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 dès qu'il est 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 (avec 4 processus seulement 3 lancerons des 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

Pour cette deuxième période du projet, l'accès au cluster **Poincare** disponible à la Maison de la Simulation du CEA nous a permis de faire plusieurs vague de tests de scalabilité sur la version parallèle de notre projet.

4.1 Lancement d'un test

Pour commencer ces tests nous devions charger les modules indispensables à l'utilisation de notre programme dans l'environnement utilisé avec *module load tool*. Les modules utilisé lors des tests sont *gnu-env/5.4.0* et *openmpi/1.10.0_gnu54*. Lancer un test sur Poincare nécessite d'utiliser un script nommé *batch.sh*, celui-ci indique donc pour un job lancé, le nombre de processus MPI (total_tasks), le nombre de nœuds nécessaire (node, 16 tâches MPI pour 1 nœud), la durée maximum avant que le job soit tué (wall_clock_limit), le type de job (job_type = mpich si parallel ou serial sinon),... Pour soumettre un tel job la commande *llsubmit batch.sh* est utilisé, cependant les sorties standard n'existant pas sur le cluster, des fichier job_name.err et job_name.log servent donc à les remplacer. Voici un exemple de fichier batch.sh :

```
#!/bin/bash
#@ class            = clallmds+
#@ job_name         = JOB1
#@ total_tasks      = 16
#@ node             = 1
#@ as_limit         = 7gb
#@ node_usage       = not_shared
#
#@ wall_clock_limit = 02:00:00
#@ output           = $(job_name).$(jobid).log
#@ error            = $(job_name).$(jobid).err
#@ job_type         = mpich
#@ environment      = COPY_ALL
#@ queue
#

module load gnu-env/5.4.0  openmpi/1.10.0_gnu54
export OMP_NUM_THREADS=4
mpirun -x OMP_NUM_THREADS -report-bindings ./prog
```

FIGURE 1 – Exemple de fichier batch.sh

4.2 Résultats

Tout d'abord nous avons réalisé une première batterie de tests sur une première plage de nombres allant de 1 à 1 million avec 10 itérations. Les **Figure 2, 3 et 4** représentent donc l'évolution du temps de calcul en fonction du nombre processus sur cette plage. Cependant deux tests de primalité

n'ont pas eu de résultats du à un manque de temps et aussi à leur problème de performance. Pour la première plage avec $4h$ donné lors de la soumission du job Pocklington et AKS n'ont pas pu finir leur exécution avant d'être tué par le supercalculateur quelque soit le nombre de processus donné.

