur un échantillon de valeurs, nous avons pu faire plusieurs comparaisons entre ces méthodes et l'ont a pu observer que les méthodes naïves sont très utiles pour les petits nombres mais deviennent très lentes pour les très grands nombres. De plus, Miller-Rabin est un test probabiliste intéressant du fait qu'il ai une exécution très rapide avec une probabilité très faible d'obtenir un résultat erroné. Cependant notre implémentation de l'algorithme de AKS ne fût pas très performante du à l'utilisation de la bibliothèque NTL pour l'arithmétique modulaire.

L'objectif de cette deuxième période consiste en premier lieu d'optimiser si possible les algorithmes utilisés puis de s'occuper de faire tourner plusieurs de ces tests en parallèle et ainsi évaluer la scalabilité de nos implémentations. Le parallélisme de calculs et de données seront utilisés. Pour le parallélisme de données le problème de l'équilibre de charge se posera. De plus pour effectuer nos tests, plusieurs visites à la Maison de la Simulation seront faites pour y faire tourner sur un supercalculateur cette nouvelle version de notre programme.

Dans la première partie de ce document, on présentera les différentes optimisations appliquées sur les tests de notre projet. Après cela, on mentionnera les différents outils et le modèle utilisé au cours de la phase de l'implémentation de la parallélisation dans une partie Parallélisation. Puis dans une autre partie l'analyse des résultats établis lors des tests du projet accompagné d'un comparatif avec sa version séquentielle. Finalement, dans les deux dernières parties, on établira un bilan quant à l'organisation interne au sein du groupe pour cette deuxième période et un bilan technique suite à l'observation des résultats qui mettra en avant les limites des outils utilisés.

2 Optimisation

Pour débuter cette deuxième période il nous fallait d'abord commencer l'optimisation des différents tests de primalité implémentés. Le premier test étant le *eratosthene()* dont le premier problème concerne la mémoire utilisé. En effet un nombre premier ne peut être un nombre paire hormis 2. Pour un très grand nombre N donné, $\frac{N}{2}-1$ nombres entre 1 et N sont paires et non-premier ce qui correspond à un espace mémoire de $4*(\frac{N}{2}-1)$ octets utilisés inutilement dans le tableau de booléen de la fonction *eratosthe()* pour indiqué si un nombre est premier. Pour résoudre cela nous avons donc diviser par deux la taille du tableau utilisé pour ne plus prendre en compte les nombres paires. Cette observation économise 50% de mémoire et est presque deux fois plus rapide que l'algorithme de base tout en ne nécessitant que des modifications mineures du code.
La seconde optimisation de cette fonction concerne la boucle interne. En effet dans la version basique il nous arrive de visiter plusieurs fois la même case du tableau de booléen. Pour cela nous ne faisons plus N itération mais $\sqrt{N}-3$ itérations.

OPTIMIASTION AKS $\rightarrow$ CONJECTURE

POURQUOI ON N'OPTIMISE PAS LES AUTRES

3 Parallélisation

Après avoir optimisé notre code, nous nous sommes consacré à la parallélisation de celui-ci. Deux types de parallélismes nous ont été proposé par notre encadrant. Le premier étant le parallélisme de calculs dans le cas où le(s) test(s) implémenté(s) nous le permettait et le parallélisme de données pour distribuer les données contenu dans une plage au sein des processus disponible lors du lancement du programme et à y opérer les mêmes opérations (les tests de primalité ici).

3.1 Outils utilisés

Pour produire cette nouvelle version du projet, certains outils ont été utilisés pour obtenir le résultat présenté. En premier lieu nous avons utilisé **OpenMP**, un API employé pour le calcul parallèle sur des architectures à mémoire partagée. Nous l'utilisons pour le parallélisme de calculs pour les fonctions de memory_bound() et eratosthene(). L'avantage de ce dernier nous permet de rajouter des "#pragma" sans pour autant modifier le code séquentiel.

Le deuxième outil **MPI**(Message Passing Interface) est une norme définie par une bibliothèque de fonctions utilisé pour le passage de messages entre processus. Le choix d'utiliser MPI provient du fait que cette norme est adaptée pour des machines massivement parallèles à mémoire distribuée, ce qui est le cas du supercalculateur situé à la Maison de la Simulation. Cela nous a donc permis d'exploiter au maximum les ressources mis à disposition lors de l'utilisation de cette machine.

3.2 Modèle utilisé

Modèle du Master Slave.

4 Analyse des résultats

5 Bilan Technique

6 Problèmes rencontrés

Problème avec implémentation si on a

Problème rencontré à la MDLS

7 Organisation interne du groupe

Pour débuter la deuxième période de ce projet, il nous fallait en premier lieu établir une nouvelle répartition du travail de groupe pour que le projet puisse avancer de façon efficace et de manière rapide. Le tableau ci-dessous va ainsi indiquer pour chaque membre du groupe la ou les fonctionnalités pour laquelle il a pu contribuer à l'élaboration :

Tâches	Jean-Didier	Maxence	Romain	Robin	Damien
Master slave	x	x			
Optimisation Memory Bound	x				
AKS Conjecture			x		
Makefile			x		x
Rapport					
Tests	x	x	x	x	x

8 Conclusion