

ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

polytech.univ-tours.fr

Projet Recherche & Développement

2018-2019

Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile

Problème de la séquence la plus courte

**POLYTECH[®]**
TOURS

Entreprise

Polytech'Tours



Tuteur entreprise

Patrick MARTINEAU

Étudiant

Ayoub IDEL (DI5)

Tuteur académique

Patrick MARTINEAU

Liste des intervenants

Entreprise

Polytech'Tours
64 Avenue Jean Portalis,
37200 Tours
polytech.univ-tours.fr



Nom	Email	Qualité
Ayoub IDEL	ayoub.idelmahjoub@etu.univ-tours.fr	Étudiant DI5
Patrick MARTINEAU	patrick.martineau@univ-tours.com	Tuteur académique, Département Informatique
Patrick MARTINEAU	patrick.martineau@univ-tours.com	Tuteur entreprise



Avertissement

Ce document a été rédigé par Ayoub IDEL susnommé l'auteur.

L'entreprise Polytech'Tours est représentée par Patrick MARTINEAU susnommé le tuteur entreprise.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Patrick MARTINEAU susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Ayoub IDEL, *Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile: Problème de la séquence la plus courte*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
  author={IDEL, Ayoub},
  title={Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile:
    Problème de la séquence la plus courte},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2018-2019}
}
```

Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	v
1 Introduction	1
1 Contexte	1
2 Objectifs	2
3 Hypothèses	3
4 Bases méthodologiques	3
2 Description générale	4
1 Environnement du projet	4
2 Caractéristiques des utilisateurs	4
3 Fonctionnalités du système	5
4 Structure générale du système	6
3 État de l'art	8
1 Structure des Systemes de calcul parallél	8
1.1 Amélioration des systèmes de calcul parallél	8
1.1.1 Les Ordinateur hautes performances (HPC).....	11
1.1.2 Les Clusters	12
1.1.3 Grilles de calcul	13

2	Programmation parallèle	14
2.1	Les modèles de programmation parallèle	14
2.1.1	Mémoire partagée (paralléliser de données).....	14
2.1.2	Mémoire distribuée (paralléliser des tâches)	16
2.1.3	Modèle hybride	18
2.2	Les langages, API et frameworks.....	19
2.3	Les avantages du modèle hybride	24
3	Conclusion	25
4	Analyse et conception	30
1	Problème de la séquence la plus courte	30
1.1	Problème NP-difficile	30
1.2	Explication du problème de la séquence la plus courte	30
1.3	Modélisation mathématique.....	31
1.3.1	Modélisation de Miller-Tucker-Zemlin	31
1.3.2	Modélisation de Dantzig-Fulkerson-Johnson	32
1.4	Conception de l'algorithme de résolution.....	33
5	Mise en oeuvre	35
1	Les librairies choisies.....	35
1.1	MPI.....	35
1.2	OpenMP	35
1.3	CUDA	35
1.4	Langage de programmation.....	36
2	Framework CUNIT	36
3	Intégration continue (GITLAB-CI).....	36
4	Versioning du code (GITLAB)	38
5	Revue du code (INDENT).....	38
6	Documentation DOXYGEN	38
7	Identification et gestion des risques.....	39
8	risques humains.....	40
9	risques sur les délais (deadline)	40
10	risques techniques	40
11	Résultats.....	40
11.1	Fichier entrée/sortie	40
11.2	Performances.....	41
6	Conclusion	43
1	Ce qui a été fait	43
2	Ce qui reste à faire	44

3	Planning.....	44
4	Bilan sur la qualité	44
5	Bilan auto-critique.....	45
Annexes		46
A Spécifications fonctionnelles		47
1	Diagramme de structure	47
2	Définition de la fonction 1 : get data from file	48
3	Définition de la fonction 2 : write into log file	48
4	Définition de la fonction 3 : compare costs	49
5	Définition de la fonction 4 : Create a queue	49
6	Définition de la fonction 5 : enqueue an element	49
7	Définition de la fonction 6 : dequeue an element	50
8	Définition de la fonction 7 : Generate next level.....	50
9	Définition de la fonction 8 : Generate any level	50
10	Définition de la fonction 9 : Generate leaves.....	51
11	Définition de la fonction 10 : Calculate Total Cost	51
12	Définition de la fonction 11 : Compare All Costs.....	52
B Spécifications non fonctionnelles		53
1	Contraintes de développement et conception	53
2	Contraintes de fonctionnement et d'exploitation.....	53
2.1	Performances.....	53
2.2	Capacités.....	54
2.3	Modes de fonctionnement	54
2.4	contrôlabilité.....	54
2.5	Sécurité	54
C Gestion de projet		55