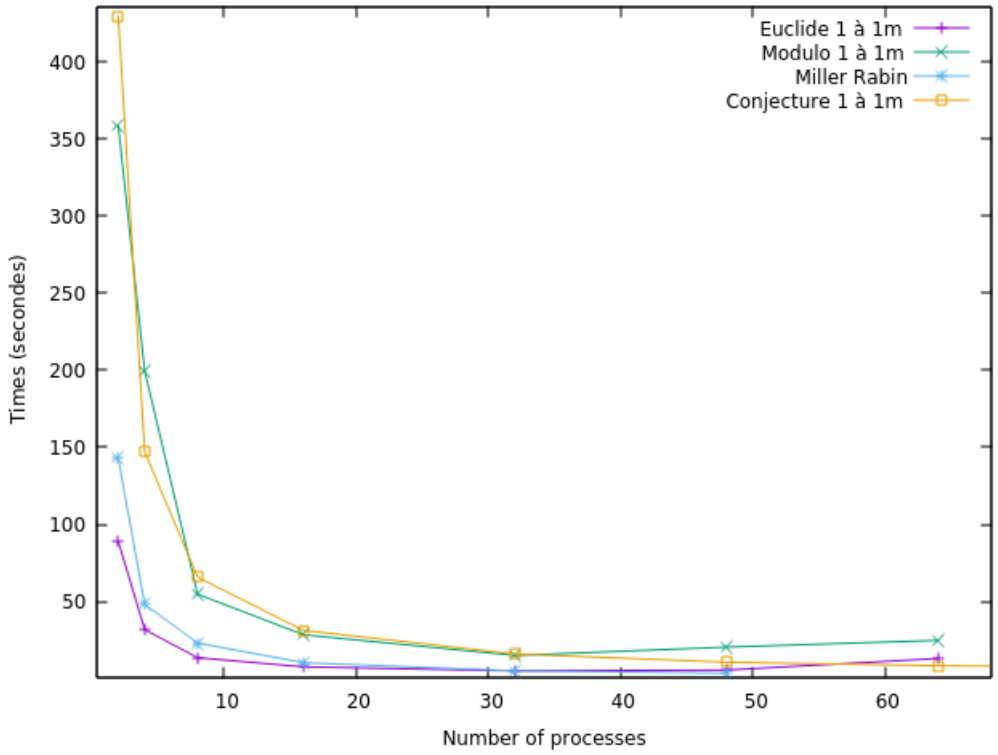
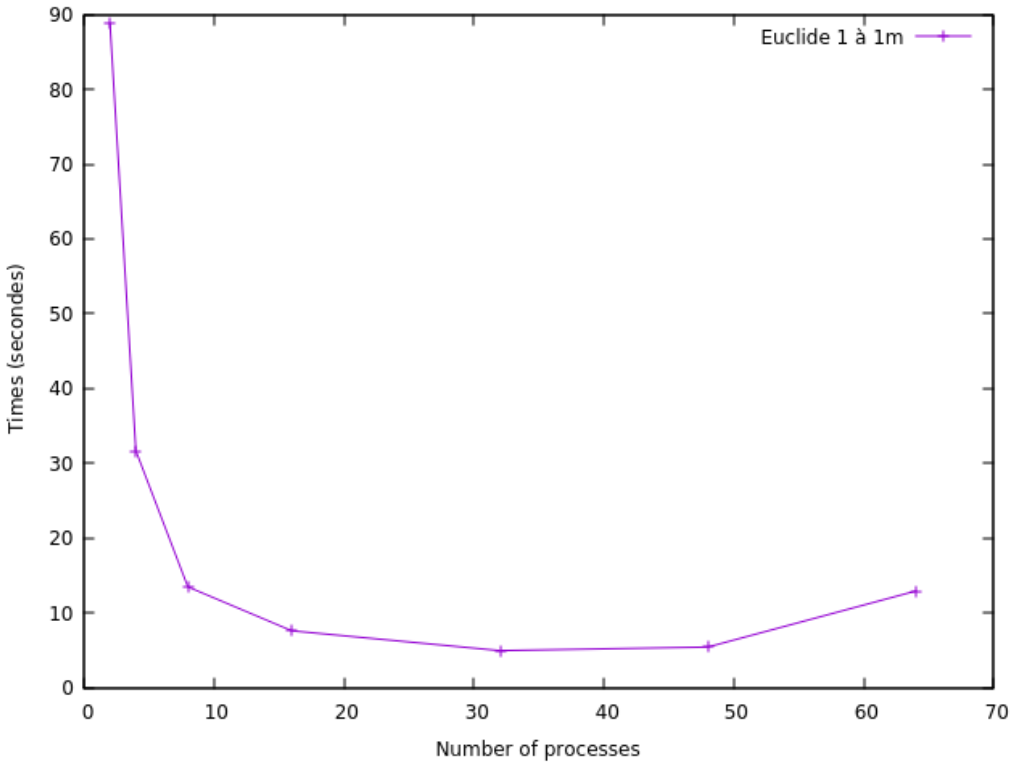


