République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université A . Mira de Béjaia

Faculté des Sciences Exactes

Département d'Informatique

Mémoire de fin de cycle

En vue de l'obtention du diplôme de Master professionnel en Informatique

Option : Administration et securité des réseaux

Thème
Implémentation des méthodes Durand-Kerner et Ehrlish-Aberth en C avec OpenMP et CUDA

Réalisé par :

AMROUCHE Bachir Aghiles

YALA Louisa

Devant le jury composé de :

Présidente :	M^{me} *BENSAID* Samia	M.A.A à l'université A. MIRA de Bejaia
Examinatrice :	M^{lle} *BEN BOUDAOUD* Lynda	Doctorante à l'université A. MIRA de Bejaia
Encadrant :	D^{r} *SIDER* Abderrahmane	M.C.B à l'université A. MIRA de Bejaia

Année universitaire 2013/2014

REMERCIEMENTS

En premier lieu, je remercie LouiSa YALA pour toutes les nuits blanches et le travail que nous avons accomplis ensemble dans ce mémoire.

Je remercie D^r. Sider , notre encadrant pour la réalisation de ce présent mémoire, qui nous a orientés dans ce domaine inconnu jusqu'ici. Il nous a guidés dans notre travail et nous a aidés à trouver des solutions appropriées pour avancer dans ce travail.

J'adresse mes remerciements aux personnes qui m'ont aidé dans la réalisation de ce mémoire.

Je remercie tous les professeurs du Département d'informatique de l'Université de Béjaïa , ainsi que tous les étudiants avec qui j'ai vécu ces 5 dernières années, particulièrement : A.DINA, A.AMINE, F.YASMINE, M.MOUNIR, K.SOFIANE, K.AMINA et tous les autres qui ont partagé avec moi les bancs de l'amphi *13*.

AMROUCHE B.Aghiles

DEDICACES

A mes très chers parents qui ont toujours été proches de moi et j'espère qu'un jour ils comprendront tant bien que mal la portée de mes études.

A la mémoire de ma chère cousine LILYA (alias ILHAM), qui nous manquera pour toujours.

A mes chers deux sœurs et à mon beau-frère, bien que loin de moi mais qui ont quand bien même continué a m'encourager dans tout ce que j'entreprenais.

A ma chère LYZÄ, le meilleur ami de l'homme.

A tous mes AMIS, surtout ceux qui étaient persuadés que j'allais concevoir un jeu-vidéo et tous les autres qui n'ont pas compris un mot de ce travail.

A chaque cousin et cousine, proche et éloigné qui ne liront probablement pas ce document.

et le meilleur pour la fin,A ma binôme sans qui rien de ceci n'aurait été.

Je dédie ce mémoire.

AMROUCHE B.Aghiles

REMERCIEMENTS

En premier lieu, je tiens à remercier Dieu de m'avoir donné la force et la patience pour pouvoir mener ce travail à terme.

Je tiens, à exprimer ma profonde gratitude à monsieur SIDER Abderahmane, Docteur à l'université de Béjaia, qui a encadré le présent travail. Qu'il veuille bien trouver ici l'expression de ma reconnaissance pour son dévouement, sa patience, sa disponibilité, ses conseils et son aide constante qu'il a apporté tout au long de ce travail.

Je remercie les membres du jury Monsieur ALOUI Abdelouhab, Madame BENSAID Samia et Mademoiselle BEN BOUDAOUD Lynda d'avoir accepté de juger et d'évaluer ce travail.

J'adresse mes vifs remerciements à tous les enseignants qui, par leur enseignement, leur encouragement et leur aide, ont contribué à ma formation.

Je remercie mes cousines Sarah pour son aide et Meriem pour son soutien.

Je remercie chaleureusement mes parents pour leur amour, leurs sacrifices en témoignage de mon éternelle reconnaissance.

Je remercie ma soeur Lila et mon frère Ali pour leur soutien.

À tous ceux qui, par leur encouragement ou leur amitié, ont contribué à l'aboutissement de ce travail. Enfin, À mon binôme depuis 5 ans Aghiles

Louisa YALA

DEDICACES

À mes adorables grands parents.

À la mémoire de Tonton Samir.

À mes parents.

À ma Tata Hassina.

À mes oncles et mes tantes.

À mes cousins et cousines.

À mes amis.

Aux personnes qui m'ont toujours aidé et encouragé.

Louisa YALA

Table des matières

Table des figures

Liste des tableaux

Liste des Acronymes

ALU	**A**rithmetic **L**ogical et **U**nit .
API	**A**pplication **P**rogramming **I**nterface.
CIP	**C**alcul **I**nformatique de **P**ointe .
CPU	**C**entral **P**rocess **U**nit .
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture .
DK	**D**urand-**K**erner .
DRAM	**D**ynamic-**R**adom **A**ccess **M**emory.
EA	**E**hrlish-**A**berth .
EDP	**É**quation à **D**érivée **P**artielle .
GLSL	Open**GL** **S**hading **L**anguage .
GPGPU	**G**eneral-Purpose Processing on **G**raphics **P**rocessing **U**nits .
GPU	**G**raphic **P**rocess **U**nit.
HLSL	**H**igh **L**evel **S**hader **L**anguage .
HPC	**H**igh **P**erformance **c**omputing .
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers .
MIMD	**M**ultiple **I**nstruction **M**ultiple **D**ata .
MISD	**M**ultiple **I**nstruction **S**ingle **D**ata .
MPMD	**M**ultiple **P**rogram **M**ultiple **D**ata .
OpenCL	**O**pen **C**omputing **L**anguage .
OpenGL	**Open G**raphics **L**ibrary .
OpenMP	**Open M**ulti-**P**rocessing .
RAM	**R**andom **A**ccess **M**emory .
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata .
SISD	**S**ingle **I**nstruction **S**ingle **D**ata .

SPDM	**S**ingle **P**rogram **M**ultiple **D**ata .
UAL	**U**nité **A**rithmétique et **L**ogique

INTRODUCTION GÉNÉRALE

Depuis la naissance de l'informatique , nous ne cessons de chercher la réponse à la question ¨*Comment résoudre plus rapidement des problèmes coûteux en temps de calcul ?*˝ : dans la simulation numérique, la cryptographie ou l'imagerie . L'une des réponses étant dans le calcul scientifique qui est une discipline aux contours, pas toujours franchement définis, mais qui regroupe un ensemble de champs mathématiques et informatiques permettant la simulation numérique des phénomènes de la physique, chimie, biologie, et sciences appliquées en général. Son corollaire, la simulation numérique, fournit un outil efficace pour l'expérimentation afin de prédire, comprendre, optimiser, voir et contrôler le comportement des systèmes physiques relevant des sciences de l'ingénierie [9]. Pour ce faire, des programmes sont réalisés par des calculs informatiques de haute performance permettant leur modélisation au plus proche de la réalité, que ce soit des expériences trop grandes telles que dans l'$astrophysique$ ou trop petite comme dans la $micro\text{-}biologie$ ou bien pour une expérience trop coûteuse comme les $crash\text{-}tests$.

Les programmes ont été conçus historiquement pour une exécution séquentielle. Les programmes devaient s'exécuter sur une seule machine, contenant une seule unité de traitement centrale (CPU) et le problème est décomposé en une suite d'instructions qui sont exécutées les unes après les autres. Ainsi, une seule instruction peut être exécutée à la fois. Le calcul parallèle est le contraire de la précédente approche, l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème. Un programme peut ainsi s'exécuter sur plusieurs $CPUs$. Le problème est décomposé en plusieurs parties qui peuvent être résolues de manière concurrente. Ces parties sont à leur tour décomposées en plusieurs instructions et chaque paquet d'instructions s'exécute de manière indépendante l'un de l'autre. Les ressources de calcul incluent une seule machine avec plusieurs processeurs, un nombre arbitraire de machines connectées via un réseau ou alors une combinaison des deux. Une bonne partie des problèmes de calcul intensif possède certaines caractéristiques qui en font de bons candidats à la parallélisation. Parmi ces caractéristiques : la possibilité de les décomposer en plusieurs sous-problèmes qui peuvent être résolus simultanément et la possibilité d'être résolus en moins de temps avec plusieurs ressources qu'avec une seule. Le calcul parallèle était auparavant, réservé exclusivement à la

modélisation de problèmes et de phénomènes scientifiques. A ce jour, le calcul parallèle s'est démocratisé grâce notamment à l'évolution fulgurante de la technologie des semi-conducteurs qui a rendu les plate-formes haute performance plus accessibles. Nous pouvons citer des applications comme les applications graphiques GPU et de réalité augmentée.

La puissance des ordinateurs ne cesse d'augmenter jour après jour, et la loi de $Moore$ [1] reste encore et toujours valide. Le fait que nos ordinateurs personnels possèdent la puissance des supercalculateurs datant de quelques années est assez époustouflant.Cet avancement matériel ne cesse de pousser les programmes informatiques vers de nouveaux horizons et de nouvelles normes. En effet, certaines applications qui gèrent un assez grand nombre de données et de contraintes comme la météorologie et les simulations nucléaires atteignent rapidement les limites des machines existantes. C'est dans ce genre d'applications que s'institue l'informatique du calcul intensif ou HPC. Cette branche de l'informatique s'intéresse aux outils d'accélération de programmes comme la programmation multi-threads et multi-cœur ($OpenMP$), la programmation multi-processus et multi-machines (MPI) ainsi que la programmation sur carte graphique ($CUDA$/$OpenCL$).

Dans ce mémoire, nous présentons notre travail qui consiste à implémenter deux méthodes d'extraction de racines de polynôme $Durand$-$Kerner$ et $Ehrlish$-$Aberth$, les paralléliser sur CPU avec $OpenMp$ et sur GPU avec $CUDA\ C$. L'objectif est d'arriver à de bons résultats dans un temps d'exécution raisonnable. Pour celà nous avons dans le premier chapitre défini le calcul scientifique et ses domaines d'application, les architectures paralléles, suivies de la parallélisation et modèle de programmation,puis des cartes graphique $NVIDIA$ et enfin $CUDA$. Dans le deuxième chapitre nous expliquons en détails les deux méthodes étudiées et implémentées. Le troisième et dernier chapitre est en premier lieu, consacré à la présentation de l'environnement de travail sur lequel est basée la programmation des deux méthodes et de la machine utilisée , en deuxième lieu,à l'explication de l'implémentation des deux méthodes, et en troisième lieu à l'interprétation des résultats de chaque exécution des deux méthodes y compris l'implémentation paralléle avec $OpenMp$ et $CUDA$ qui sont évaluées selon les résultats obtenus par différents degrés de polynôme, après une analyse visuelle grace à l'outil graphique $GnuPlot$. Nous terminerons par une conclusion generale et perspectives.

1. datant de 1965, son application dit Qu'un PC acheté en 2003 soit à la fois cinq fois moins cher, dix fois moins lourd, cent fois plus puissant et beaucoup plus ergonomique que notre premier ordinateur

Chapitre 1

Etat de l'art

1.1 Introduction

Dans ce premier chapitre, nous définissons d'abord le calcul scientifique, ses étapes et ses domaines d'applications, suivi par les architectures parallèles et les catégories de la classification de *Flynn*, puis de la parallélisation et modèle de programmation, enfin nous introduisons les cartes graphiques *NVIDIA* et *CUDA* avec un historique de leur avènement.

1.2 Calcul scientifique

Le calcul scientifique est par essence un domaine interdisciplinaire, tant au sein des sciences en général qu'au sein des mathématiques, puisqu'il repose sur des champs aussi divers que l'analyse, l'analyse numérique, la théorie des probabilités, etc. Il est donc important d'avoir une bonne culture scientifique générale pour travailler dans ce domaine [16].

- Exemple

Lorsque nous résolvons des systèmes d'équations différentielles, nous nous servons parfois de la Transformée de $Laplace$ dont le principe est de transformer une équation differentielle [1] linéaire à coefficients constants en un problème algèbrique, et nous travaillons donc sur des polynômes [8].

1. Une équation différentielle d'ordre n est une relation du type : $F(x, y, y', y^{(2)}...y^{(n)}) = 0$ où y est une fonction de x.

1.2.1 Exemples d'applications

Un premier champ d'action pour le calcul scientifique concerne les situations où nous ne pouvons pas ,du moins, pas complètement, réaliser une expérience ou une simulation telle que [16] :

- La simulation du fonctionnement des armes nucléaires en l'absence d'essais,
- La conception d'un centre de stockage définitif des déchets nucléaires.
- Le calcul de trajectoires des satellites et des fusées.
- En économie, l'étude du coût de production d'un produit dépend de la quantité de produits confectionnés, et la représentation graphique du coût de production (mais aussi du coût marginal) en fonction du nombre de produits réalisés est une parabole correspondant donc à un polynôme : plus nous en produisons, moins ça coûte mais cela n'est vrai que jusqu'à un certain point. Au delà de ce point (l'extremum de la parabole), l'augmentation de la production coïncide avec une augmentation des coûts de production (problèmes logistiques, effets d'échelle ...)
- En finance(trading), nous cherchons à anticiper des événements par le biais de la simulation et plus traditionnellement la prévision climatique ou météorologique

Dans toutes ces situations, une bonne simulation numérique permettera de donner quelques indications sur le comportement attendu de l'objet de la simulation – sans garantie totale que tout se passe comme prévu [16].

Il y a également des situations où il est moins coûteux de réaliser des tests numériques préliminaires : la simulation moléculaire des principes actifs pour l'industrie pharmaceutique, les tests de résistance mécanique (crash tests) dans l'industrie automobile, la synthèse de nouveaux matériaux pour l'industrie (alliages, polymères). Dans ces cas, la simulation numérique ne remplace pas une expérience, mais elle la complète, en suggérant des processus ou des comportements à tester spécifiquement de manière expérimentale [16].

1.2.2 Etapes du calcul scientifique

L'approche d'un problème par le biais du calcul scientifique est une démarche globale, qui se déroule en plusieurs temps successifs avec en pratique des aller-retours d'une étape à l'autre [16]. Les étapes du calcul scientifique sont les suivantes [16] :

a- **Modélisation du système** : Consiste à décrire les phénomènes observés, par des équations mathématiques, souvent en collaboration avec les scientifiques des disciplines applicatives concernées. Il est essentiel de connaître les hypothèses qui sous-tendent un modèle, leur domaine de validité et le comportement qualitatif que nous attendons des

solutions avant d'en chercher une approximation. De nombreux modèles des sciences de l'ingénieur sont posés sous forme d'une équation aux dérivées partielles (EDP) ou d'un système d'$EDPs$.

b- **Analyse théorique du modèle** : c'est l'étude des propriétés du modèle existence - unicité de la solution. Ceci peut faire intervenir des résultats profonds d'analyse, de théorie spectrale et/ou de théorie des probabilités, etc.

c- **Proposition d'une méthode numérique** : Qui consiste en la proposition d'une méthode numérique adaptée aux propriétés théoriques du modèle, et nous en faisons l'analyse numérique. Ceci permet de déterminer la vitesse de convergence de la méthode numérique et sa stabilité. Nous pouvons également chercher des estimations d'erreurs *à priori et à posteriori*.

d- **Implémentation informatique** : l'implémentation de la méthode avec éventuellement sa parallélisation sur un gros cluster de calcul, et sa validation sur des cas tests académiques pour vérifier le comportement de la méthode dans des situations bien connues.

Pour les mathématiciens, l'implémentation informatique représente l'étape finale de leur travail, en revanche c'est à se stade que commence la vraie aventure scientifique pour les chercheurs et les ingénieurs dans les domaines d'application qui consiste à l'utilisation de la nouvelle méthode sur des cas réels.

1.2.3 Calcul Haute Performance

Le calcul Informatique de Pointe (CIP) ou encore le Calcul Haute Performance où HPC pour :***H**igh **P**erformance **C**omputing* est un domaine interdisciplinaire, à l'intersection de la modélisation de processus et de l'utilisation des ordinateurs pour produire des résultats quantitatifs [1].

Autrefois réservé aux chercheurs et entreprises œuvrant dans des domaines spécifiques tels la dynamique des fluides, la physique ou la chimie computationnelle [2],le HPC est aujourd'hui un outil essentiel à la recherche et à l'innovation dans la plupart des secteurs de pointe de l'économie numérique du XXI^e *siècle*, que ce soit dans les domaines des sciences dites « *dures* » [3] et de l'ingénierie, les sciences de la santé ou encore les sciences humaines et sociales [1].

2. ou chimie informatique : est une branche de la chimie et/ou de la physico-chimie qui utilise les lois de la chimie théorique exploitées dans des programmes informatiques spécifiques afin de calculer structures et propriétés d'objets chimiques tels que les molécules...

3. Science naturelle et science formelle

1.3 Architectures parallèles

Un ordinateur parallèle est un ensemble d'éléments de traitement qui communiquent et coopèrent pour résoudre des problèmes de manière efficace.

1.3.1 Modèle Von NeuMann

Ce modèle fut inventé par le mathématicien hongrois $John\ Von\ Neumann$ qui a posé les premières bases de la conception d'un ordinateur dans son papier en 1945 . A partir de ce moment, la majorité des ordinateurs ont été conçus sur ces bases. L'architecture $Von\ Neumann$ (Fig. 1.1) est constituée de 4 composants principaux : une mémoire, une unité de contrôle, une unité arithmétique et logique (UAL) et des entrées/sorties (E/S). La mémoire à accès aléatoire (RAM) en lecture/écriture est utilisée pour stocker les instructions ainsi que les données. L'unité de contrôle va chercher les instructions ou les données de la mémoire, décode les instructions et coordonne séquentiellement les opérations afin d'accomplir la tâche programmée. L'UAL effectue les opérations arithmétiques de base. Les E/S font l'interface avec l'utilisateur humain [15].

