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

06/05/2018

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.

AKS étant un algorithme qui prenait beaucoup de temps a s'exécuter nous avons décidé de l'optimiser. Pour cela nous avons d'abord effectué un profilage du code avec *gprof* pour observer quelles sont les fonctions qui prennent le plus de temps dans l'exécution. Sans surprise la partie 5 de l'algorithme (la seule partie qui utilise NTL) occupe quasiment la totalité du temps d'exécution. Le tableau ci-dessous montre un aperçu du profil de AKS :

% time	cumulative secondes	self seconds	calls	total ms/Call	ms/call	name
100.15	0.01	0.01	213844	0.00	0.00	NTL : :operator==
0.00	0.01	0.00	427689	0.00	0.00	NTL : :WrappedPtr
0.00	0.01	0.00	213844	0.00	0.00	NTL : :operator !=
0.00	0.01	0.00	74203	0.00	0.00	unsigned long modpow()

Nous avons donc choisi d'implémenter une variante d'AKS en sélectionnant celle qui serait la plus faible en terme de complexité et qui utiliserais le moins possible la bibliothèque NTL. Pour rappel NTL est une bibliothèque haute performance qui nous permet d'implémenter facilement des calculs modulaire sur les polynômes. Une variante sortait du lot : la Conjecture d'Agrawals. C'est un algorithme très performant avec une complexité en $O(logn)^3$. Cependant ce dernier est non prouvé car à partir d'un certain rang il existerait une infinité de contres exemples. Un projet appelé *distributed computing project Primaboinca* a pour but de trouver les contres exemples de cette conjecture et affirme que leur inexistence pour $n < 10^{12}$. Or notre projet étant limité a 2^{64} nous pouvons donc utiliser cet algorithme sans risque d'avoir des résultats faussé.

Après optimisation on obtient un algorithme efficace qui nous permet de tester des grands nombres dans un temps raisonnable, avec cet optimisation on passe d'une complexité de $O(logn)^{12}$ (AKS 2002) à $O(logn)^3$. Voici l'algorithme :

Algorithm 1 AKS Conjecture

a. Vérifier si n n'est pas un Perfect power
b. r = 2
c. Trouver r, tel que r ne divise pas $n^2 - 1$
d. Vérifier que $(x-1)^n \equiv x^n - 1 \pmod{x^r - 1, n}$

Ensuite nous avons l'algorithme de Pocklington qui repose sur la factorisation en nombres premier, qui même avec un algorithme optimisé tel que le crible quadratique, se trouve être un problème appartenant à la classe NP. C'est-à-dire qu'il appartient aux problèmes les plus difficile en informatique car ceux-ci augmentent très rapidement en temps d'exécution en fonction de l'entrée de façon exponentielle. De plus, l'implémentation des algorithmes de factorisation en nombres premier plus efficace représente un problème car ils sont basés sur des principes mathématiques difficile à implémenter. Enfin l'algorithme de Miller-Rabin ne peut être d'avantage optimisé.

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 Parallélisme de calculs

Pour le parallélisme de calculs plusieurs algorithmes tels que AKS et la conjecture ne supportent pas ce type de parallélisme. Pour le test de Miller-Rabin nous avons paralléliser les k itérations per-

mettant d'affiner la précision de l'algorithme, après cette parallélisation nous avons effectué des tests de performances, ces tests montraient que cette modification n'améliorait pas le temps d'exécution de l'algorithme mais qu'au contraire elle l'augmentait. Pour cette implémentation nous avions remplacer les valeurs de retours qui étaient des booléens par des entier 0 pour vrai et 1 pour false, ainsi les différents cœurs se répartissant les itérations faisaient la somme des valeurs de retours et lorsque cette somme était supérieur à 0 le nombre testé n'était donc pas premier. L'idée était donc ensuite d'utiliser la version optimal de Miller-Rabin avec le système master/slave pour avoir de meilleur temps.