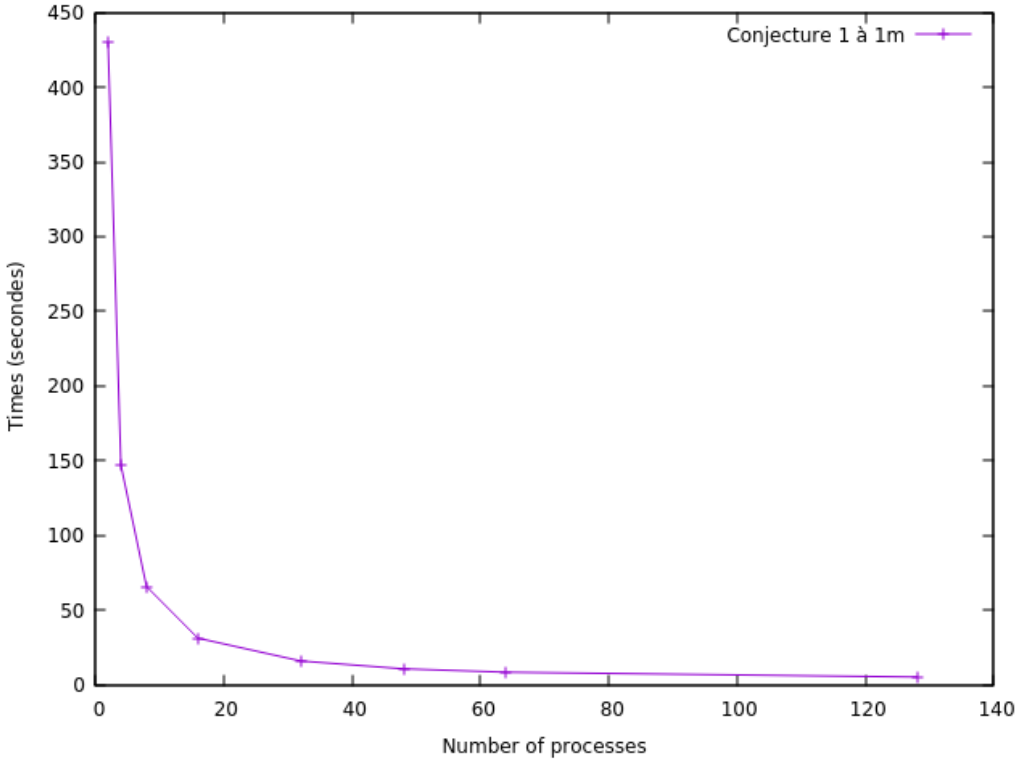
FIGURE 2 – Évolution du temps de calcul pour Conjecture, Miller-Rabin, Euclide et Modulo.

D'après ces résultats nous pouvons voir que l'implémentation du modèle Master-Slave montre une bonne scalabilité. De plus, la **Figure 2** indique que lorsque nous augmentons le nombre de processus les tests les plus lent tel que la Conjecture ont un temps d'exécution équivalent aux tests les plus rapides. En séquentiel nous avons la Conjecture qui tourne avec 424.5s en moyenne, Miller-Rabin avec 129.1s, Euclide avec 89.9s et Modulo 375s. Nous pouvons donc voir que dès 3 processus le programme exécute plus rapidement la plage de données. Cependant en ce qui concerne la plage [1, 1 000 000] nous pouvons observer qu'au bout d'un certain nombre de processus il devient peu avantageux d'augmenter ce nombre. La cause de cette augmentation serait les communications MPI qui prennent le dessus sur le temps d'exécution final. En effet, les nombres de cette plage sont très petits, l'exécution d'un test serait donc beaucoup trop rapide.

Ensuite nous avons réalisé une deuxième vague de tests sur une deuxième plage de nombres allant de 4 Milliard à 4 Milliard 1 million (4M1m) avec 10 itérations. Les **Figure 5 et 6** représentent donc l'évolution du temps de calcul en fonction du nombre processus sur cette plage. Par manque de temps le test de Miller Rabin n'a pu avoir avoir de résultat pour cette plage de données.