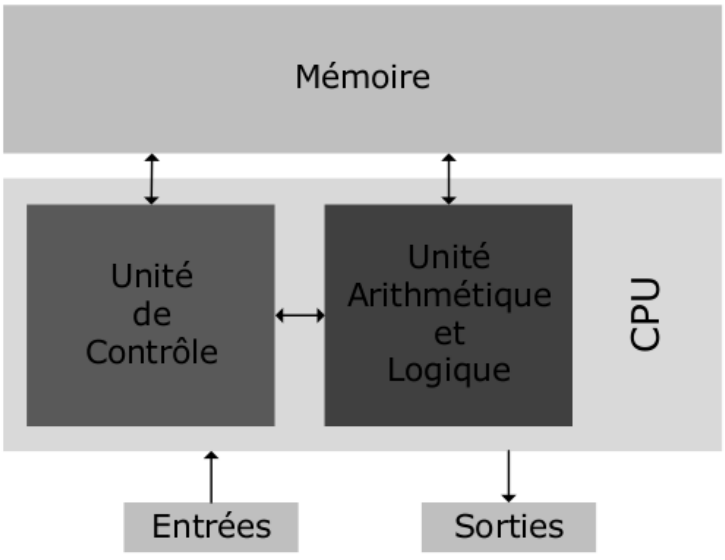

FIGURE 1.1 – Von NeuMann

1.3.2 Classification de Flynn

Une classification connue sous le nom de classification de *Flynn* caractérise les machines selon leurs flots de données et d'instructions qui sont exécutés simultanément.

Cette classification comprend quatre (04) catégories qui sont les suivantes :

- **Single Instruction, Single Data SISD** :
  Les premiers types d'ordinateurs correspondent aux ordinateurs *SISD : **S**ingle **I**nstruction **S**ingle **D**ata* appelé aussi machine de $Von\ Neumann$ ayant des processeurs purement séquentiels, incapables de toute forme de parallélisme. Ils exécutent une instruction sur un seul ensemble de données. La figure 1.2 illustre ce type d'ordinateur [5] :

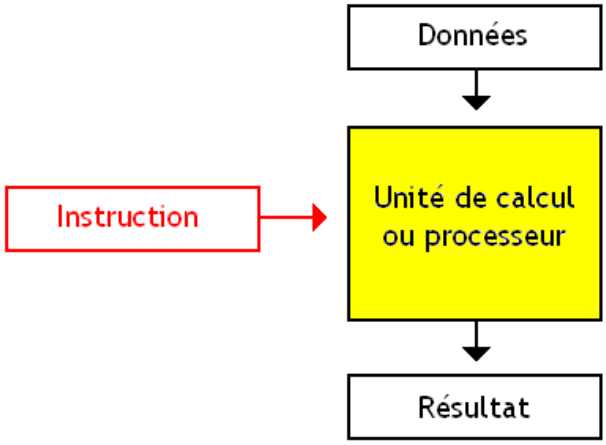

FIGURE 1.2 – Architecture SISD

- **Multiple Instruction, Single Data MISD** :
  Les ordinateurs de type *MISD : **M**ultiple **I**nstruction **S**ingle **D**ata* peuvent exécuter des instructions différentes en parallèle, sur une donnée identique comme l'illustre la figure 1.3 [5] :

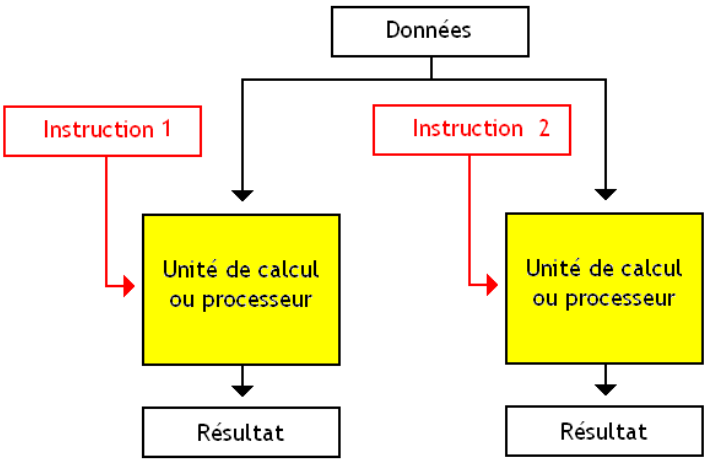

FIGURE 1.3 – Architecture MISD

- **Single Instruction, Multiple Data SIMD** :
  Les ordinateurs d'architectures *SIMD :**S**ingle **I**nstruction **M**ultiple **D**ata*, sont des ordinateurs permettant d'exploiter le parallélisme de données, ils peuvent exécuter les mêmes instructions sur des données différentes. La figure 1.4 montre ceci [5] :

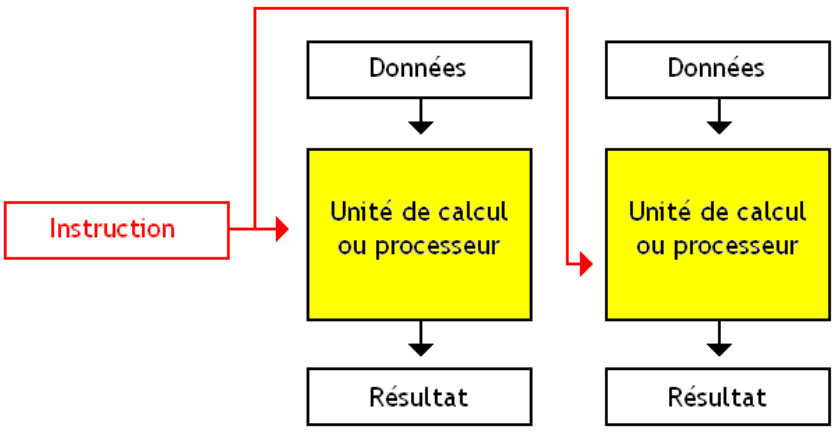

FIGURE 1.4 – Architecture SIMD

  Ce type d'architecture est adapté aux calculs matriciels et traitements d'images.
- **Multiple Instruction Multiple Data MIMD** :
  Les ordinateurs de la categorie *MIMD **M**ultiple **I**nstruction **M**ultiple **D**ata*, peuvent éxecuter des instructions différentes sur des données différentes [5].La figure 1.5 montre ceci :

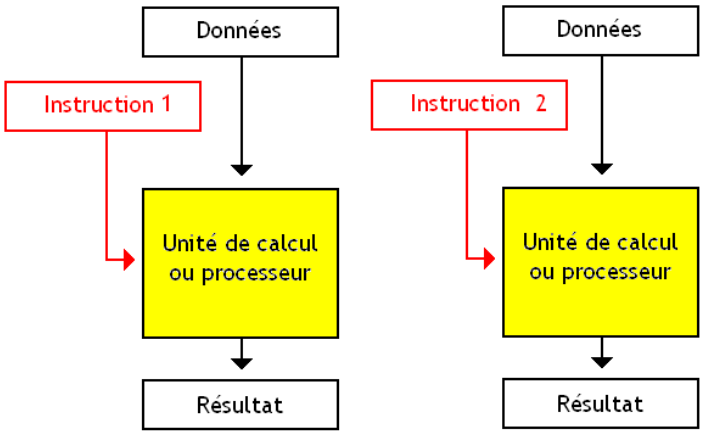

FIGURE 1.5 – Architecture MIMD

Cette catégorie peut être divisée en deux sous catégories *SPMD : **S**ingle **P**rogram **M**ultiple **D**ata* et *MPMD : **M**ultiple **P**rogram **M**ultiple **D**ata*.

- **SPDM** : consiste à exécuter un seul programme sur plusieurs données à la fois.
- **MPMD** : consiste à exécuter des programmes en parallèle sur des données différentes.

1.4 Parallélisation et modèle de programmation

1.4.1 Parallélisation

La parallélisation des codes de simulation numérique veut dire leur adaptation aux architectures des calculateurs parallèles. Celà s'avére être une nécessité impérieuse pour parvenir à faire du calcul à haute performance. Dans les codes de calcul scientifique, une part importante du temps d'exécution se passe dans la résolution de grands systèmes linéaires. Or cette partie des logiciels est généralement la plus difficile à paralléliser. La mise au point d'algorithmes parallèles efficaces pour résoudre les systèmes linéaires sur des calculateurs parallèles comportant un grand nombre de processeurs représente donc un enjeu fondamental pour le calcul scientifique [10].

1.4.2 Critères de performance

Les critères de performance sont utilisés dans le domaine du calcul parallèle, tels que le degré de parallélisme , l'équilibrage des tâches et la granularité [10].

- **Degré de parallélisme :**
  Le degré de parallélisme d'un programme est le nombre de tâches pouvant être exécutées simultanément [10].
  La façon classique de considérer la question d'un code de calcul scientifique consiste à évaluer la part qui est parallélisable et la part qui ne l'est pas. Nous parlons de la part de temps d'exécution parallèle T_p et de la part séquentielle T_s, sur un temps total T. tel que $T = T_s + T_p$. Ce qui conduit à une formule, appelée loi d'$Amdhal$, qui détermine le temps minimum d'exécution sur une machine parallèle comptant np processeurs, en supposant que la part parallélisable du code soit parfaitement répartie entre les différents processeurs sans que l'exécution en parallèle n'entraîne de surcroît [10] :

  $$T_{np} = T_s + \frac{T_p}{np}$$

  Le rapport entre le temps d'exécution sur un processeur et sur np processeurs, $\frac{T}{T_{np}}$ représente l'accélération due au parallélisme. Si α est la part du temps d'exécution parallélisable, $\frac{T_p}{T}$, l'accélération vaut, dans le cadre de la loi d'$Amdhal$ [10] :

  $$A_n = \frac{T}{T_{np}} = \frac{1}{(1-\alpha) + \frac{\alpha}{np}}$$

  L'accélération optimale sur np processeurs vaut évidemment np [10].
- **Efficacité :**
  Nous appelons l'efficacité le rapport entre l'accélération effective et l'accélération optimale, $E_{np} = \frac{A_{np}}{np}$. L'efficacité est un nombre compris entre 0 et 1.
  Dans le cadre de la loi d'$Amdhal$, l'efficacité est donnée par la formule [10] :

  $$E_{np} = \frac{1}{(1-\alpha)np + \alpha}$$

  .
- **Equilibrage des tâches :**
  Si une tâche requiert un temps d'exécution plus élevé que les autres, c'est elle qui va déterminer le temps minimal d'exécution en parallèle. Le second critère d'efficacité concerne donc la régularité en termes de temps d'exécution des différentes tâches. Nous parlons *d'équilibrage des tâches*. Dans le cas d'un petit nombre de processeurs, l'équilibrage peut être obtenu en divisant la charge de travail en un nombre de tâches nettement plus élevé. Chaque processeur se verra allouer un certain nombre de tâches à exécuter de façon à assurer l'équilibre global, une telle allocation est dite *statique* [10].

Dans certains cas, la durée de chaque tâche est difficile à connaître exactement à priori, par exemple : losque le type et la compléxité dépendent des valeurs des données. Dans ce cas, les différents processus parallèles vont exécuter les différentes tâches au fur et à mesure de leur disponibilité. Chaque processus qui vient de terminer une tâche prend en charge la première qui n'a pas été exécutée dans la liste globale des tâches. Ce mode de répartition *dynamique* des tâches est qualifié de **glouton** [10] .

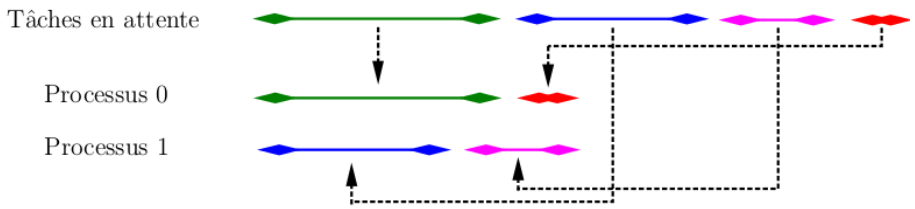

FIGURE 1.6 – Exécution parallèle en mode glouton

Pour assurer l'équilibrage, il serait plus efficace que le nombre de tâche soit grand et que ces tâches soient classées en ordre de taille décroissant : il est plus facile de combler les vides avec beaucoup de petits éléments qu'avec peu de gros éléments. Dans le cas oú le nombre de processus est élevé, il est généralement illusoire d'avoir un degré de parallélisme suffisant pour permettre ce mode de fonctionnement. Le découpage sera donc le plus souvent *statique*, avec une tâche par processeur [10].

- **Granularité :**
  La granularité d'une tâche parallèle est définie comme le rapport entre le nombre d'opération arithmétique que la tâche réalise et le nombre de données que la tâche doit recevoir ou envoyer.

1.4.3 Mécanismes de communication

Il existe deux modèles de communication inter-processeurs

Mécanisme Synchrone

Le processeur 1 emetteur attend que le processeur 2 récepteur ait signalé qu'il est prêt à recevoir les données émises (figure 1.7).

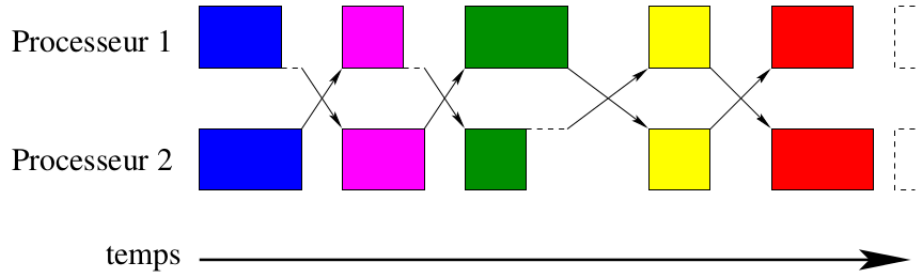

FIGURE 1.7 – modèle Synchrone

Mécanisme Asynchrone

Le processeur 1 n'attend pas de signal et transmet les données au processeur 2. ce dernier viendra les lire plus tard, quand le code lui dira de le faire (figure 1.8).

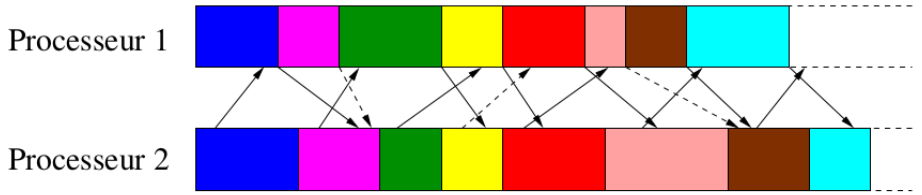

FIGURE 1.8 – modèle Asynchrone

1.5 Cartes graphiques Nvidia et Compute Unified Device Architecture

1.5.1 Les cartes graphiques

L'apparition des cartes graphiques *GPU* permettant de programmer les traitements déclencha de nombreuses recherches sur la possibilité d'utiliser les circuits graphiques pour autre chose qu'un affichage *OpenGL* ou $DirectX$. Ceci dit, l'approche générale de la programmation des premiers traitements *GPU* était extraordinairement compliquée. Les API graphiques standard comme *OpenGL* et $DirectX$ étant toujours le seul moyen d'interagir avec un *GPU*, toute tentative d'effectuer un traitement quelconque sur un *GPU* était soumise aux contraintes de la programmation avec une *API* graphique. Ceux qui souhaitaient effectuer des traitements généraux en passant par les *API* graphiques essayaient alors de faire croire au *GPU* que leur problème à résoudre était un problème d'affichage classique [3].

Essentiellement, les *GPUs* du début des années 2000 étaient conçus pour produire une couleur pour chaque pixel de l'écran à l'aide d'unités arithmétiques progrommables appelées "$pixel\ shaders$". En règle générale, un pixel shader utilise sa position (x, y) sur l'écran ainsi que certaines informations afin de produire la couleur finale à partir des données diverses. Ces informations supplémentaires peuvent, par exemple, être les couleurs d'entrée, les coordonnées de texture ou d'autres attributs passés au $shader$ lorsque ce dérnier s'exécute. L'arithmétique réalisée sur les couleurs d'entrée et les textures étant totalement contrôlée par le programmeur, certains ont remarqué que ces "couleurs" d'entrée pouvaient, en fait, représenter n'importe quelle information [3].

Si les entrées étaient des données numériques signifiant autre chose que des couleurs, les programmeurs pouvaient alors programmer les pixel $shaders$ pour qu'ils effectuent n'importe quel calcul sur ces données. Les résultats étaient restitués au *GPU* comme étant la "couleur" finale du pixel alors que celle-ci était, en fait, le résultat du calcul que le programmeur avait demandé au *GPU* d'effectuer sur ses entrées. En réalité, le principe consistait donc à tromper le *GPU* en lui faisant effectuer des traitements qui n'avaient rien à voir avec du graphisme. Cette astuce était rusée mais également très complexe [3].