1	Aperçu de gestion de projet	55
2	Découpage en tâches	57
2.1	Tâche 1 : Compréhension du sujet et du contexte	57
2.2	Tâche 2 : Etude de faisabilité.....	57
2.3	Tâche 3 : Analyse du problème de la séquence la plus courte.....	57
2.4	Tâche 4 : Etudier quelques algorithmes d'application.....	58
2.5	Tâche 5 : Etude des différentes technologies	58
2.6	Tâche 6 : Rédaction de l'état d'art	58
2.7	Tâche 7 : Rédaction du cahier des spécifications	58
2.8	Tâche 8 : Conception de la première version séquentielle de l'algorithme...	58

2.9	Tâche 9 : Rédaction du rapport final pour S09	58
2.10	Tâche 10 : Préparation de la soutenance mi-parcours	59
2.11	Tâche 11 : Conception et modélisation de premiers algorithmes	59
2.12	Tâche 12 : Développement des méthodes pour la gestion des fichiers	59
2.13	Tâche 13 : Développement de l'algorithme qui génère les décisions	59
2.14	Tâche 14 : Développement de la file d'attente	60
2.15	Tâche 15 : Développement des fonctions coté Client	60
2.16	Tâche 16 : Développement de la partie séquentielle	60
2.17	Tâche 17 : Développement de la partie MPI	60
2.18	Tâche 18 : Développement de la partie OpenMP	60
2.19	Tâche 19 : Développement de la partie CUDA	61
2.20	Tâche 20 : Conception des tests.....	61
2.21	Tâche 21 : Implémentation des tests	61
2.22	Tâche 22 : Exécution des tests	61
2.23	Tâche 23 : Rédaction du rapport final.....	61
2.24	Tâche 24 : Préparation de la soutenance S10	61
3	Planning.....	62
3.1	Le planning de recherche	62
3.2	Le planning de développement.....	62
D	Manuel d'installation	64
E	Manuel de configuration	65
F	Cahier de développeurs	66
G	Cahier de tests	67
H	Bibliographie	68
	Comptes rendus hebdomadaires	69

Table des figures

2 Description générale

1	Tableau des caractéristiques des utilisateurs	5
2	Le diagramme de cas d'utilisation	5
3	Le diagramme de séquence	6
4	Les différentes parties du système	7

3 État de l'art

1	Croissance du nombre de transistors dans les microprocesseurs Intel par rapport à la loi de Moore. En vert, la prédiction initiale voulant que ce nombre double tous les 18 mois	9
2	Loi d'Amdahl	10
3	Loi de Gustafson.....	11
4	Exemple d'Architecture d'une grille de calcul	14
5	Architecture d'un système à mémoire partagée	15
6	n système distribué connecte les processeurs via un réseau de communication	16
7	Architecture d'un système distribué.....	17
8	Architecture Hybride (OPENMP + MPI).....	18
9	Threads et leur affectation par l'OS.....	26
10	Variables privées et partagée.....	27
11	Exemple d'association entre processus MPI et matériel physique	27
12	Exemple d'une Grid	28
13	Exécution d'un programme CPU - GPU	29

4 Analyse et conception

1	Exemple des points de départ	31
---	------------------------------------	----

2	L'arbre des décisions.....	33
5	Mise en oeuvre	
1	Les pipelines	37
2	Les pipelines	37
3	Les pipelines	38
4	Exemple d'une documentation DOXYGEN	39
5	Exemple d'un fichier d'entrée	40
6	Exemple d'un fichier de sortie	41
7	Temps d'exécution en fonction de la longueur de la séquence	42
A	Spécifications fonctionnelles	
1	Temps d'exécution en fonction de la longueur de la séquence	47
C	Gestion de projet	
1	Modèle en cascade	55
2	Diagramme de Gantt	56
3	Diagramme de Gantt	56
4	Diagramme de Gantt : Les différentes tâches	56
5	Diagramme de Gantt : Les différentes tâches	57
6	Diagramme de Gantt : Planning de recherche et documentation	62
7	Diagramme de Gantt : Planning de développement	63

1

Introduction

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon PRD et qui m'ont aidé lors de la rédaction de ce rapport.

Tout d'abord, j'adresse mes remerciements à mon professeur et tuteur, Mr Patrick MARTINEAU qui m'a beaucoup aidé et su me guider dans ma recherche. Son écoute et ses conseils m'ont permis de cibler les objectifs attendus.

Je tiens également à remercier Mr Jean-Yves RAMEL ainsi que toutes les personnes qui m'ont conseillé et relu lors de la rédaction de ce rapport.

Ce document constitue les spécifications de Projet de Recherche et de Développement « Parallélisme » .

Ce sujet de projet est proposé par Monsieur Patrick MARTINEAU qui représente le MOA (Maitrise d'ouvrage) au sein de l'École Polytechnique de l'Université de Tours. Le projet sera réalisé par Ayoub IDEL, élève ingénieur en 5ème année à Polytech Tours.