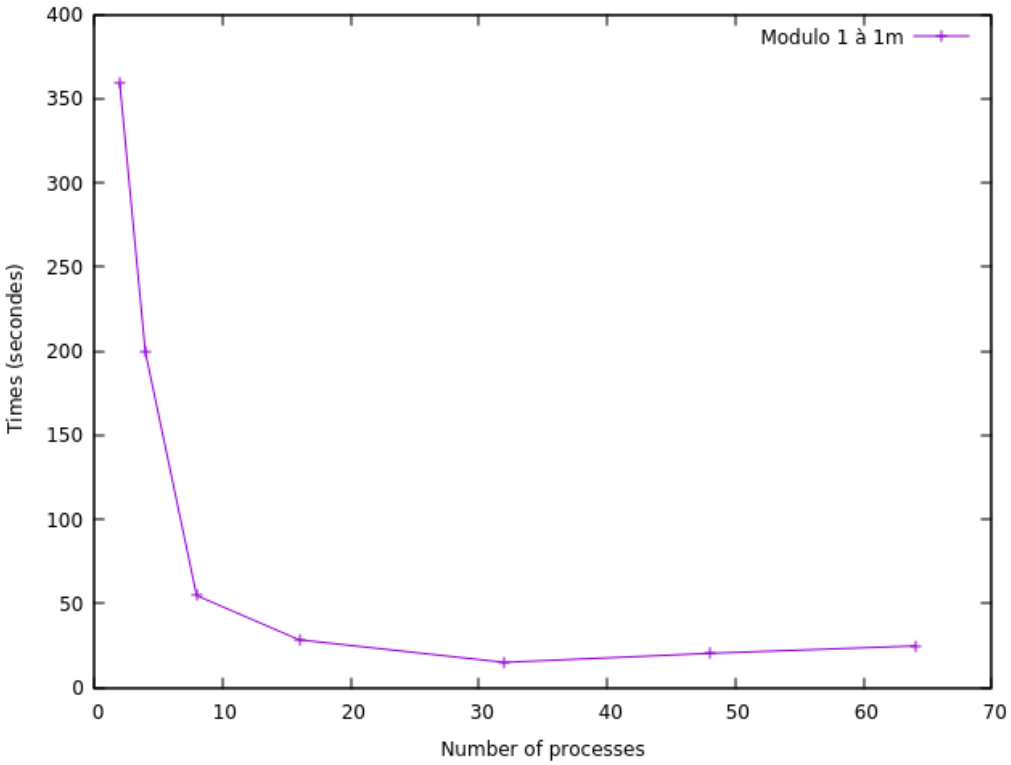


(a) Euclide

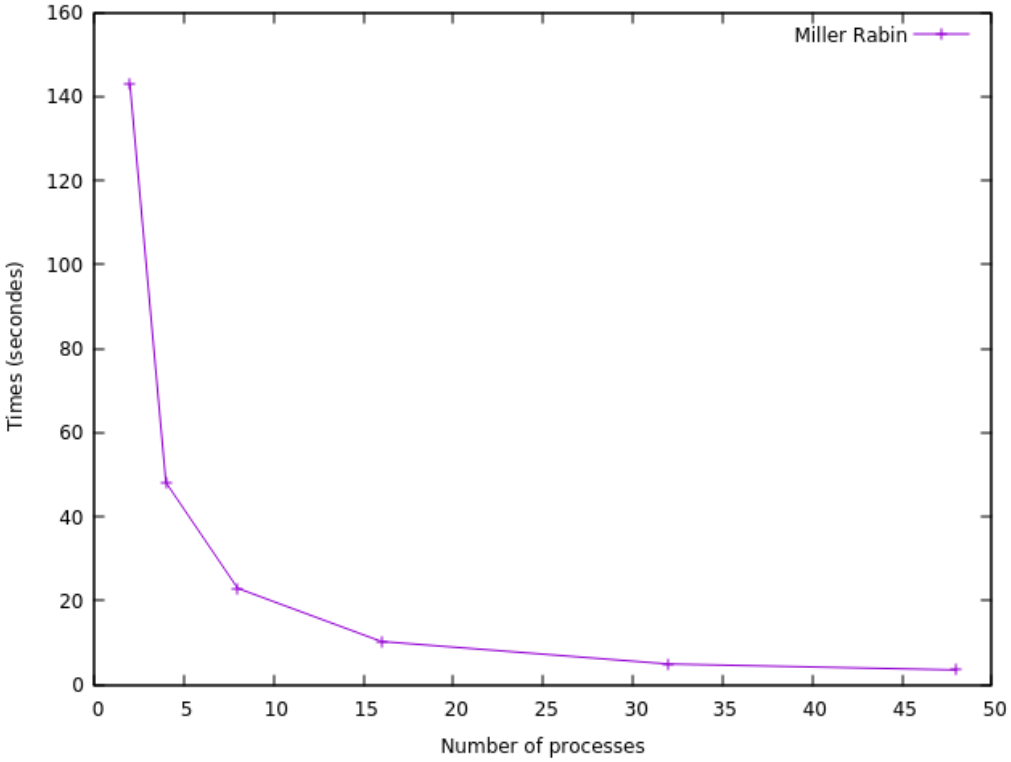


(b) AKS Conjecture

FIGURE 3 – Évolution du temps de calcul en fonction du nombre processus : Plage [1, 1 000 000]