Grâce au débit de sortie élevé des *GPU*, les résultats de ces expériences promettaient un futur intéressant aux traitements *GPU*. Cependant, le modèle de programmation était encore bien trop limité pour intéresser un nombre suffisant de développeurs. Les programmes ne pouvaient recevoir des données qu'à partir de quelques couleurs et d'unités de texture. En outre, de sérieuses contraintes pesaient sur la façon d'écrire en mémoire les résultats et sur les emplace-

ments où les écrire : les algorithmes qui devaient pouvoir écrire n'importe où en mémoire ne pouvaient donc pas s'exécuter sur un *GPU*. Il était également impossible de prévoir comment un *GPU* particulier traiterait les données en virgule flottante, pour autant qu'il puisse les gérer : la plupart des calculs scientifiques ne pouvaient donc pas se servir d'un *GPU*. Enfin, si le programme produisait un résultat incorrect, n'arrivait pas à se terminer ou simplement figeait la machine, il n'existait alors aucune méthode adéquate pour mettre au point le code exécuté sur le *GPU* [3].

Comme si toutes ces limitations ne suffisaient pas, toute personne souhaitant quand même utiliser un *GPU* pour des traitements généraux devait apprendre *OpenGL* ou DirectX, ceux-ci étant les seuls moyens d'interagir avec un *GPU*. Ceci signifiait donc non seulement qu'il fallait appeler des fonctions *OpenGL* ou DirectX pour stocker les données dans des textures graphiques et pour exécuter les calculs, mais également que les traitements devaient être exprimés dans des langages de programmation graphiques spéciaux, appelés langages de shading. Gérer ces restrictions sur les ressources et la programmation et devoir apprendre des langages de shading et le graphisme sur ordinateur afin de pouvoir profiter de la puissance d'un *GPU* étaient bien trop compliqués pour séduire la grande majorité des programmeurs [3].

1.5.2 CUDA

- **Historique :**
  Ce n'est que cinq ans après la sortie des GeForce3 [4] que le traitement sur *GPU* allait devenir accessible à tous. En novembre 2006, *NVIDIA* révéla le premier *GPU* DirectX10, la *GeForce* 8800 GTX, qui fut également le premier *GPU* reposant sur l'architecture *CUDA* de *NVIDIA*. Cette architecture a en effet, introduit plusieurs nouveaux composants conçus spécialement pour les traitements *GPU* généraux et gommer ainsi la plupart des restrictions induites par les *GPU* précédents [3].
- **Architecture CUDA :**
  À la différence des générations précédentes, qui partitionnaient les ressources de traitement en vertex et en pixel shaders, l'architecture *CUDA* comprend un traitement unifié permettant ainsi à chaque unité arithmétique et logique (ALU) du circuit d'être gérée par un programme afin de réaliser des traitements généraux. *NVIDIA* ayant souhaité que cette nouvelle famille de processeurs graphiques soit utilisée pour des traitements généraux, ces ALU implémentent la norme *IEEE* pour l'arithmétique virgule flottante simple précision et ont été conçues pour utiliser un jeu d'instructions adapté à ces traitements généraux au lieu d'être spécialisées dans les traitements graphiques. En outre, les unités d'exécution du *GPU* peuvent désormais lire et écrire n'importe où en mémoire et également accéder à un cache (géré de façon logicielle) appelé "mémoire partagée". Toutes

4. Famille de processeurs graphiques

ces fonctionnalités de l'architecture *CUDA* ont été ajoutées afin de créer un *GPU* qui excelle dans les traitements généraux, avec de bonnes performances pour les traitements graphiques classiques [3].

1.6 Conclusion

Dans ce chapitre il a été question de définir le calcul scientifique qui est par essence un domaine interdisciplinaire tant au sein des sciences en général qu'au sein des mathèmatiques.

Le calcul scientifique permet des études numériques dans les domaines oú nous ne pouvons pas , du moins complètement, réaliser des expériences. Pour arriver à ces études numériques, nous devons passer par des étapes telles que la modélisation du système,son analyse théorique, la proposition d'une méthode numérique et enfin l'implémentation informatique de cette méthode. Pour celà, la machine doit avoir une architecture spécifique, cette dernière permet de réaliser la parallélisation et le modèle de programmation adéquat à la situation. Nous avons aussi parlé des critères de parallélisation tels que le degré , l'efficacité et l'équilibrage des tâches, et que pour accélérer des applications professionnelles de science, d'ingénierie et d'entreprise, le calcul GPU qui consiste à utiliser le processeur graphique GPU en paralléle du CPU a été mis en oeuvre et programmable avec $CUDA$.

Après documentation et études , nous avons choisi d'implémenter deux méthodes d'extraction de racine de polynôme dont l'objectif est d'arriver à des résultats correcte en des temps d'exécution raisonnable. L'étude théorique de ces méthodes est présentée dans le prochain chapitre.

Chapitre 2

Méthodes Durand-Kerner et Eurlish-Aberth

2.1 Introduction

Contrairement à ce qui est dans le cas des systèmes d'équations linéaires, la résolution des systèmes non-linéaires s'avère beaucoup plus délicate. En général, il n'est pas possible de garantir la convergence vers la solution correcte, et la complexité des calculs croît très vite en fonction de la dimension du système. La méthode du point fixe que nous detaillerons dans ce chapitre avec la méthode de $Newton$ et ses dérivées, les méthodes de $Durand\text{-}kerner$ et $Ehrlish\text{-}Aberth$. L'ensemble de ces méthodes représente les techniques les plus souvent utilisées pour la résolution des systèmes d'équations non-linéaires : calcul des racines qui concerne plus particulièrement les polynômes creux qui sont des polynômes identifiés par peu de monômes non nuls (moins de vingt (20)).

2.2 Differentes sources d'erreur dans une méthode numérique

Les solutions des problèmes calculées par une méthode numérique sont affectées par des erreurs que nous pouvons principalement classer en trois catégories [11] :

- Les *erreurs d'arrondi* : dans les opérations arithmétiques, proviennent des erreurs de représentation dues au fait que tout calculateur travaille en *précision finie*, c'est à dire dans un sous-ensemble discret du corps des réels $\Re$, l'arithmétique naturelle étant alors approchée par une arithmétique de *nombres à virgule flottante*.

- Les *erreurs sur les données* : imputables à une connaissance imparfaite des données du problème que nous cherchons à résoudre, comme lorsqu'elles sont issues de mesures physiques soumises à des contraintes expérimentales.
- Les *erreurs de troncature, d'approximation ou de discrétisation* : introduites par les *schémas de résolution numérique* utilisés, comme le fait de tronquer le développement en série infini d'une solution analytique pour permettre son évaluation , d'arrêter un processus itératif dés qu'une itération satisfait un critère donné avec tolérance préscrite, ou encore d'approcher la solution d'une équation aux dérivées partielles en un nombre fini de points .

Nous pouvons également envisager d'ajouter à cette liste les erreurs qualifiées d'*humaines*, telles que les erreurs de programmation, ou celles causées par des dysfonctionnements des machines réalisant le calcul [11].

L'étude des erreurs de troncature , d'approximation ou de discrétisation constitue pour sa part un sujet majeur traité par l'analyse numérique.

2.3 Méthode du point fixe

Soit une fonction $f(x) = 0$, nous isolons un terme contenant x de sorte à pouvoir écrire [11] :

$$x_{nouveau} = g(x_{prcedent})$$

Nous appelons la fonction $g(x)$ la fonction d'itération et une itération du point fixe est définie par l'algorithme suivant [11] :

Algorithm 1: Point fixe

Entrée
Initialiser $x^{(0)}$;
pour $k = 1, 2, ...jusqu'aconvergence$ **faire**
  $x^{(k)} = g(x^{(k-1)})$;
fin pour

Remarque [11] :

- La notion de convergence doit être approfondie.
- Les valeurs successives de x ne doivent pas être conservées, il suffit de conserver deux valeurs successives $x^{(k)}$ et $x^{(k+1)}$ dans des variable $x0$ et $x1$ et de remplacer $x0$ par $x1$ à chaque itération.
- Si la solution x_s vérifie $x_s = g(x_s)$ fait que nous appellons x_s un *point fixe*.
- Le choix de la fonction $g(x)$ est déterminant pour la convergence de la méthode.

2.3.1 Précision de la machine

La precision de la machine d'une arithmetique en virgule flottante est definie comme le plus petit nombre positif eps tel que [11] :

$$float(1 + eps) > 1$$

2.3.2 Convergence de la méthode du point fixe

Les itérations du point fixe convergent pour $x \in [a, b]$ si l'intervalle contient un zéro et si $|g'(x)| < 1$ pour $x \in [a, b]$ [11] .
Si $-1 < g'(x) < 0$ les itérations oscillent autour de la solution et si $0 < g'(x) < 1$ les itérations convergent de façon monotone [11].

2.4 Methode de Newton

La méthode de $Newton$ est basée sur le développement de $Taylor$ [1] . Soit x^* une solution de l'équation non linéaire $f(x) = 0$ [11]. Nous savons d'après la formule des rectangles à gauche que [11] :

$$f(x^*) - f(x_k) = \int_{x_k}^{x^*} f'(t)\,\mathrm{d}t = (x^* - x_k)f'(x_k) + \frac{(x^* - x_k)^2}{2} f''(\eta k).$$

En notant que $f(x^*) = 0$ et en négligeant l'erreur de quadrature numérique $\frac{(x^*-x_k)^2}{2} f''(\eta k)$ nous obtenons la méthode de $Newton$:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

2.5 Methodes itératives Gauss Seidel et Jacobi

Ces méthodes remontent à Gauss Liouville (1837) et Jacobi (1845) .
La méthode itérative contrairement à d'autres a l'avantage de ne pas avoir besoin de garder en

1. une fonction plusieurs fois dérivable au voisinage d'un point peut être approximée par une fonction polynôme dont les coefficients dépendent uniquement des dérivées de la fonction en ce point.

mémoire la totalité d'une matrice de très grande taille gourmande en capacité mémoire [11]. Cette méthode permet de garder en mémoire que les coefficients non nuls d'une matrice de grande taille [11].

2.5.1 Gauss Seidel

La méthode de Gauss-Seidel est une méthode itérative de résolution d'un système linéaire de dimension finie de la forme $Ax = b$, ce qui signifie qu'elle génère une suite qui converge vers une solution de cette équation, lorsque celle-ci en a une et lorsque des conditions de convergence sont satisfaites. Le principe de la méthode peut s'étendre à la résolution de systèmes d'équations non linéaires et à l'optimisation, mais avec des conditions d'efficacité moins claires. En optimisation, l'utilité de cette approche dépendra beaucoup de la structure du problème. L'algorithme est le suivant [11] :

Algorithm 2: Gauss Seidel

Entrée
Initialiser $x^{(0)}$;
pour $k = 1, 2, ...jusqu'aconvergence$ **faire**
  pour $i = 1...jusqu'an$ **faire**
    $x_i^{(k+1)} = b_i - \sum_{j \neq i} a_{ij} x_j^{(k+1)} / a_{ii}$;
  fin pour
fin pour

2.5.2 Jacobi

La méthode de $Jacobi$ est une méthode itérative de résolution d'un système matriciel de la forme $Ax = b$. Pour cela, nous utilisons une suite $x^{(k)}$ qui converge vers un point fixe x. L'itération de $Jacobi$ est particulière dans le sens ou nous n'utilisons pas les résultats les plus récents et nous procèdons à n résolutions unidimensionnelles indépendantes. Ainsi lorsque nous calculons, par exemple, x_2^{k+1} nous utilisons x_1^k et non x_1^{k+1} qui est déjà connu. Ceci définit l' *itération de Jacobi* qui est formalisée dans l'algorithme suivant [11] :

Si nous modifions l'itération de $Jacobi$ de sorte à tenir compte des résultats les plus récents nous obtenons l'itération de $Gauss\text{-}Seidel$ [11] :

$$x_i^{(k+1)} = b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} / a_{ii}$$

Algorithm 3: Jacobi

Entrée
Initialiser $x^{(0)}$;
pour $k = 1, 2, ...jusqu'aconvergence$ **faire**
  pour $i = 1...jusqu'an$ **faire**
    $x^{(k+1)} = b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} / a_{ii}$;
  fin pour
fin pour

2.6 Methode DURAND-KERNER

2.6.1 Historique

En analyse numérique la méthode Durand-Kerner établie entre 1960 et 1966 et nommée d'après $E.Durand$ et $Immo\ Kerner$ aussi appelée par abus de langage la méthode de $Weierstrass$ (établie entre 1859 et 1891 par $Karl\ Weierstrass$) est une méthode d'extraction approximative des racines d'un polynôme basée sur la méthode de $Newton$.

2.6.2 Principe

Nous voulons trouver les racines $w_1, ..., w_n$ pour $n - 1$degré $P(z) = \sum_{i=0}^{n} a_i z^{n-i}$ avec $a_0 = 1$, $a_n \neq 0$ avec a_i et z dans C

La méthode Durand-Kerner est une méthode de résolution simultanée, qui permet l'extraction de toutes les racines à la fois. Cette méthode se compose de quatre phases principales [13] :

Phase 1 : Initialisation du polynôme

L'initialisation du polynôme à coéfficients complexes $P(z)$ peut s'effectuer ainsi :

$$\forall a_i \in C; P(z) = \sum_{i=0}^{n} a_i.z^{n-i} avec(a_0 = 1, a_n \neq 0)$$

Phase 2 : Initialisation du vecteur $Z^{(0)}$

La deuxième phase de la méthode consiste en l'initialisation du vecteur $Z^{(0)}$ constituant un ensemble de points. Cette initialisation du vecteur est importante, car les éléments du vecteur doivent être distincts les uns des autres. C'est la méthode de $Guggenheimer$ qui est employée. Un rayon σ_0 est déterminé à partir des coefficients du polynôme et les racines sont placées à équidistance sur le cercle. Le calcul de σ_0 se réalise ainsi :

$$\sigma_0 = \frac{u+v}{2}$$

$$avec, u = \frac{\sum_{i=1}^{n} u_i}{n.\max_{i=1} u_i} \; v = \frac{\sum_{i=0}^{n-1} v_i}{n.\min_{i=0} v_i}$$

$$et, u_i = 2.|a_i|^{\frac{1}{i}} \; v_i = \tfrac{1}{2}.|\tfrac{a_n}{a_i}|^{\frac{1}{n-i}}$$

Phase 3 : Fonction itérative H_i

La troisième phase de la méthode est le calcul de la fonction itérative H_i qui va permettre de converger vers les racines solutions du polynôme, à conditions que toutes les racines soient distinctes :

$$\forall i \in [1,n]; H_i(Z) = z_i - \frac{P(z_i)}{\prod_{\substack{j=1 \\ j\neq i}}^{j=n} (z_i - z_j)}$$

Phase 4 : Conditions de convergence

La dernière phase de la méthode est la condition d'arrêt de l'itération. C'est elle qui détermine le succès de la convergence.

- La première solution consiste à stopper la fonction itérative lorsque l'ensemble des modules du polynôme pour toutes les racines est inferieur à une valeur fixée

$$\forall i \in [1,n]; |P(z_i)| < eps$$

- Dans la deuxième solution, nous arrêtons la fonction itérative lorsque les racines sont stables, c'est-à-dire que la méthode converge suffisamment :

$$\forall i \in [1, n]; |\frac{z_i^{(k)} - z_i^{(k-1)}}{z_i^{(k)}}| < eps$$

Utilisation du logarithme

Dans certains cas, nous évaluons le logarithme d'un polynôme au lieu du polynôme lui même, ce qui nous permet de manipuler indirectement des nombres de grande valeur absolue. [13].

2.6.3 Vitesse de Convergence

La vitesse de convergence de la méthode de $Durand\text{-}Kerner$ est quadratique [7]. Dans la pratique, nous espèrons rarement obtenir une convergence plus que quadratique, elle même déjà très satisfaisante.

2.7 Methode d'Aberth-Ehrlich

2.7.1 Historique

La méthode $Aberth\text{-}Ehrlich$, ou plus communément appelée méthode d'$Aberth$, nommée d'après $Oliver\ Aberth$ et $Louis\ W.Ehrlich$ est comme la méthode de $Durand\text{-}Kerner$, une méthode d'extraction de racines à partir d'un polynôme.

2.7.2 Principe

La méthode d'$Aberth$ suit les même phases que la méthode de $Durand\text{-}Kerner$ excépté la phase 3 (Fonction itérative). La principale différence est que l'algorithme de Durand-Kerner se contente d'évaluer le polynôme donné $P(z)$ en les n approximations des racines alors que l'algorithme d'$Aberth\text{-}Ehrlich$ évalue aussi sa dérivée $P'(z)$ en ces mêmes approximations.

Les principales exigences lors de l'utilisation de la méthode d'$Ehrlich\text{-}aberth$ pour calculer les racines de $P(z)$ sont un calcul rapide, robuste et stable de la correction de Newton $P(z)/P'(z)$ [7]. La phase 3 de la méthode de $Ehrlich\text{-}aberth$ est la suivante :

Phase 3 :Fonction itérative H_i

La fonction itérative H_i va permettre de converger vers les racines solutions du polynôme [7] :

$$\forall i \in [1,n]; H_i(Z) = z_i - \frac{\frac{P(z_i)}{P'(z_i)}}{1 - \frac{P(z_i)}{P'(z_i)} \sum_{\substack{j=1 \\ j \neq i}}^{j=n} \frac{1}{(z_i - z_j)}}$$

2.7.3 Vitesse de Convergence

Comme cela a été dit précedemment, dans la pratique, nous nous attendons rarement à obtenir une convergence plus que quadratique. Pourtant, la vitesse de convergence de la méthode d'$Aberth$ est super-linéaire (cubique ou encore plus élevée si la mise en œuvre est dans le style de $Gauss\text{-}Seidel$) [7].