Une parallélisation possible de Pocklington semblait intéressante. Dans un premier temps nous avons utilisé OpenMP pour la parallélisation des tests sur les facteurs premier $N-1$. Cette tentative nous donna cependant aucun résultat intéressant. En effet, cela nous a donc fait perdre la possibilité d'arrêter l'exécution lorsque l'une des vérifications était fausse. La majeur partie du temps de calcul se trouvant dans la factorisation en nombre premier de $N-1$, la parallélisation des tests sur les facteurs premiers et la perte de la fin sur une règle non respecté ne donne pas de résultats intéressant.

Enfin le crible d'Ératosthène (Memory Bound) est donc le seul test dont nous avons implémenté un parallélisme de calculs. Pour paralléliser cette fonction nous avons utilisé OpenMp. Pour cela nous avons placé un *#pragma omp parallel for* pour paralléliser l'initialisation du tableau de booléen, puis *#pragma omp parallel for schedule(dynamic)* pour la partie appliquant l'algorithme du crible. L'utilisation de "schedule(dynamic)" indique qu'OpenMP affecte une itération à chaque thread. Lorsque le thread finit, il lui sera affecté l'itération suivante qui n'a pas encore été exécutée. Cela permet donc d'avoir une bonne répartition des tâches sur l'ensemble des threads car on suppose que le temps de traitement pour une itération n'est pas constant.

3.3 Parallélisme de données

La majorité de nos tests n'étant pas parallélisé il fallait donc opter pour un autre type de parallélisme qui est celui de données. Pour cela nous sommes donc parti sur un modèle basé sur le Master Slave (Maître-Esclave). C'est à dire qu'un processus sera désigné en tant que "Maître" et sera chargé de distribué du travail aux esclaves qui vont donc demander à l'inverse du travail au maître qu'il sera libre. Le choix de ce modèle vient du fait qu'il est principalement utilisé pour une application décomposable en différentes tâches indépendantes (chaque test de primalité étant indépendant). De plus il est nécessaire de faire attention à l'équilibre de charges car lorsque nous donnons une plage de données à analyser il nous est impossible de prévoir à l'avance le temps mis pour le traitement d'un nombre et donc de répartir de façon équitable le travail sur les processus. Pour notre implémentation le processus Maître correspond au rang 0, il ne lancera jamais de tests de primalité. Le rang maître envoi seulement un paquet d'un seul entier. Des tentatives d'envoyer des paquets de 10, 100 ou plus n'ont pas influencé le temps d'exécution final du programme. Cela peut se justifier par le fait que le temps de communication d'un message MPI était négligeable par rapport au reste (de l'ordre de $10^{-}4$s).

4 Analyse des résultats

5 Bilan Technique

6 Problèmes rencontrés

Lors de nos séances à la maison de la simulation, nous avons été dans l'impossibilité d'effectuer nos tests en raison d'un problème technique au niveau des noeuds de calcul qui n'étaient plus disponibles. A la première séance nous avons eu besoin d'implémenter un Makefile, nous utilisions à la base cmake qui s'occupait alors de créer le Makefile pour nous mais cmake n'était pas compatible avec les machines auquel nous avion accès à la maison de la simulation, lors de l'implémentation de ce Makefile nous avons rencontrer des problèmes d'intégration des bibliothèques NTL et GMP, ce qui nous a retardé dans les tests. Enfin, lors de notre dernière séance certains de nos résultats peuvent avoir été faussé causé par des nœuds qui n'étaient pas sain (un temps d'exécution de $300s$ au lieu de $25s$ par exemple).

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			
Parallélisation /Optimisation Memory Bound	x				
AKS Conjecture			x		
Tests parallélisation Miller-Rabin		x		x	
Makefile			x	x	x
Rapport	x	x	x	x	x
Tests	x	x	x	x	x

8 Conclusion

En conclusion, l'utilisation de techniques de parallélisation et d'optimisation montre les même résultats que nous avons déjà observé, les algorithmes les plus rapides permettent d'obtenir des résultats très rapidement sur une plage de données. Et la parallélisation mise en place permet de traiter une grande plage de données rapidement sur une machine parallèle mais les performances dépendent très fortement des performances individuelles des algorithmes quand on se trouve à 100% des ressources alloués utilisés.