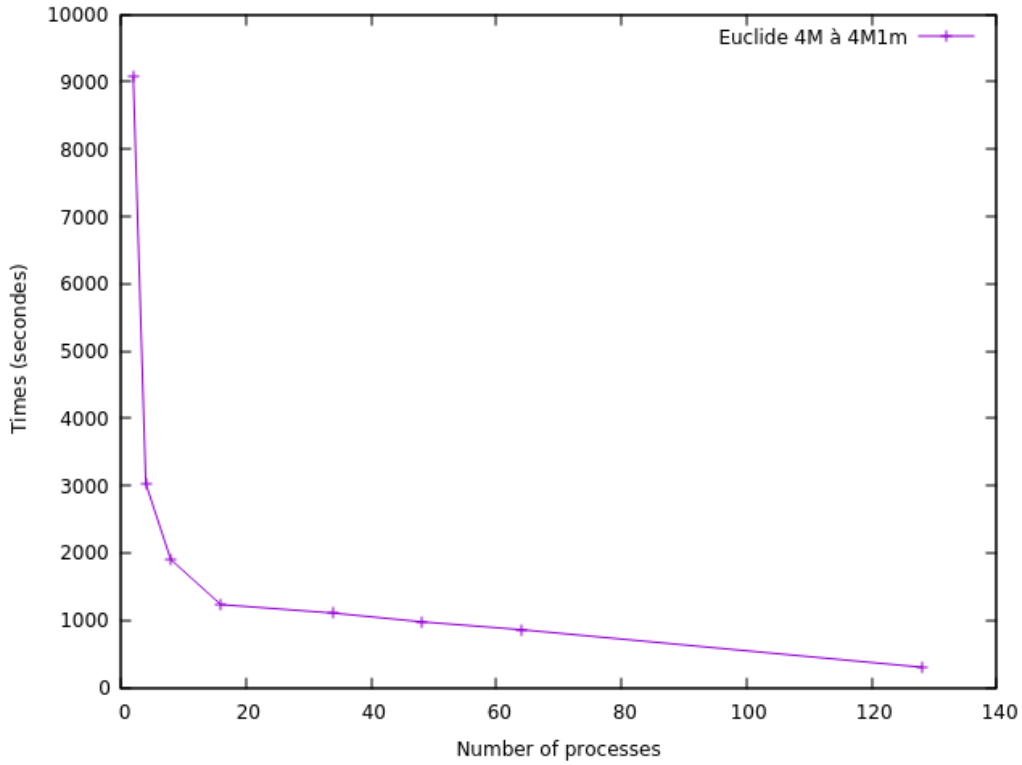


(a) Modulo

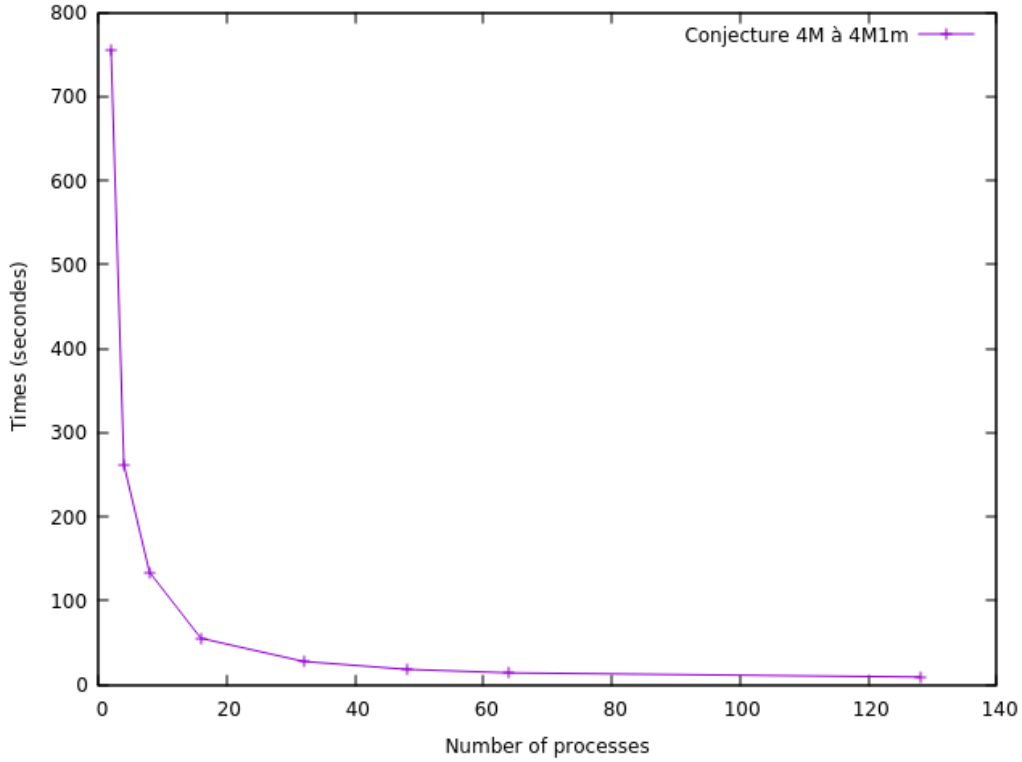


(b) Miller-Rabin

FIGURE 4 – Évolution du temps de calcul en fonction du nombre processus : Plage [1, 1 000 000]



(a) Euclide



(b) AKS Conjecture