2.8 Conlusion

Dans ce chapitre nous avons expliqué en détails les méthodes de calcul de racines $Durand\text{-}Kerner$ et $Aberth\text{-}Ehrlich$, qui sont dérivées de la méthode de $Newton$. En conclusion la convergence de la méthode de $Durand\text{-}Kerner$ est quadratique, théoriquement prouvée, quant à celle d'$Ehrlish\text{-}Aberth$, elle est cubique, ce que nous vérifirons dans l'implémentation dans le prochain chapitre, oú nous expliquerons les étapes de notre implantation en langage C.

Chapitre 3

Implantations séquentielles et parallèles

3.1 Introduction

Dans ce chapitre, nous décrivons tout d'abord les environnements de programmation et les outils utilisés pour intérpréter nos résultats, puis la mise en oeuvre de notre implantation, ensuite nous illustrons et interprétons les différents résultats de l'exécution des algorithmes $Durand$-$Kerner$ et $Ehrlish$-$Aberth$ sur différents degrés de polynômes. Enfin, nous effectuons une comparaison entre les graphes résultants des différents tests effectués.

3.2 Environnement de programmation et outils de développement

Pour mener à bien notre travail, nous avons utilisé un langage de programmation de bas niveau , pour pouvoir faire du calcul intensif, et nous avons fait appel à une panoplie d'outils tels que *GnuPlot* pour la génération des graphes de comparaison et *Valgrind* pour faire le profilage de nos programmes.

3.2.1 Langage *C*

Le C est un langage de programmation impératif, généraliste, issu de la programmation système. Inventé au début des années 1970,par *Dennis Ritchie* et *Bell Labs*, pour réécrire UNIX.

C est qualifié de langage de bas niveau dans le sens où chaque instruction du langage est conçue pour être compilée en un nombre d'instructions machine assez prévisible en termes d'occupation mémoire et de charge de calcul. Il propose un éventail de types entiers doubles et flottants conçus pour pouvoir correspondre directement aux types supportés par le processeur [14]. C'est devenu un des langages les plus utilisés. De nombreux langages plus modernes comme : *awk, csh, C++, C#, Objective C, BitC, D, Concurrent C, Java, JavaScript, Perl* et *PHP* reprennent des aspects de *C* [14] .

3.2.2 *OpenMP*

OpenMP (Open Multi-Processing) est une bibliothèque supportée par plusieurs langages (C,C++ et Fortran) disponible sur plusieurs plate-formes (Linux, Windows, OS X, ...). OpenMP regroupe des directives de compilation et des fonctions. Le compilateur *gcc* supporte *OpenMP* depuis la version 4.2, en ajoutant simplement une option sur la ligne de commande et en incluant le fichier d'en-têtes omp.h. La gestion des différentes fonctions est assurée par la librairie libgomp. [4][12].

3.2.3 Architecture CUDA

Pour intéresser le maximum de développeurs possibles, *NVIDIA* a choisi le langage C – une norme incontournable de l'industrie informatique – et lui a ajouté un nombre relativement restreint de mots-clés afin d'exploiter certaines des fonctionnalités spéciales de l'architecture *CUDA*. Quelques mois après le lancement de la GeForce 8800 GTX, *NVIDIA* mit à disposition un compilateur pour ce langage, *CUDA C* , qui devint le premier langage conçu spécifiquement par un constructeur de *GPU* pour faciliter le développement de traitements généraux sur ses circuits graphiques [3].

Outre la création de ce langage pour écrire un code, *NVIDIA* fournit également un pilote matériel permettant d'exploiter l'énorme puissance de calcul de l'architecture *CUDA*. Désormais, les utilisateurs n'ont donc plus besoin d'apprendre les *API* graphiques *OpenGL* ou $DirectX$ ni de transformer leurs problèmes en traitements graphiques [3].

3.2.4 Langage *CUDA C*

Compute Unified Device Architecture (*CUDA*) est une technologie de *General-Purpose Computing on Graphics Processing Units*(*GPU*) , c'est-à-dire qu'un processeur graphique *(GPU)* est utilisé pour exécuter des calculs généraux habituellement exécutés par le *processeur central* (*CPU*). *CUDA* permet de programmer des GPU en C. Cette technologie a été développée par $NVIDIA$ pour leurs cartes graphiques *GeForce 8 Series*, et utilise un pilote unifié utilisant une technique de streaming (flux continu). Le premier kit de développement pour *CUDA* a été publié en février 2007[3].

3.2.5 Gnuplot

Gnuplot est un programme souple qui peut produire des représentations graphiques en deux ou trois dimensions de fonctions numériques ou de données. Le programme fonctionne sur tous les ordinateurs et systèmes d'exploitation principaux et peut envoyer les graphiques à l'écran ou dans des fichiers dans de nombreux formats. Nous avons utilisé *Gnuplot* pour évaluer graphiquement les résultats des algorithmes que nous avons implémenté [2].

3.2.6 Valgrind

Valgrind est un système pour le débogage et le profilage de programmes *Linux*. Avec une série d'outils, permettant de détecter automatiquement de nombreux bogues de gestion de la mémoire ou de fils d'exécution, en s'évitant des heures à chasser les bogues et en rendant les programmes plus stables. *Valgrind* effectue également du profilage détaillé pour aider à accélérer les programmes. L'outil *callgrind*,anciennement nommé *CallTree*, et *KCachegrind* étant l'interface graphique permettant de visualiser les *callgraphs*, Tout exécutable peut être profilé à l'aide de *callgrind* sans recompilation,y compris les applications multi-threadées, les bibliothèques partagées,et les plugins. Nous l'avons utilisé pour identifier les portions de code à optimiser et à paralléliser [6].

3.2.7 Caractéristiques de la machine utilisée

Les caractéristiques de la machine sur laquelle nous avons mis en œuvre nos algorithmes sont les suivantes :

- Type de processeur Intel $Core^{TM}$ *i7 8 cœurs*.
- Version Intel $Core^{TM}$ i7 - 4700MQ CPU 2.4GHz , 8.00 Go de RAM.
- Type d'architecture :$64bits$.
- Nom de la carte mère *TOSHIBA*.
- Carte graphique NVIDIA GeForce GT 740M , 2048 Mbytes.

3.2.8 Caractéristiques de la carte *NVIDIA*

Les caractéristiques de la carte graphique utilisée , résultat de l'execution du programme $deviceQuery.cu$, fournit par *NVIDIA* sont dans l'annexe A.1 :

3.3 Implémentations

Dans le cadre de ce travail, nous avons implémenté deux méthodes de calcul de racines de polynômes, $Durand\text{-}Kerner$ et $Ehrlish\text{-}Aberth$. Celà consiste en l'implémentation des quatre (04) phases dans chaque méthode. Pour ce faire nous avons défini des fonctions correspondantes à chacune des phases.

3.3.1 Méthode Durand-Kerner

Nous passons à la présentation des fonctions correspondantes aux phases du calcul des racines de $Durand\text{-}Kerner$.

Avec le langage C dans lequel les nombres complexes n'existent pas , pour faire notre calcul, nous avons définis une structure $complexe$, qui est composée de deux(02) $double$(une partie imaginaire et une partie réelle). Nous avons aussi défini les fonctions arithmétiques de base pour effectuer les calculs entre nombres complexes comme détaillées dans l'annexe A.2. Ceci nous a permis de manipuler les nombres complexes et ainsi arriver à de bons résultats. Nous avons également défini les fonctions $Geugneimer()$ et $initialiser()$, la première pour l'initialisation de Gegneimer et la seconde pour l'initialisation du Polynôme. Ces fonctions sont

des applications directes des deux (02) premières phases de la méthode.

La fonction $FirstH()$, correspond à la phase trois (03) de la méthode. Dans celle-ci, le calcul d'une racine à une itération donnée se fait en faisant appel à la fonction $Fonction()$ oú $P(Z)$ est calculé.

La fonction nommée $H()$ appelle la fonction $FirstH()$ si la racine n'est pas d'une très grande valeur absolue, sinon $H()$ fait appel à la fonction $NewH()$ qui correspond à l'utilisation du logarithme qui permet de manipuler des nombres de grande valeur comme expliqué dans le chapitre précedent.

Nous avons aussi implémenté la condition de convergence appelée $ArretEcart()$ qui implémente la phase 4 et qui est vérifiée à chaque itération, cette fonction renvoie le resultat de $|\frac{z_i^{(k)} - z_i^{(k-1)}}{z_i^{(k)}}|$ que nous comparons avec eps.

Les algorithmes suivants correspondent aux fonctions $Fonction()$,$FirstH()$, $NewH()$ et enfin $H()$.

Algorithm 4: *Fonction()*

Entrées: $Complexe\ z, Polynome\ P$
Sorties: Cp
$Entier\ i, n$;
$Complexe\ Cp, Z_i$;
pour *i=0 ;i<degrePolynome ;i++* **faire**
  $Cp = P(z_i)$;
Fin Pour
return Cp ;

Algorithm 5: *FirstH()*

Entrées: $int\ i, Complexe * Z, Polynome\ P$
Sorties: DK
$Entier\ j$;
$Complexe\ A, Dk$;
$Dk = Fonction(Z[i], P)$; $A.x = 1$; $A.y = 0$; //initialiser A à 1 A étant le produit de $Durand\text{-}Kerner$
pour *j=0 ;j<degrePolynome ;j++* **faire**
  si $(i! = j)$ **alors**
    $A = A * (Z[i] - Z[j])$//Le produit de Durand Kerner $\prod_{\substack{j=1 \\ j \neq i}}^{j=n} (z_i - z_j)$
  Fin Si
Fin Pour $Dk = Dk / A$
$Dk = Z[i] - Dk$ // cette partie correspond au calcul de : $z_i - \frac{P(z_i)}{\prod_{\substack{j=1 \\ j \neq i}}^{j=n} (z_i - z_j)}$
return Dk ;

Algorithm 6: *NewH()*

Entrées: $int\ i, Complexe\ *Z, Polynome\ P$
Sorties: Dk_l
$Entier\ j$;
$Complexe\ A$,Dk_l;
$Dk_l == Fonction(Z[i], P)$; //initialiser A à 1 A étant le produit de $Durand\text{-}Kerner$
pour *j=0 ;j<degrePolynome ;j++* **faire**
  si $(i! = j)$ **alors**
    $Dk_l = Dk_l - ln(Z[i] - Z[j])$//Le produit de Durand Kerner avec le logarithme nipérien (ln) pour manipuler des valeurs pas très grande
  Fin Si
Fin Pour
$Dk_l = Z[i] - exp(Dk_l)$ // cette partie correspond au calcul de l'expenentiel pour réajuster la valeur dans le cas ou elle est trop grande
return Dk_l ;

Algorithm 7: *H()*

Entrées: $int\ i, Complexe\ *Z, int\ rayon$
Sorties: $Complexe\ Z_i$
//le rayon est une valeur fixée qui détérmine la valeur à partir de la quelle nous devons utilisé le Logarithme.
si $|P(z_i)| < rayon$ **alors**
  $FirstH(i, Z, P)$
sinon
  $NewH(i, Z, P)$
return $Z[i]$;

Le programme principal $main()$ fait appel à la fonction H et à la fonction $ArretEcart$, pour l'extraction des racines du polynôme.

Algorithm 8: *main()*

$Entier\ i, iteration$;
$Complexe\ ZPrec$,Z;
$initialiser(P)$;
$Gegneihmer()$
pour *i=0 ;i<degrePolynome ;i++* **faire**
  $Z[i] = H(i, ZPrec)$; //ZPrec represente le vecteur racine de l'itération précédente
  //Nous gardons les premières valeur de Z dans ZPrec
Fin Pour
répéter
  pour *i=0 ;i<degrePolynome ;i++* **faire**
    $ZPrec[i] = Z[i]$;
    $Z[i] = H(i, Z)$;
  Fin Pour
  $iteration++$ //pour calculer le nombre d'itéartions
jusqu'à $ArretEcart >= eps$;
$Resultat()$; //Affichage des résultats

La figure suivante a été générée grâce à Callgrind pour un polynôme de degré 1000 pour la méthode $Durand\text{-}Kerner$, elle montre les fonctions définies précedemment , le nombre d'appel à ces fonctions et le pourcentage du temps d'execution par rapport au temps total.

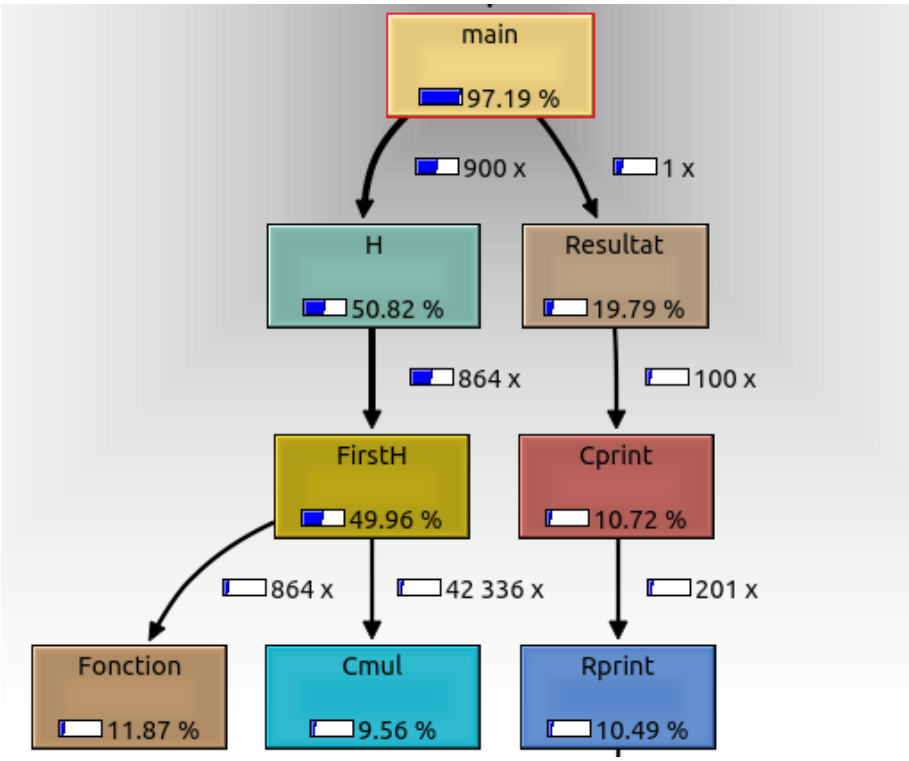

FIGURE 3.1 – Résultat de profilage de *Durand-Kerner*

Dans la figure 3.1, nous remarquons les fonctions $Resultat()$,$Cprint()$ et $Rprint()$ qui sont respéctivement les fonctions d'affichage des résultats et l'affichage des complexes que nous avons définis.

3.3.2 Méthode Ehrlish-Aberth

Nous passons à la présentation des fonctions correspondantes aux phases du calcul des racines de $Ehrlish\text{-}Aberth$.

L'implémentation de cette méthode consiste à garder pratiquement les mêmes fonctions que celles de $Durand\text{-}Kerner$, de ne changer que la phase 3 de la méthode, c'est à dire introduire la notion de dérivée , calculer la derivée du polynôme en une valeur de racine donnée, une fonction que nous appelerons $FonctionD()$ qui a comme paramètre une racine z, par la suite nous changeons $FirstH$ de sorte à faire le calcul des racines en appliquant $H_i(Z) = z_i - \dfrac{\frac{P(z_i)}{P'(z_i)}}{1-\frac{P(z_i)}{P'(z_i)}\sum\limits_{\substack{j=1 \\ j\neq i}}^{j=n}\dfrac{1}{(z_i - z_j)}}$ c'est à dire en introduisant la correction de $Newton$. L'algorithme suivant correspond à l'implémentation de $FirstH$ de la méthode d'$Ehrlish\text{-}Aberth$.

Algorithm 9: *FirstH*

Entrées: $int\ i, Complexe * Z, Polynome\ P$

Sorties: EA

$Entier\ j;$

$Complexe\ sum, A, E;$

$E = Fonction(Z[i], P);\ EA = FonctionD(Z[i]);$ //$P'(Z)$

$EA = E/EA;$ //$\frac{P(Z)}{P'(Z)}$;//correction de Newton

$sum.x = 0;\ sum.y = 0;$//somme d'ehrlish initialisé à 0

$A.x = 1;\ A.y = 0;$ //initialiser A à 1

pour *j=0 ;j<degrePolynome ;j++* **faire**

  si $(i! = j)$ **alors**

    $sum = sum + A/(Z[i] - Z[j])$//La somme de Ehrlish Aberth $\sum_{\substack{j=1 \\ j \neq i}}^{j=n} \frac{1}{(z_i - z_j)}$

  $FinSi$

Fin Pour

$EA = EA/(A - (EA * sum))$

$EA = Z[i] - EA$ // cette partie correspond au calcul de : $z_i - \frac{\frac{P(z_i)}{P'(z_i)}}{1 - \frac{P(z_i)}{P'(z_i)} \sum_{\substack{j=1 \\ j \neq i}}^{j=n} \frac{1}{(z_i - z_j)}}$

return EA ;

3.3.3 Paralélisation avec OpenMP

Dans le cadre de la parallélisation CPU avec OpenMP, nous avons parallélisé la portion suivante, nous verrons par la suite pourquoi notre choix s'est porté sur cette partie du code : Notons que #pragma omp parallel **pour** shared(Z,Zprec) private(i) est le mode de parallélisation avec OpenMP.