FIGURE 5 – Évolution du temps de calcul en fonction du nombre processus : Plage [4M, 4M1m]

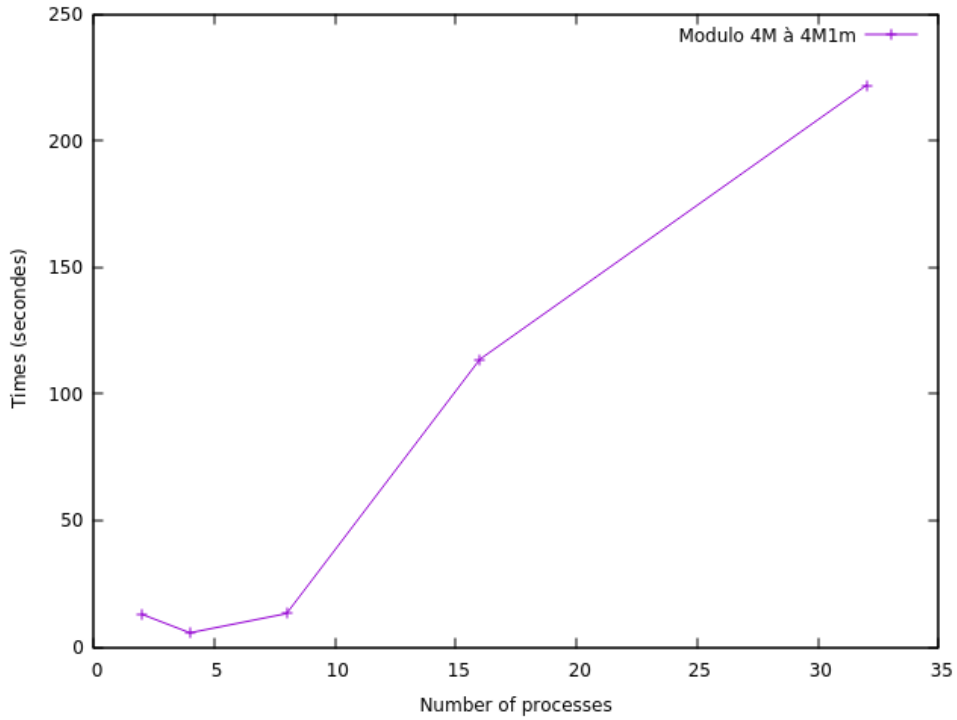


FIGURE 6 – Évolution du temps de calcul en fonction du nombre processus Modulo : Plage [4M, 4M1m]