Algorithm 10: *main()*

```
Entier i, iteration;
Complexe ZPrec,Z;
initialiser(P);
Gegneihmer()
pour i=0 ;i<degrePolynome ;i++ faire
    Z[i] = H(i, ZPrec); //ZPrec represente le vecteur racine de l'itération précédente
    //Nous gardons les premières valeur de Z dans ZPrec
Fin Pour
répéter
    #pragma omp parallel pour shared(Z,Zprec) private(i)
    pour i=0 ;i<degrePolynome ;i++ faire
        ZPrec[i] = Z[i];
        Z[i] = H(i, Z);
    Fin Pour
    iteration + + //pour calculer le nombre d'itéartions
jusqu'à ArretEcart >= eps;
Resultat(); //Affichage des résultats
```

3.3.4 Condition des études

Les expériences que nous avons effectuées ont été faites sur la machine décrite précédemment, tout en diminuant au minimum l'utilisation de son CPU par d'autre programme. Dans les graphes qui suivent, pour chaque degré de polynôme, nous avons calculé la moyenne du temps d'exécution et le nombre d'itérations pour l'extraction de racines d'un même polynôme après dix (10) tests. Les résultats de nos expériences sur les deux méthodes d'extraction de racine de polynôme,la première méthode $Durand - Kerner$, et la seconde $Ehrlish - Aberth$. Cette dernière ne fut pas implémentée auparavant .

Pour réaliser des versions parallèles des méthodes $Durand-Kerner$ et $Ehrlish-Aberth$, nous avons tout d'abord cherché la partie du code la plus gourmande en temps *CPU* et/ou parallelisable, pour les deux(02) méthodes , puis nous avons parallélisé cette partie avec *OpenMP* et avec *CUDA C*.

Nous tenons à préciser que le degré minimal du polynôme que nous avons testé est le degré 20

3.4 Analyse et interprétation des résultats

Notre travail consiste à implémenter deux méthodes de résolutions de racines de polynômes creux déjà définies dans le chapitre précédent *Durand-Kerner* et *Ehrlish-Aberth* en C, de manière séquentielle puis parallèle sur CPU avec $OpenMP$, et sur GPU avec *CUDA C*.

3.4.1 Études séquentielles

- **Méthode de Durand-Kerner**

Résultats : La figure suivante illustre les résultats du temps d'exécution de l'implémentation de la méthode de *Durand-Kerner* selon le degré du polynômes (sa taille).

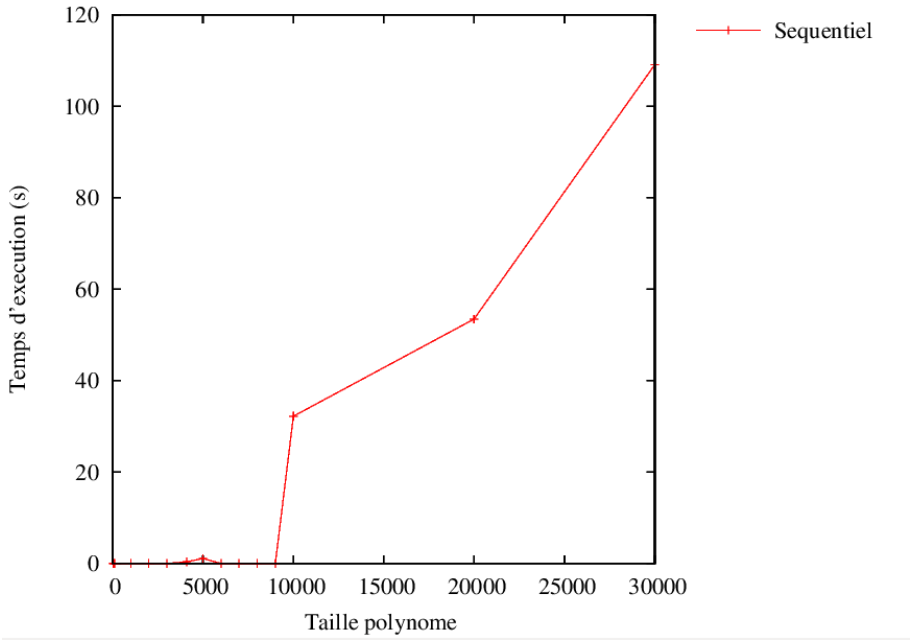

FIGURE 3.2 – Résultats de la méthode *DURAND-KERNER* en séquentiel

Discussions : Nous remarquons que le temps d'exécution dépend du degré du polynôme, plus le degré est grand , plus elle prend du temps pour la résolution du polynôme. Notons que la machine utilisée n'a pu atteindre un degré supérieur à *30 000*.

- **Méthode de Ehrlish-Aberth**

Résultats : La figure qui suit illustre les Résultats du temps d'exécution de l'implémentation de la méthode de *Ehrlish-Aberth* selon le degré du polynômes (sa taille).

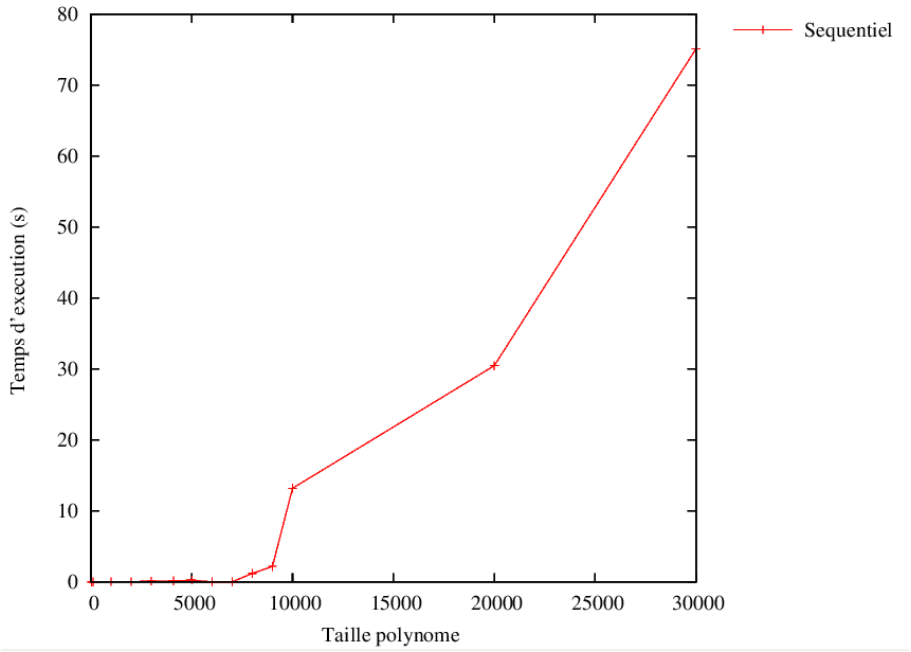

FIGURE 3.3 – Résultats de la méthode *Ehrlish-Aberth* en séquentiel

Discussions : Nous remarquons ici aussi que le degré du polynôme influence linéairement le temps d'exécution, c'est à dire plus grand sera le polynôme plus long serra le calcul.

- **Comparaison entre Durand-Kerner et Ehrlish-Aberth**

Résultats 1 : Dans la figure suivante, nous avons, sur un même graphe, les temps d'exécution des deux méthodes implémentées séquentiellement

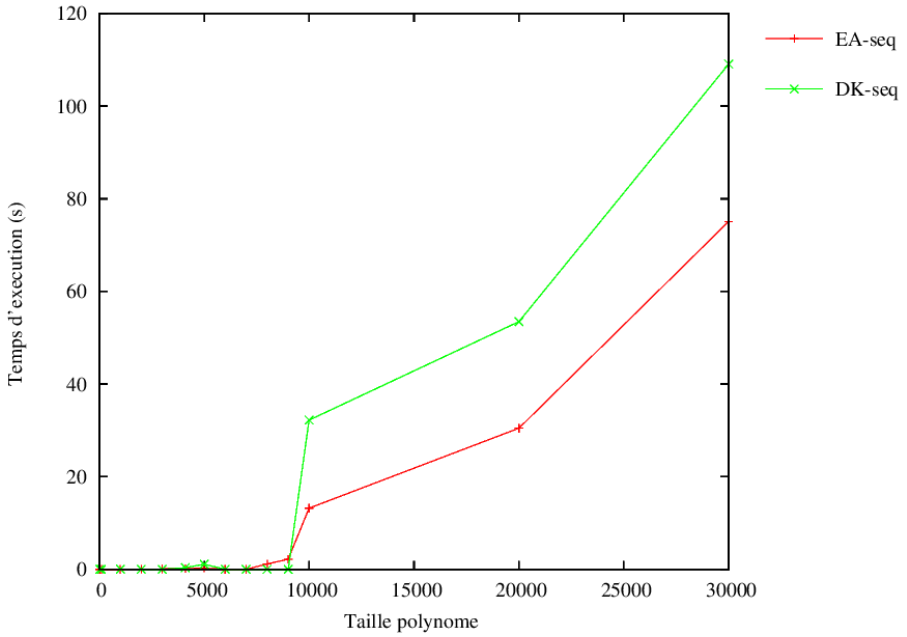

FIGURE 3.4 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth

Discussions 1 : Nous voyons clairement que la méthode de *Ehrlish-Aberth* (en rouge) est plus rapide que celle de *Durand-Kerner*(en vert) de prés de 30 secondes pour les polynômes de degré *30 000*, et cela s'explique par le faite que la méthode de *Ehrlish-Aberth* utilise la correction de $Newton\ P(z)/P'(z)$ ce qui lui permet de converger vers la solution plus rapidement.

Résultats 2 : La figure suivante illustre le nombre d'itération par rapport au degré des polynômes qu'execute les deux méthodes *Durand-Kerner* et *Ehrlish-Aberth* , pour converger et donc trouver les racines solutions du polynôme.

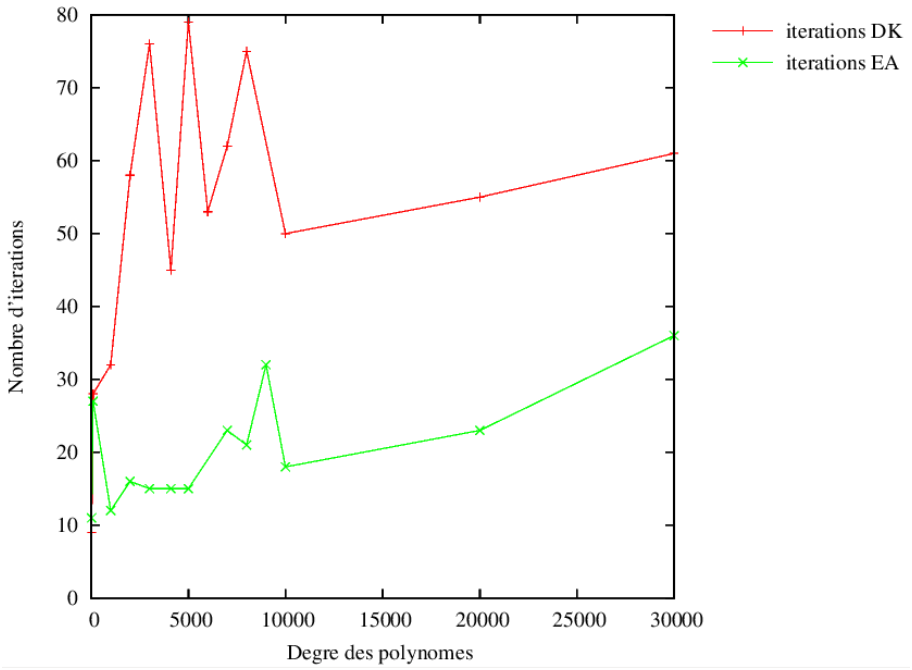

FIGURE 3.5 – Comparaison entre les méthodes *Ehrlish-Aberth* et *Durand-Kerner* sur le nombre d'itérations

Discussions 2 : Dans la figure ci-dessus, nous remarquons que la méthode de *Ehrlish-Aberth*(illustré en vert) a recours à moins d'itération avant de converger vers une solution ,par exemple pour la plus grande valeur du graphe qui est pour le degré *5000* est de *79* itérations avec la méthode de *Durand-Kerner* et de seulement *15* itérations avec *Ehrlish-Aberth* . Encore une fois, cela s'explique par son utilisation de la correction de $Newton$ qui accélére la convergence.

3.4.2 Études parallélisées

Grâce à *Valgrind* et *callgrind* deux outils de profilage ,nous obtenons ceci :

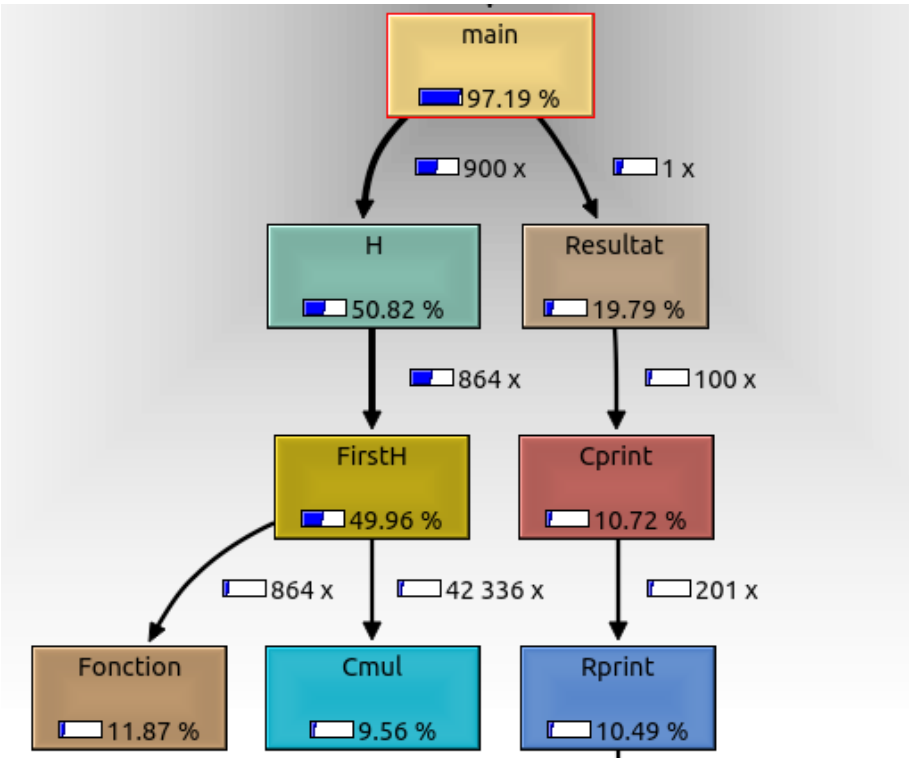

FIGURE 3.6 – Résultats de l'analyse de la méthode *Durand-Kerner*

Discussions : Dans la figure ci-dessus, nous voyons clairement que les fonctions *H()*et *First H()* prennent à elles deux 50% du temps d'exécution. Comme détaillées précédemment, ces fonctions sont celles qui font l'extraction des racines du polynôme et donc elle sont appelées tout au long de la résolution du polynôme et donc pour chaque itération.

- **Parallélisation avec OpenMP**

Avec *OpenMP*, il est possible de paralléliser une portion de code. Ce que nous voulons faire est de choisir la portion de code faisant appel à la fonctions *H* qui elle même appel *FirstH(), NewH() et Fonction()*. Par contre il faut prendre garde à ne pas créer un cas de concurrence [1] . ou d'interblocage [2] entre les processus ,la figure 3.7 est le résultat graphique du profilage après parallélisation de la seconde boucle $pour$ du $main()$ avec $OpenMP$:

1. La Concurrence correspondent à la situation dans laquelle se retrouvent plusieurs processus tentant d'accéder au même moment à une même ressource partagée(variables,fichiers,...)
2. L'interblocage se produit lorsque deux processus concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement.

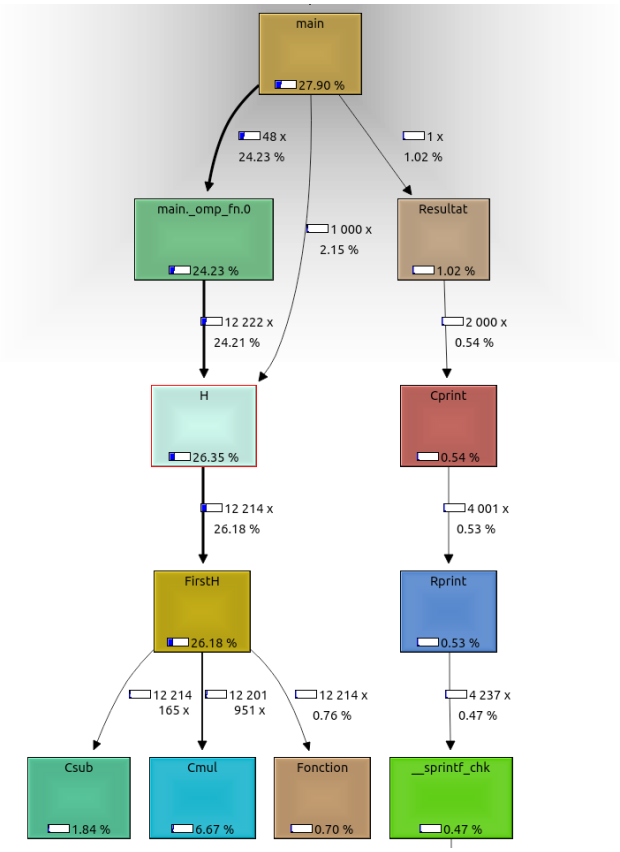

FIGURE 3.7 – Résultats de l'analyse de la méthode *Durand-Kerner* avec *OpenMP* avec 04 threads utilisées

Discussions : En observant cette figure ,nous remarquons que la fonction *H()* est passée d'une occupation du temps de 50% à seulement 25%, ce qui nous permet un gain de temps d'exécution sans générer ni interblocage ni concurrence entre les processus, pour ce faire, nous avons partagé les variables Z et $Zprec$ entre tous les *threads* afin qu'aucun d'eux n'ait de valeurs incohérente ou ne fausse les calculs.

- **Méthode de Durand-Kerner**

Résultats : Voici les résultats des temps d'exécutions de la méthode de *Durand-Kerner* selon la taille du polynômes avec différents nombres de threads utilisés, allant de un (01) *thread* (soit une exécution séquentielle) jusqu'à dix (10) *threads* :

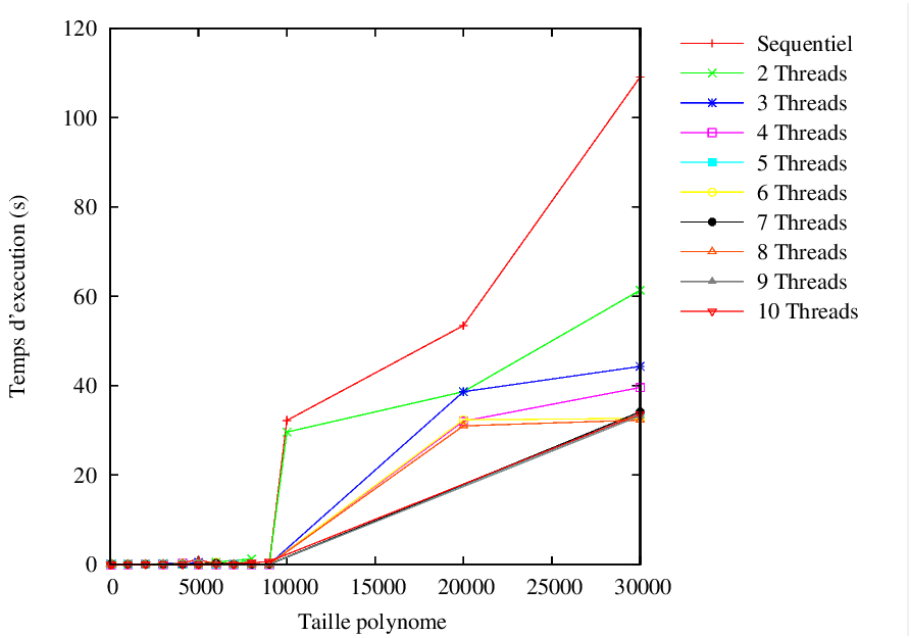

FIGURE 3.8 – Résultats de la méthode *DURAND-KERNER* avec*OpenMP* et nombre de threads de allant de 1 à 10.

Discussions : En analysant ces résultats, nous voyons clairement l'utilité de la parallélisation de l'algorithme. Plus le niveau de parallélisation est haut, en augmentant le nombre de *threads*,plus vite l'extraction de racines se fait , jusqu'à atteindre le maximum de parallélisation utile, c'est-à-dire utiliser les pleines capacités du processeur (CPU), au delà de ça, les threads doivent s'attendre mutuellement ce qui génère une perte de temps de calcul. Nous pouvons déduire aussi le nombre de *thread* le plus optimal à utiliser, avec la machine sur laquelle nous avons travaillé, le meilleur temps d'exécution est réalisé avec huit (08) *threads* , cela s'explique, par le fait que la machine dispose d'un processeur 8 cœurs lui permettant d'exécuter les huits (08) *threads* simultanément.

- **Méthode de Ehrlish-Aberth**

Résultats : La figure suivante correspond aux résultats des temps d'exécutions de la méthode de *Ehrlish-Aberth* selon la taille du polynômes avec différents nombres de *threads* utilisés, allant de un (01) *thread* (soit une exécution séquentielle) jusqu'à dix (10) *threads* :

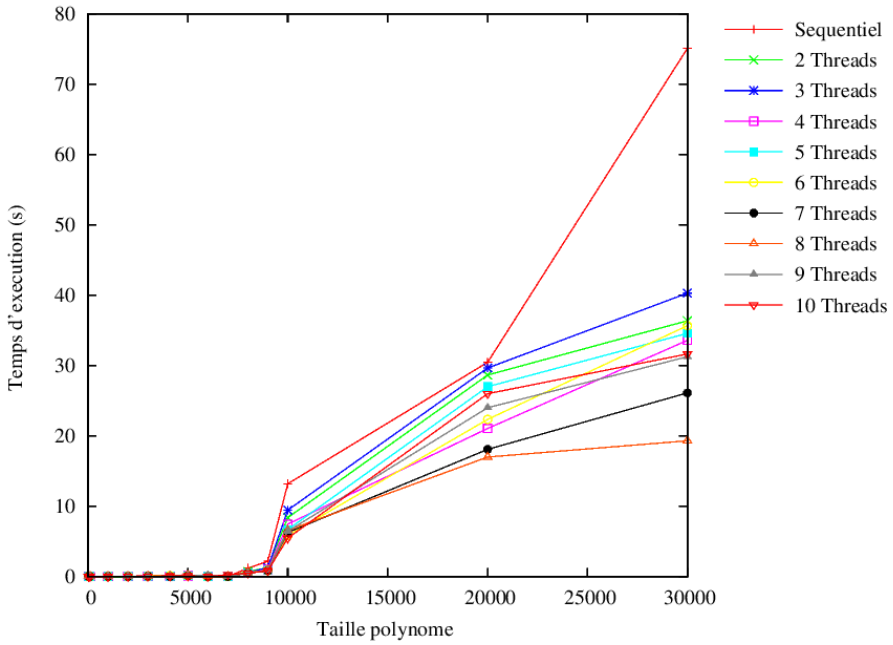

FIGURE 3.9 – Résultats de la méthode de *Ehrlish-Aberth* avec*OpenMP* et nombre de threads allant de 1 à 10.

Discussions : L'analyse de ces résultats nous montre quand le parallélisme est le plus optimal. Nous pouvons voir que pour le degré *30 000* le temps d'exécution n'est que de *19,3001 secondes* avec *huits (08) threads*, le choix sur le nombre de *threads* à utiliser est cette fois encore le nombre de cœurs de la machine, ceci permet de donner le meilleur des performance du *CPU* .

- **Comparaison entre Durand-Kerner et Ehrlish-Aberth avec OpenMP**

Résultats : Passons à la comparaison des deux méthodes en variant le nombre de *threads*. Dans notre expérience nous avons varié ce nombre jusqu'à dix (10) *threads* , ici nous exposons les résultats pour *2,3,4,8* et enfin *10 threads* sur les deux méthodes en simultané.

- **OpenMP avec 2 *threads* :**

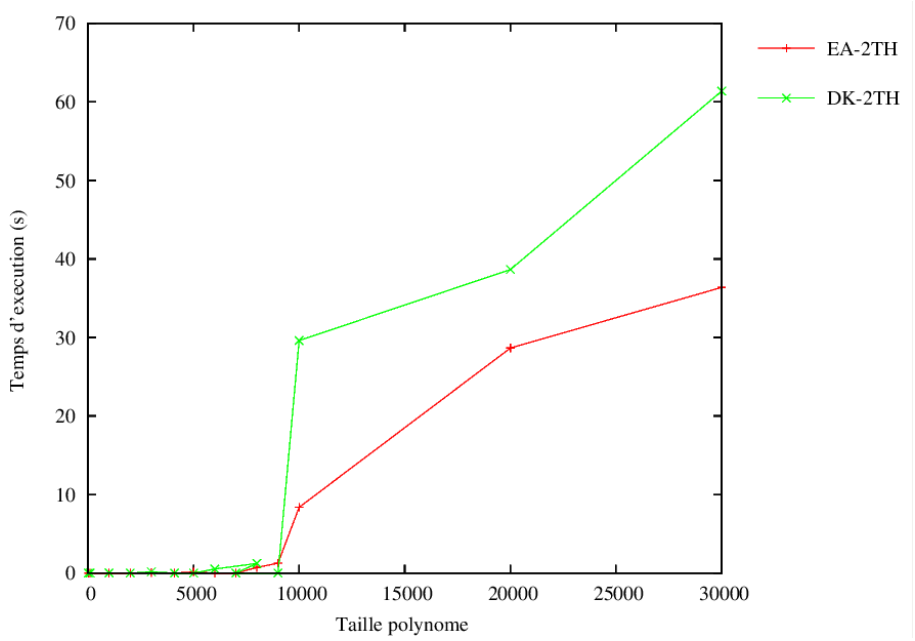

FIGURE 3.10 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth avec 02 threads

- **OpenMP avec 3 *threads* :**

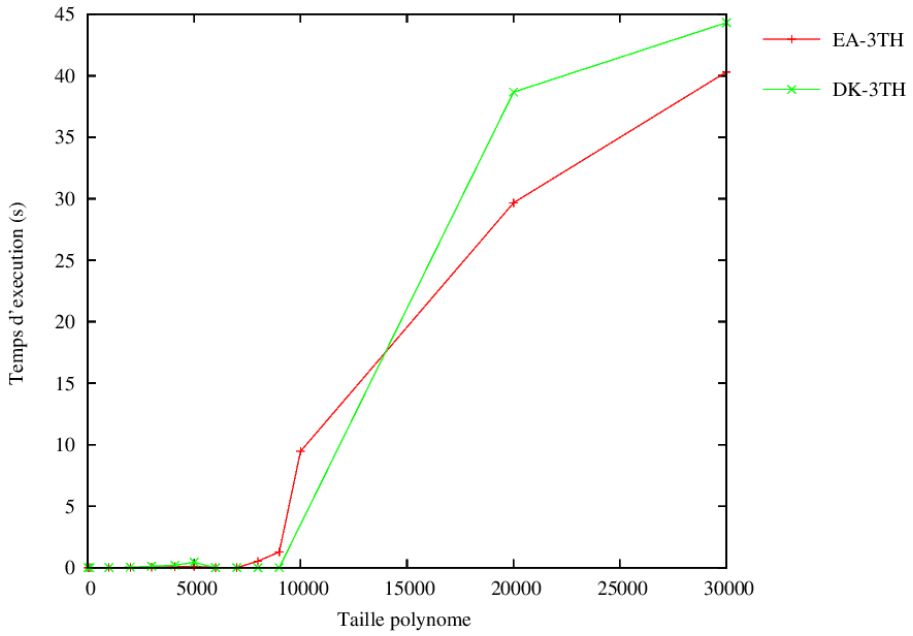

FIGURE 3.11 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth avec 03 threads

- **OpenMP avec 4 *threads* :**

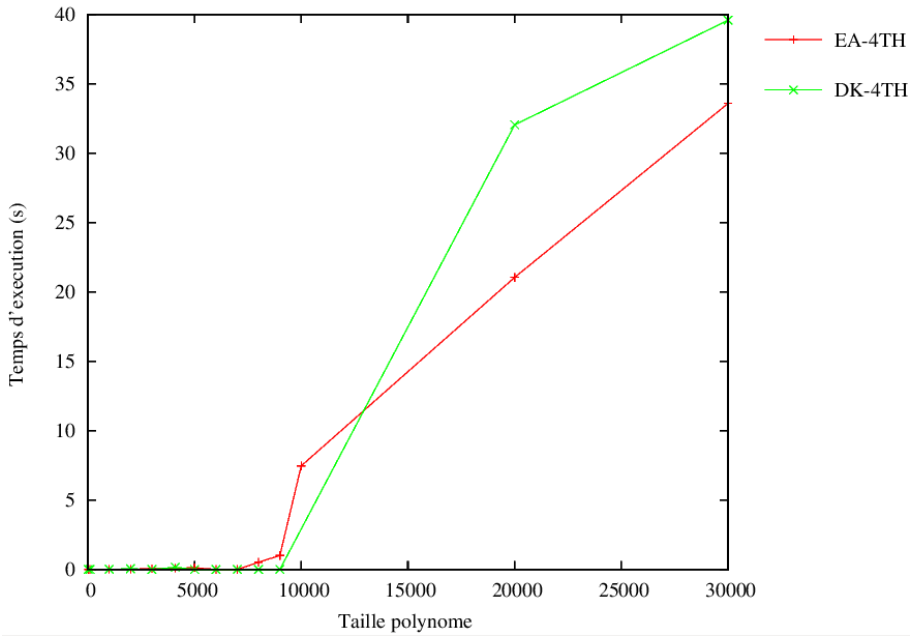

FIGURE 3.12 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth avec 04 threads

- **OpenMP avec 8 *threads* :**

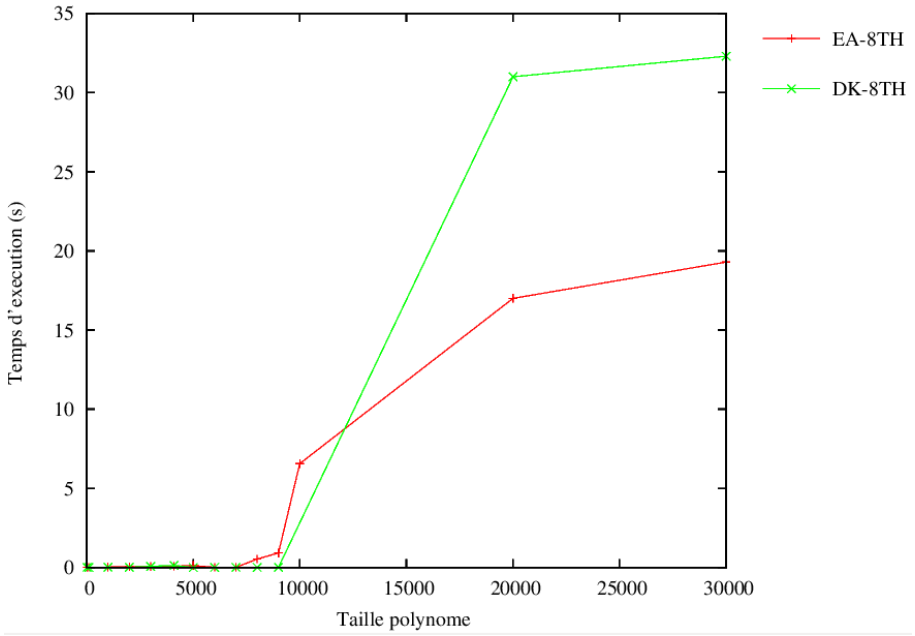

FIGURE 3.13 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth avec 08 threads

- **OpenMP avec 10 *threads* :**

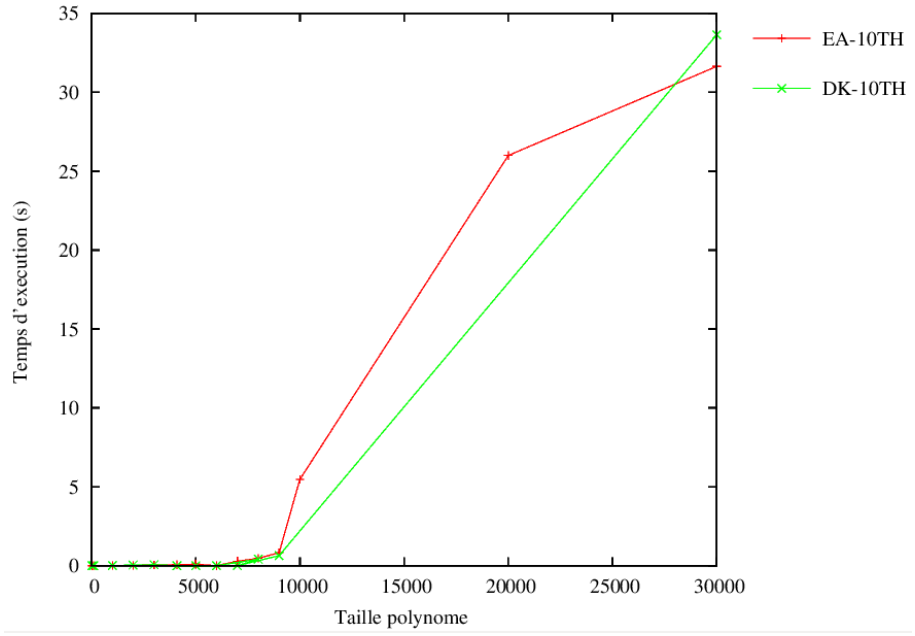

FIGURE 3.14 – Comparaison entre les méthodes Durand-Kerner et Ehrlish-Aberth avec 10 threads

Discussions : De tous les graphes précèdents, nous déduisons que :

- La méthode de *Ehrlish-Aberth* est plus rapide que celle de *Durand-Kerner* cela s'explique encore une fois par sa convergence rapide grâce à l'inclusion de la dérivé du polynôme P dans le calcul de ses racines.
- Une augmentation significative du temps d'exécution à partir du degré *8 000* passant de moins de deux secondes à plusieurs dizaines de secondes , cela s'explique par l'appel à la fonction *NewH* qui est apppelée qu'en cas de valeur de racine du polynôme trop grand.
- Pour les deux *(02)* méthodes *Durand-Kerner* et *Ehrlish-Aberth* le choix du nombre de *threads* afin d'avoir les résultats les plus optimaux est le même, il est égal huit (08) *threads* , ce qui veut aussi dire que ce choix n'est pas influencé par la méthode ou la taille du polynôme, mais par la nature de la machine utilisée plus précisemment de son processeur (*CPU*) étant donné que le calcul s'effectue sur ce dernier.