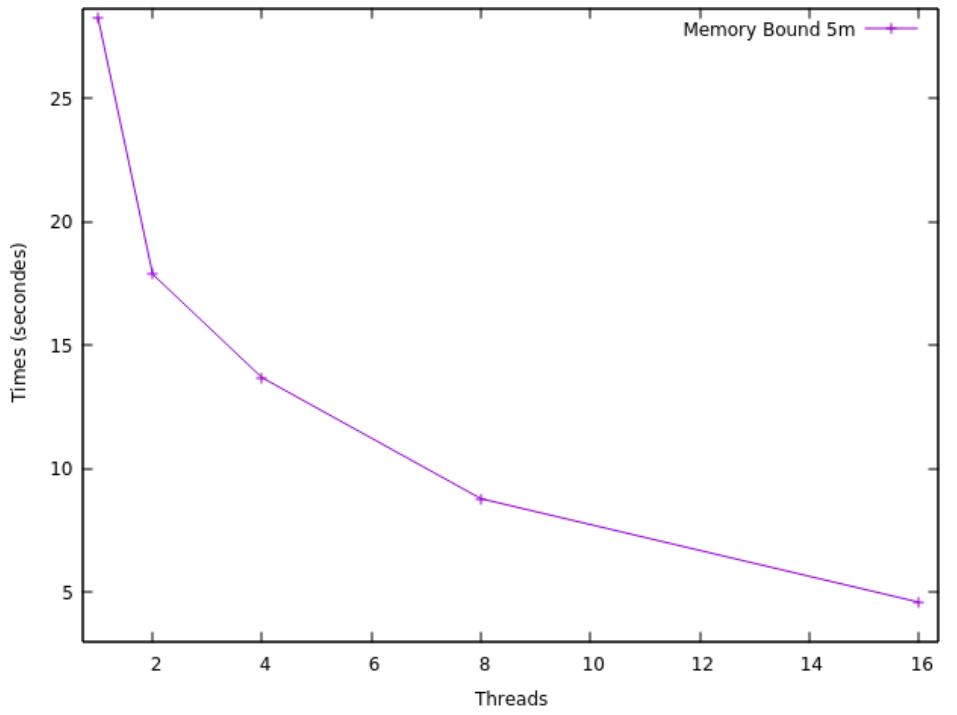


FIGURE 7 – Évolution du temps de calcul en fonction du nombre processus pour Memory Bound : Plage [1, 500 000 000]

Pour cette plage de données, nous pouvons voir que l'implémentation possède toujours une bonne scalabilité à l'exception du résultat pour Modulo. D'après notre encadrant Matthieu Haefele la cause de ce résultat serait causé par l'utilisation de nœuds qui ne seraient pas sains lors de nos tests. Ce résultat ne serait donc pas causé par notre implémentation et donc ne serait pas représentatif.

Les tests concernant MemoryBound ont été fait à part des autres étant donné que ce test utilise un autre type de parallélisme et qu'il calcul les nombres premier compris entre 1 et N. Les résultats de la parallélisation associé à ce test nous indique qu'il supporte plutôt bien le parallélisme de calculs même si la scalabilité affiché de celui-ci n'est pas parfaite. De plus, l'optimisation faite nous permet donc d'économiser de la mémoire au niveau de la RAM ce qui nous autorise d'utiliser ce test pour de plus grand nombre.

5 Bilan Technique

Suite à nos résultats nous pouvons conclure que notre implémentation d'une communication Maitre-Esclave pour paralléliser nos recherches de nombres premiers est relativement efficace. L'augmentation du nombre de cœur de calcul diminue bien par la même échelle le temps nécessaire à la recherche des nombres premiers d'une plage donnée. On remarque cependant que cette augmentation de l'efficacité atteins un palier pour la plupart de nos tests. Selon notre analyse des temps passés dans les différentes parties de notre programme, il semblerait que le processus Maître, qui envoi les données à traiter aux processus Esclaves, soit le responsable de ce plafonnement.