Résultats : Voici le tableau comparatif des temps d'exécutions *(en seconde)* pour les deux méthodes avec un polynôme de degré 30000 en séquentiel et avec *OpenMP* :

TABLE 3.1 – Tableau comparatif des deux méthodes au degré *30 000*

Méthode	Durand-Kerner (seconde)	Ehrlish-Aberth (seconde)
Séquentiellement	109,102	75,102
2 threads	61,379	36,379
3 threads	44,3113	40,3113
4 threads	39,5919	33,5811
5 threads	33,591	34,5919
6 threads	32,669	35,669
7 threads	34,1209	26,1209
8 threads	**32,3001**	**19,3001**
9 threads	33,2602	31,2602
10 threads	33,6416	31,6416

Discussions : Ce tableau nous apporte la même information que les graphes précèdents plus détaillé. Nous constatons que la méthode de *Ehrlish-Aberth* converge plus vite et choisir le nombre de cœurs comme nombre de *threads* pour *OpenMP* est le plus efficace et nous donne des résultats corrects en des temps raisonnables, pour *Durand-Kerner* nous avons tout de même réussi à gagner en temps de calcul avec $OpenMP$ et en utilisant huit (08) threads, il est à préciser que le degré 30000 est le degré le plus élevé qu'a supporté notre machine.

3.4.3 Parallélisation avec CUDA

Contrairement à un CPU, sur une carte graphique, les unités de calcul sont très nombreuses. L'utilisation d'un processeur graphique pour effecteur certaines parties du calcul en parallèle permet de multiplier les performances du programme d'un facteur très important, mais cela nécessite des connaissances du problème en question et des portions de traitements pour lesquels une exécution sur une GPU est favorable. Dans ce qui suit, nous présentons les étapes suivies pour implémenter les deux méthodes sur $CUDA$.

- **CPU vs GPU**

Un CPU est capable de traiter tout type de tâches. Le GPU, quant à lui va traiter un nombre de tâches plus limitées, mais va être capable de l'appliquer à un très grand nombre de données. Leur architecture utilise une forme de parallélisme qui sont les architecture SIMD, où chaque unité de calcul du processeur effectue le même traitement sur des données différentes [17].

Le processeur graphique (GPU) est bien distinct de son prédécesseur le processeur central (CPU). Nous remarquons la différence importante du nombre d'unités de calcul en vert (Figure 3.15) appelés threads sur GPU et processus sur CPU [17].

Cependant un ensemble de threads partage le même espace de mémoire virtuelle alors qu'un processus possède la sienne. Cela induit un temps d'accès différents aux données. Là où chaque processus peut accéder de manière simultanée aux données, les threads doivent le faire de manière séquentielle. De plus, les processus ont un espace de mémoire cache dédié bien plus important, ce qui réduit leur temps d'accès aux instructions et aux données [17].

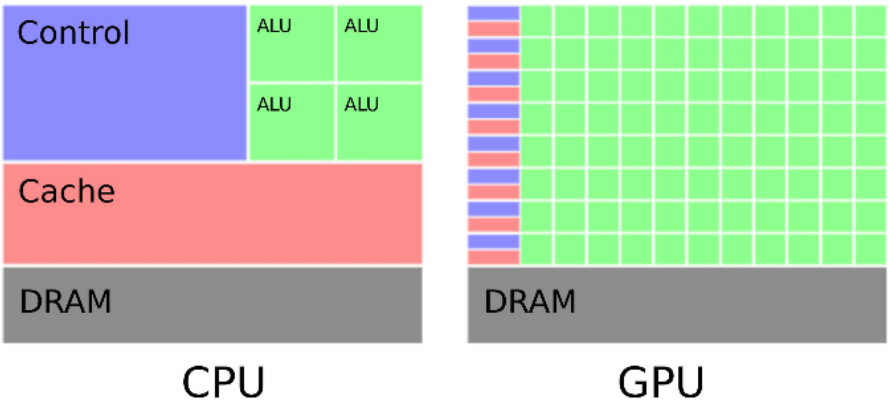

FIGURE 3.15 – CPU vs GPU (NVIDIA)

Un CPU dispose d'un nombre réduit d'unités de calcul, mais d'un cache d'une grande taille (plusieurs Mégaoctets) et une unité de contrôle importante. Cela est dû à la gestion de plusieurs tâches très différentes en parallèle qui nécessitent beaucoup de données. Ainsi, les données sont stockées en cache pour accélérer leur accès, et l'unité de contrôle va optimiser le flux d'instructions pour maximiser l'occupation des unités de calcul et optimiser la gestion du cache [17].

L'architecture GPU dispose d'un grand nombre d'unités de calcul qui disposent de peu de cache (quelques Kilooctets) et de faibles unités de contrôle. Cela lui permet de calculer de façon massivement parallèle le rendu de ces petits éléments indépendants, tout en ayant un débit important de données traitées[17].

Sur le GPU, les threads que nous pouvons lancer en parallèle sur la carte graphique sont organisés en groupe de threads. La figure 3.16 résume cette hiérarchie des threads avec un exemple en 2 dimensions [17].

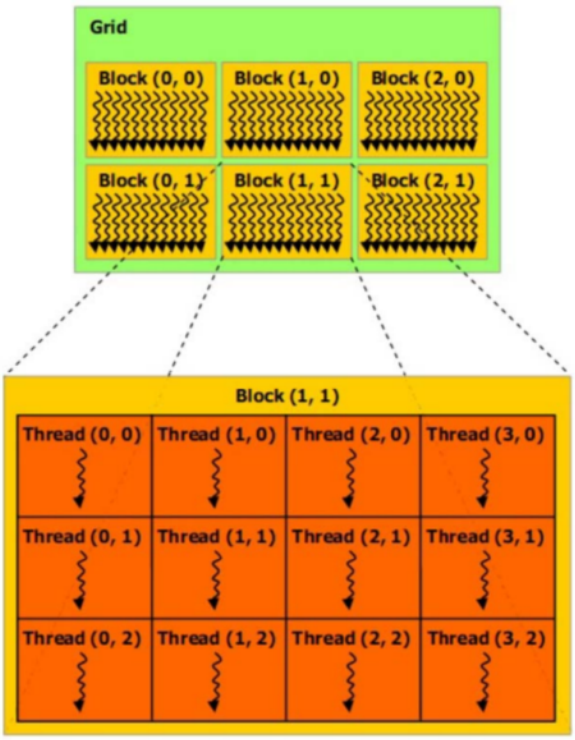

FIGURE 3.16 – Organisation des threads, blocks et grid dans une carte graphique [Nvidia]

Au plus bas niveau, il y a le thread. C'est la plus petite unité de traitement que nous pouvons lancer sur la carte graphique. Ce sont les threads qui exécutent les programmes en parallèle. Leur nombre est impressionnant : nous pouvons atteindre les 8 millions sur une simple carte GeForce 9300 GE[17].

Au niveau intermédiaire, il y a le block : c'est un regroupement de threads. Les threads d'un même block partagent une mémoire commune très rapide. La position d'un thread dans un block est repérée par ses coordonnées sur 2 ou 3 dimensions[17].

Au plus haut niveau, il y a le grid : c'est l'ensemble des blocks de la carte graphique. En fait, du point de vue des processus parallèles, le grid, c'est la carte graphique. Les mémoires qui lui sont liées, sont accessibles à tous les threads. De même que pour les threads dans un block, la position d'un block dans un grid est repérée par ses coordonnées sur 2 ou 3 dimensions [17].

- **CUDA C**

Pour pouvoir effectuer des opérations en paralléle sur le GPU en CUDA, il faut ecrire des fonctions appelées $kernels$. Ces fonctions sont comme les fonctions en C, mais n'auront pas le même comportement.Nous les appellons ensuite dans la fonction $main()$ en indiquant le nombre de blocs et le nombre de threads par bloc que nous voulons utiliser. Il faut préciser ici que c'est du travail du programmeur que de trouver la bonne organisation paralléle.

Le GPU est organisé en plusieurs $grilles$ ($grids$) qui sont elles mêmes constituées de plusieurs $blocs$ ($blocks$). Lors de l'utilisation du GPU, ces $blocs$ seront organisés de $threads$. Les langages tel que $CUDA$ ou $OpenCL$ permettent d'effectuer plusieurs opérations en parallèle. Ce type de programmation permet, dans des bonnes conditions d'utilisation, d'obtenir une vitesse d'exécution bien supérieure aux limites du paradigme séquentiel.

Il faut aussi noter que du fait de l'organisation dichotomique d'un GPU, le nombre de $threads$ et de $blocks$ est nécessairement une puissance de 2.

- **Déclaration de variables sur GPU :**

- Nombres scalaires sur CPU.
- Pour les vecteurs et matrices : $cudaMalloc$(void tab, size t dim) ;
- Pour les vecteurs et matrices : $cudaMalloc$(void tab, sizet dim) ;
- Pour copier un tableau CPUGPU (ou vice$-$versa) : $cudaMemcpy$(variabledestination, variablesource, dimension, Méthode) ; où Méthode prend la valeur $cudaMemcpyHostToDevice$(CPU vers GPU) ou $cudaMemcpyDeviceToHost$ (GPU vers CPU).
- L'espace allouée est libérée avec $cudaFree$.

- **Differents types de kernel :**

Il existe 3 types de $kernel$, et il relève du travail du développeur de préciser le type de chaque fonction déclarée dans un code CUDA, à l'aide d'un des mots clé suivants :

- —*global* — correspond à un $kernel$ exécuté sur le GPU mais appelé sur le CPU.
- —*device* — correspond à un $kernel$ exécuté et appelé sur le GPU.
- —*host* — correspond à un $kernel$ exécuté et appelé sur le CPU (comme une fonction normale en C).

La figure suivante détaille l'organisation particulère décrite plus haut (un GPU = des $grilles$, une $grille$ = des $blocs$ et chaque $bloc$ est utilisé avec plusieurs $threads$) :

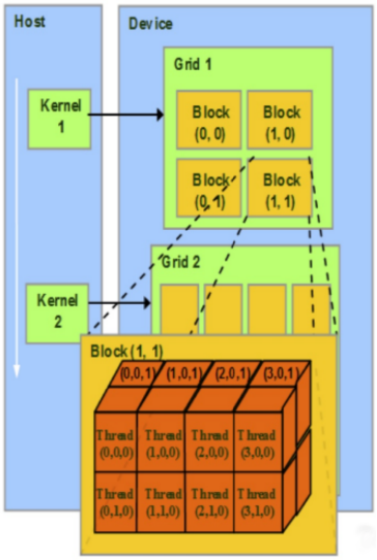

FIGURE 3.17 – Organisation d'un GPU

Un appel à une fonction f de type —*global* — doit être effectué comme suit : $f <<< dimGri, dimBloc >>> (arg1, arg2, ...)$ où $dimGri$ et $dimBloc$ sont, respectivement, deux structures de type $dim3$ (x, y et z) contenant le format de la $grille$ et le nombre de $threads$ par $bloc$.

Pour l'implémentation des deux méthodes sur GPU, nous avons procédé comme suit

- Redéfinition des nombres $Complexe$ le type étant déjà existant sur cuda dans la librairie $cuComplex.h$.
- Initialisation du Polynôme et du vecteur se font sur CPU
- Copie des valeurs initiales de Z, $Zprec$ sur le $Device$
- Création de trois $kernels$, le premier fait la mise à jour du vecteur Z en calculant sa valeur equivalent à $FirstH$ (CPU). Le deuxième $kernel$ calcule la convergence des racines, tire la maximale afin de tester la convergence, et le troisième $kernel$ sauvegarde

les valeurs de Z dans $ZPrec$ à chaque itération.

- **Parallélisation avec** $CUDA\ C$

Voici les différents types de l'algorthme utilisé pour faire du parallélisme sur GPU.

Algorithm 11: *main()*

$Entier\ i, iteration$;
$Complexe\ ZPrec, Z$;
$initialiser(P)$;
$Gegneihmer()$
pour *i=0 ;i<degrePolynome ;i++* **faire**
  $Z[i] = H(i, ZPrec)$;
Fin Pour
copie-variables-host-vers-device $(Z, ZPrec)$ //Z–>dZ et ZPrec–>dZPrec
répéter
  // Nous avons choisi DimGrid=1 soit une seule grille de calcul et DimBloc >= Degré polynome et multiple de 64
  <DimGrid,DimBloc> kernel-sauvgarde(dZ,dZPrec) ;//Sauvegarde des valeurs antérieur de dZ dans dZprec de toute les racines simultanément
  <DimGrid,DimBloc> kernel-maj(dZ) ;// pour la mise à jour des valeurs des racines
  <DimGrid,DimBloc> kernel-converg(dZ,dZPrec,C) ;// Calculer la convergence de toute les racines simultanément, et tirer la maximale qui est C
  copie-variables-device-vers-host (C) ; //Recopier seulement C pour test de convergence de la boucle répété.
  $iteration++$ //pour calculer le nombre d'itéartions
jusqu'à $C >= eps$;
copie-variables-device-vers-host (dZ,dZPrec) ;//
$Resultat()$; //Affichage des résultats

- **Résultat de la méthode de Durand-Kerner sur GPU**

Résultats : La courbe suivante montre les résultats obtenus après les tests effectués sur différentes taille de polynômes et en exécutant la version GPU du programme de $Durand\text{-}Kerner$.

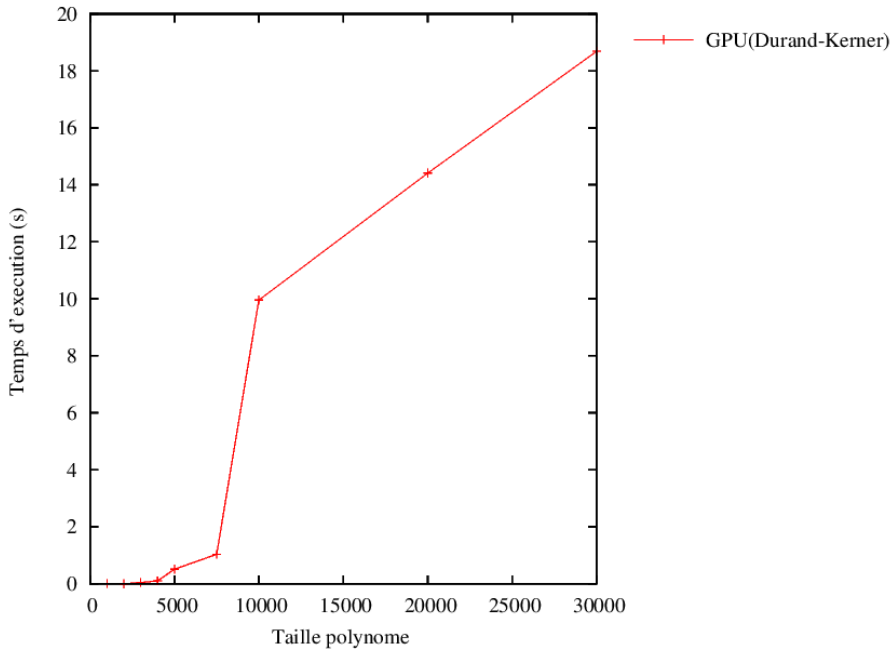

FIGURE 3.18 – Résultat de la méthode $Durand\text{-}Kerner$ sur GPU

Discussions : Nous remarquons que le temps d'exécution dépend du degré du polynôme, plus le degré est grand ,plus le programme prend du temps pour la résolution du polynôme ce qui est normal. Ce temps d'exécution reste néanmoins inférieur au temps d'exécution sur CPU.

- **Résultat de la méthode de Ehrlish-Aberth sur GPU**

Résultats : La courbe suivante montre les résultats obtenus après les tests effectués sur différentes taille de polynômes et en exécutant la version GPU du programme de $Ehrlish\text{-}Aberth$.

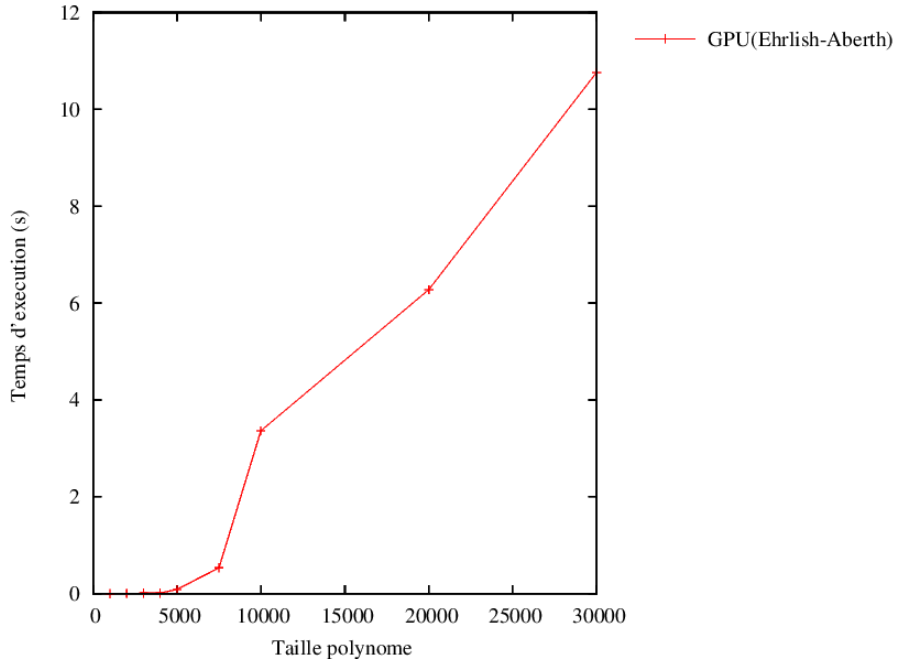

FIGURE 3.19 – Résultat de la méthode $Ehrlish\text{-}Aberth$ sur GPU

Discussions : Nous remarquons que le temps d'exécution dépend du degré du polynôme, plus le degré est grand ,plus le programme prend du temps pour la résolution du polynôme ce qui est normal. Ce temps d'exécution reste néanmoins inférieur au temps d'exécution sur CPU.

- **Comparaison entre la méthode Durand-Kerner sur CPU vs GPU**

Résultats : La figure 3.20 montre les courbes des résultats de l'implémentation séquentielle et paralléle sur CPU et la version GPU de la méthode de $Durand\text{-}Kerner$.

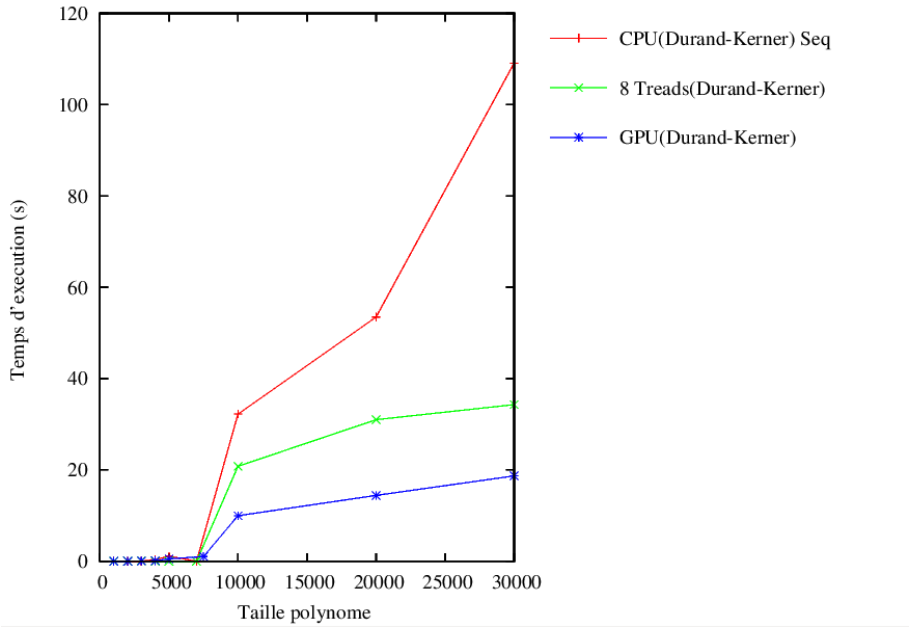

FIGURE 3.20 – Comparaison de la méthode $Durand\text{-}Kerner$ sur CPUs vs GPU

Discussions : Nous remarquons que la version GPU donne des résultats interessant par rapport à la version CPU que ça soit séquentiel ou paralléle, prenons pour exemple le degré *30 000* oú nous avons un gain de *90 secondes* par rapport aux résultats séquentiels et *15 secondes* par rapport à la version paralléle avec $OpenMP$. Nous déduisons que le calcul sur GPU est beaucoup plus performant, ceci est dû au fait que le processeur graphique utilise plusieurs unités de calcul , ce qui permet de mieux exploiter le parallélisme.

- **Comparaison entre la méthode Ehrlish-Aberth sur CPU vs GPU**

Résultats : La figure suivante montre les courbes des résultats de l'implémentation séquentielle et paralléle sur CPU et la version GPU de la méthode de $Ehrlish\text{-}Aberth$.

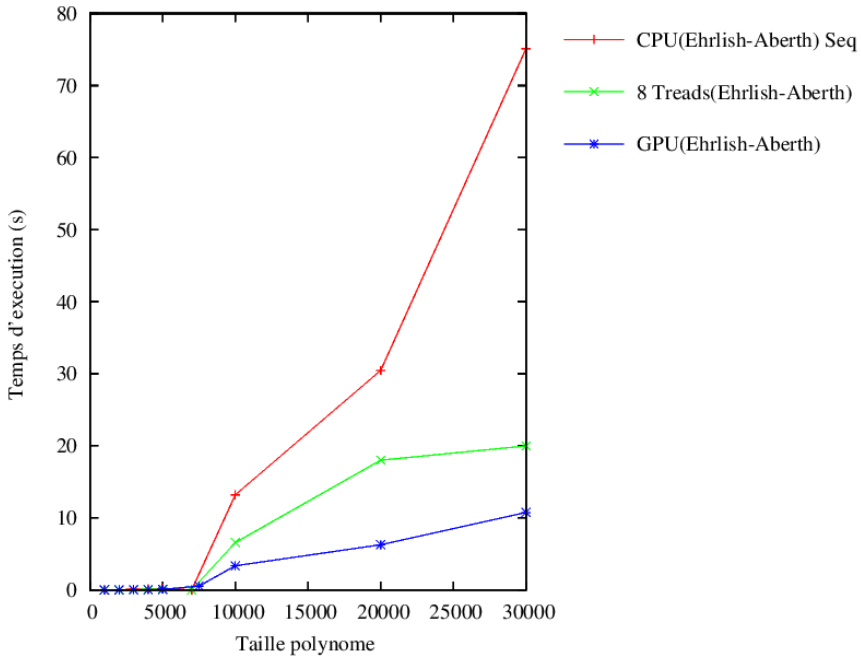

FIGURE 3.21 – Comparaison de la méthode $Ehrlish\text{-}Aberth$ sur CPUs vs GPU

Discussions : Nous observons que les résultats obtenus avec le calcul sur GPU est meilleur et plus performant que sur CPU. Bien que la méthode de Ehrlish-Aberth nous a donné de bons résultats en CPU, en GPU les résultats sont plus remarquables. Comme le souligne l'exmple suivant : oú nous avons pour un polynôme de degré *30 000*, un temps d'exécution séquentiel de *75.10 secondes* et paralléle avec $OpenMP$ de *19.98 secondes* contre un temps d'exécution record de *10.762 secondes* en GPU.

3.5 Conclusion

Nous avons vu dans ce chapitre l'ensemble de moyens mis en œuvre pour mener à bien l'étude que nous avons réalisée, aussi bien les moyens logiciels que les moyens matériels. Nous avons exposé les implémentations réalisées des deux méthodes $Durand\text{-}Kerner$ et $Ehrlish\text{-}Aberth$ en langage C sur CPU d'abord en séquentiel puis en paralléle grâce à OpenMP . Nos tests nous ont permis de mieux calibrer les paramétres de notre programmation parallèle et nous avons suivi par leurs implémentations sur GPU en langage $CUDA\,C$ en expliquant la structure de l'architecture CUDA et ses avantages. Pour finir, nous avons donné les resultats de nos tests sur CPU et GPU afin de les comparer. Nous avons pu conclure du grand écart existant entre les capacités du CPU et du GPU, ce qui explique l'interêt pour la programmation GPU de ces derniers années.

CONCLUSION GÉNÉRALE & PERSPECTIVES

Dans ce mémoire, notre motivation est portée sur la nécessité de trouver un moyen plus facile de réduire le temps d'exécution d'un programme d'extraction de racines de polynôme de degré elevé. Nous avons opté pour l'étude, l'implémentation séquentielle et parallèle et la comparaison des résultats des deux méthodes $Durand\text{-}Kerner$ et $Ehrlish\text{-}Aberth$. Notre approche consiste non seulement à minimiser le temps d'execution grâce à la parallélisation sur CPU mais aussi sur GPU.

Au cours de ce travail nous avons procédé comme suit : La première étape consiste à implémenter les deux méthodes séquentiellement , de comparer les résultats d'execution obtenus, l'implémentation des méthodes s'est faite avec le langage C, les résultats sont obtenus par des graphes que nous visualisons grâce à l'outil $GnuPlot$.

La deuxième étape consiste à implémenter les deux méthodes $Durand\text{-}Kerner$ et $Ehrlish\text{-}Aberth$ en parallèle grâce à $OpenMp$, sur CPU, et à faire une comparaison avec le séquentiel, ainsi que la comparaison entre les deux méthodes en utilisant différents nombre de threads et en illustrant grâce à $GnuPlot$, le nombre de threads qui nous donne les meilleurs résultats en terme de temps d'execution.

La troisième étape consiste à implémenter les deux méthodes avec $CUDA$, l'une des plus récente technique de parallélisation encore rarement utilisée, exploitant sa capacité à traiter d'immenses collections de données d'une manière parallèle, mais cette fois sur GPU, qui atteint un bon niveau de performance et une qualité de solution correcte. Comme dans nos attentes , la version GPU de notre implémentation reste la meilleure car dans celle-ci nous exploitons au maximum le processeur graphique doté d'un très bon niveau de performance, ce qui a conduit à des résultats très satisfaisants en terme de temps d'exécution.

Comme perspectives nous proposons l'amélioration de la performance au niveau séquentiel par le biais d'optimisation de structures mémoires, des révisions des segments de codes redondants, etc. Nous proposons aussi d'executer ces implémentations sur des $Supercalculateurs$ et/ou sur un système distribué pour atteindre la résolution de polynômes de degré très élevé .

En analyse numérique, la méthode de surrelaxation successive est une variante de la méthode de $Gauss$-$Seidel$ pour l'extraction des racines. La convergence de cet algorithme est généralement plus rapide. Une approche similaire peut être appliquée à bon nombre de méthodes itératives. Nous envisageons d'implanter la méthode de surrelaxation successive dans le calcul des racines de $Durand$-$Kerner$ et $Ehrlish$-$Aberth$, afin d'arriver encore plus rapidement aux résultats obtenus.

Références bibliographiques & Webographiques

[1] Site Web de ANSYS, Inc. : *www.ansys.com*, consulté en mai 2014.

[2] Site Web de GnuPlot : *www.gnuplot.sourceforge.net*, consulté en mai 2014.

[3] Site Web de NVIDIA Corp : *www.docs.nvidia.com*, consulté en mars 2014.

[4] Site Web de OpenMP :*www.openmp.org*, consulté en janvier 2014.

[5] Site Web de Programme Du Peuple : *www.progdupeu.pl*, consulté en janvier 2014.

[6] Site Web de Valgrind : *www.valgrind.org*, consulté en janvier 2014, 2014.

[7] A.DARIO, BINI, L.GEMIGNANI, and F.TISSEUR. THE EHRLICH-ABERTH METHOD FOR THE NONSYMMETRIIC TRIDIGONAL EIGENVALUE PROBLEM. 2005.

[8] Ronald N Bracewell. Fourier Transform and its Applications. 1999.

[9] Alexandre ERN and Gabriel STOLTZ. Calcul scientifique parallèle, 2013.

[10] F.Magoulés and F-X.Roux. *Calcul scientifique parallèle*, 2013.

[11] M.GILLI. *Méthodes numérique*, March 2006.

[12] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2011.

[13] R.Couturier and F.SPIES. Extraction de racines dans des polynômes creux de degré élevé.

[14] D.M. Ritchie and B.W. Kernighan. *The C programming language*. Bell Laboratories, 1988.

[15] TARIK SAIDANI. *Optimisation multi-niveau d'une application de traitement d'images sur machines parallèles*. PhD thesis, Université PARIS-SUD Ecole doctorale STITS : institut d'Electronique Fondamentale, 06/11/2012.

[16] Gabriel Stoltz. *Introduction au Calcul Scientifique*. juillet 2013.

[17] Guillaume VIGUIE. Apport de la programmation graphique pour la reconstruction rapide d'images 3d en tomographie par emission monophotonique., thése de doctorat, 2008.

Annexe A

Implantations séquentielles et paralléles

A.1 Caractériques de la carte graphique

```
Device 0 : "GeForce GT 740M"
CUDA Driver Version / Runtime Version 6.0 / 5.5
CUDA Capability Major/Minor version number : 3.5
Total amount of global memory : 2048 MBytes (2147352576 bytes)
( 2) Multiprocessors, (192) CUDA Cores/MP : 384 CUDA Cores
GPU Clock rate : 1032 MHz (1.03 GHz)
Memory Clock rate : 900 Mhz
Warp size : 32
Memory Bus Width : 64-bit
L2 Cache Size : 524288 bytes
Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory : 65536 bytes
Total amount of shared memory per block : 49152 bytes
Total number of registers available per block : 65536
Maximum number of threads per multiprocessor : 2048
Maximum number of threads per block : 1024
Max dimension size of a thread block (x,y,z) : (1024, 1024, 64)
Max dimension size of a grid size (x,y,z) : (2147483647, 65535, 65535)
Maximum memory pitch : 2147483647 bytes
Texture alignment : 512 bytes
Concurrent copy and kernel execution : Yes with 1 copy engine(s)
Run time limit on kernels : Yes
Integrated GPU sharing Host Memory : No
Support host page-locked memory mapping : Yes
Alignment requirement for Surfaces : Yes
Device has ECC support : Disabled
Device supports Unified Addressing (UVA) : Yes
Device PCI Bus ID / PCI location ID : 1 / 0
Compute Mode :
< Default (multiple host threads can use : :cudaSetDevice() with device simultaneously) >
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.0, CUDA Runtime Version = 5.5 , NumDevs = 1, Device0 =
GeForce GT 740M
Result = PASS
```

A.2 Fonctions de calculs sur les complexes

Structure complexe :

```
typedef struct complexe { double a, b ; } complexe ;
```

FIGURE A.1 – structure complexe en C

Fonction multiplication :

```
complexe Cmul(complexe Cp1, complexe Cp2)
{
  complexe Cp ;

  Cp.a = Cp1.a*Cp2.a - Cp1.b*Cp2.b ;
  Cp.b = Cp1.a*Cp2.b + Cp1.b*Cp2.a ;
  return Cp ;
}
```

FIGURE A.2 – Fonction multiplication des complexes en C

Fonction division :

```
complexe Cdiv(complexe Cp1, complexe Cp2)
{
  complexe Cp ;

  Cp.a = (Cp1.a*Cp2.a + Cp1.b*Cp2.b) / (Cp2.a*Cp2.a + Cp2.b*Cp2.b) ;
  Cp.b = (Cp1.b*Cp2.a - Cp1.a*Cp2.b) / (Cp2.a*Cp2.a + Cp2.b*Cp2.b) ;
  return Cp ;
}
```

FIGURE A.3 – Fonction division des complexes en C

Fonction addition :

```
complexe Cadd(complexe Cp1, complexe Cp2)
{
  complexe Cp ;

  Cp.a = Cp1.a + Cp2.a ;
  Cp.b = Cp1.b + Cp2.b ;
  return Cp ;
}
```

FIGURE A.4 – Fonction addition des complexes en C

Fonction soustraction :

```
complexe Csub(complexe Cp1, complexe Cp2)
{
  complexe Cp ;

  Cp.a = Cp1.a - Cp2.a ;
  Cp.b = Cp1.b - Cp2.b ;
  return Cp ;
}
```

FIGURE A.5 – Fonction soustraction des complexes en C

Fonction module :

```
double Cmodule(complexe Cp) {

        if(Cp.a*Cp.a + Cp.b*Cp.b<0) {printf("pb Cmodule\n");exit(0);}
  return sqrt(Cp.a*Cp.a + Cp.b*Cp.b) ;
}
```

FIGURE A.6 – Fonction calcul du module d'un complexe en C

Résumé

L'informatique vise depuis sa naissance à résoudre plus rapidement des problèmes coûteux en temps de calcul, elle a recours à plusieurs domaines , tels que la simulation numérique, la cryptographie et l'imagerie. Dans ce projet nous nous sommes interessés à la simulation numérique qui a pour domaine le calcul scientifique. Le but est d'étudier et par la suite implémenter deux méthodes d'extraction de racines de polynôme $Durand\text{-}Kerner$ et $Ehrlish\text{-}Aberth$, en sequentiel et paralléle avec $OpenMP$ en langage C sur CPU. Notre travail consiste aussi à paralléliser ces méthodes sur GPU basé sur l'exploitation des processeurs graphiques avec $CUDA\ C$ exécutable sur les technologies $NVIDIA$. Ces implémentations parallèlisées ont pour objectif la diminution du temps de calcul et celà en exploitant pleinement le CPU et le GPU. Ces approches ont été validées à l'aide des résultats obtenus et en comparant en terme de temps d'execution les différents cas.

Mots clés **:** CUDA C,$Durand\text{-}Kerner$, $Ehrlish\text{-}Aberth$,extraction de racine de polynôme,GPU,OpenMP.

Abstract

The Computer science since its inception aims to faster solve costly problems on computation time, it uses several field such as computer simulation, cryptography and imagery. In this work we were interested in the numerical simulation that has for domain scientific computing , the goal is to study and then implement two root extraction methods polynomial $Durand\text{-}Kerner$ and $Ehrlish\text{-}Aberth$, in sequential and parallel with $OpenMP$ into language C on CPU. Our job is also to parallelize these methods on GPU based on the exploitation of the graphic processes with $CUDA\ C$ executable on $NVIDIA\ GPU$ cards. These parallelized implementations capabilities are intended to decrease the computational time by fully exploiting CPU and GPU. These approaches were validated using the results and comparing in terms of execution time the different cases.

Key words **:** CUDA C,$Durand\text{-}Kerner$, $Ehrlish\text{-}Aberth$,roots extraction of polynomial,GPU,OpenMP.