En effet avec l'augmentation du nombre de cœur de calculs, le nombre de communications que doit effectuer le processus Maître en un instant T augmente considérablement afin d'assurer le fait qu'aucun processus n'attende trop longtemps un nouveau travail. Or le calcul d'une donnée étant relativement court, le processus Maître arrive à saturation avec l'augmentation du nombre de nœud de calcul. Notre schéma de communication arrive donc à ses limites dû au fait que des processus Esclaves vont certainement ne pas être utilisé 100% du temps.

Une optimisation de notre schéma de communication aurait pu être d'envoyer des centaines voir des milliers de données à traiter à chaque processus Esclave en un envoi. Ceci aurait pu permettre de diminuer le nombre de communication nécessaire de la part du processus Maître et ainsi le rendre plus réactif à la demande de travail d'un nouveau processus Esclave. Cette implémentation n'a été testé que sur nos ordinateurs personnels (avec seulement 8 cœurs) et n'a cependant pas donné de résultats probant en gain de performance.

Une autre implémentation aurait pu être d'augmenter le nombre de processus Maître. En effet en divisant la taille du problème en plusieurs processus Maîtres, on aurait diviser d'autant la charge qui pèse sur leur épaules.

Concernant l'équilibrage des charges sur nos processeurs, il est important de rappeler que nos calculs sont très courts, de l'ordre d'un centième de seconde pour la plupart de nos algorithmes d'évaluation. Ainsi un cœur ne va pas prendre beaucoup plus de temps qu'un autre pour analyser sa donnée même si celle çi demande un peu plus de calculs pour déterminer sa primalité. Notre équilibrage de charge est donc presque parfait, de l'ordre de $1/10$ème de seconde de différence entre les deux processus les plus éloignés en terme de fin d'exécution pour 40 secondes de temps total d'exécution. Donc aucun équilibrage compliqué n'a été nécessaire.

Le problème de recherche des nombres premiers dans une plage de donnée est donc hautement parallélisable et notre implementation en Maitre/Esclave est efficace. Cependant la limite de ce problème réside avant tout dans la non parallélisation des tests en eux-même du à leur forme très simple. Les limites techniques de rapidité de calculs sont donc rapidement atteintes et seul l'augmentation du volume de donnée laisserait entrevoir l'utilité de l'augmentation du nombre de noeud de calcul et leur bonne utilisation.

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 rencontré 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

Pour terminer, l'utilisation de techniques de parallélisation et d'optimisation nous montre les mêmes résultats que nous avons déjà pu observer, 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. Cependant les performances observé dépendent très fortement des performances individuelles des algorithmes quand on se trouve à 100% des ressources alloués utilisés ou de l'état de "santé" des nœud utilisés.

Pour cette deuxième partie nous avons malheureusement uniquement étudié des plages de données. Nous aurions pu explorer d'autres options comme la comparaison de nos algorithmes avec des implémentations déjà existantes mais des problèmes rencontrés à la Maison de la Simulation nous ont limité dans nos tests. Malgré cela, le résultat final reste satisfaisant, la partie parallèle est efficace, l'équilibrage de charge est très bien respecté et les optimisations effectuées ont bien amélioré les temps d'exécution.

Le groupe s'est une nouvelle fois très bien entendu tout au long du projet, l'entraide était vraiment présente. Pour élargir ce sujet il serait possible d'implémenter d'autres algorithmes efficace comme ECPP(Elliptic curve primality test) ou encore APR (Adleman-Pomerance-Rumely primality test) des équivalents de l'algorithme AKS avec de bien meilleurs résultats pratiques. Mais également à d'autres schémas de parallélisation (*ex : avoir plusieurs processus maître*) du programme pour comparer les performances.

Ce projet nous aura donc permis d'affiner et d'améliorer nos connaissances sur la parallélisation et l'optimisation d'algorithmes, mais il nous aura également permis de découvrir et d'utiliser du matériel spécialisé et performant tel que le cluster Poincare du CEA.