1 Contexte

Presque tous les calculs effectués au cours des quarante premières années de l'histoire de l'informatique pourraient être appelés séquentiel.

Une des caractéristiques du calcul séquentiel est qu'il utilise un seul processeur pour résoudre un problème (ici, le terme « problème » est utilisé au sens large, c'est-à-dire que l'on exécute une seule tâche). Ces processeurs étaient devenus de plus en plus rapides et moins cher au cours des trois premières décennies, en doublant leur vitesse tous les deux ou trois ans. Cependant, en raison de la limite imposée par la vitesse de la lumière, il semble extrêmement improbable que nous puissions construire des Ordinateurs ne contenant qu'un seul processeur et pouvant atteindre des performances nettement supérieures à 1 000 000 000 d'opérations par seconde (Gflops).

Toutefois, si l'on prend en compte l'appétit sans cesse croissant de la puissance informatique

pour le calcul (c'est-à-dire pour résoudre plus rapidement des problèmes plus importants), il devient nécessaire de choisir une autre voie pour le calcul séquentiel. Le calcul parallèle semble être l'alternative la plus prometteuse si ce n'est la seule.

Le calcul parallèle se définit comme la pratique consistant à employer un nombre de processeurs associés, communiquant entre eux pour résoudre rapidement de gros problèmes. Il est rapidement devenu un domaine important de la science informatique. Au cours des cinq dernières années, le calcul parallèle a pris de l'ampleur et confirme que la plupart des recherches menées dans les domaines de la conception et de l'analyse d'algorithmes, de langages informatiques, d'applications informatiques et d'architectures informatiques se situent dans son contexte. De nouvelles machines parallèles à la nouvelle architecture sont construites chaque an.

Nous allons exploiter le parallélisme dans ce projet afin d'avoir les meilleurs résultats et temps d'exécution possibles d'un algorithme.

Dans ce rapport, le premier chapitre décrit le contexte de la réalisation, y compris l'objectif du logiciel, l'hypothèse et la méthodologie utilisée. Le deuxième chapitre présente les descriptions générales du logiciel sur l'environnement, les utilisateurs ainsi que la structure générale du système, ensuite nous allons exposer en détaille les fonctionnalités. Le troisième chapitre présente l'état de l'art relatif à ce projet. Le quatrième chapitre explique les analyses faites sur le problème choisi. Le chapitre 5 explique en détails l'algorithme proposé et des améliorations possibles. Quant au sixième chapitre, nous mettrons en place un bilan générale du projet.

La partie annexe traite les points suivants : planification, spécifications fonctionnelles et non-fonctionnelles détaillées, gestion de projet, documents d'installation et d'utilisation, enfin les différents tests à savoir : tests unitaires, fonctionnels et de performance.

2 Objectifs

Sur un problème NP-difficile, je dois mettre en oeuvre les différentes technologies du parallélisme : multithread (OpenMP), multicore (CUDA), multi-processus (MPI).

À l'aide de ces outils, je dois résoudre le problème de la séquence la plus courte en évaluant toutes les combinaisons possibles. La complexité est de $O(n!)$, le problème est NP-difficile.

Dans un premier temps, je dois mettre en place un algorithme séquentiel qui résout le problème, ensuite y affecter quelques modifications afin de le paralléliser.

Dans un deuxième temps, je dois adapter l'algorithme pour qu'il soit fonctionnel sur réseau. En effet, un processus "serveur" gère la file d'attente des décisions à évaluer et les distribues à plusieurs clients. En ce qui concerne la distribution ou l'évaluation des décisions, plusieurs stratégies peuvent être mises en place, comme :

- évaluation par demande
- mécanisme de reporting
- taille des lots
- LIFO ...

Lorsque les différents calculs sont attribués à différents clients, des fichiers de log et d'erreurs doivent exister pour la reprise de ces calculs en cas d'échec d'un client ou d'un serveur.

3 Hypothèses

1. Quant au projet, nous avons supposé que toutes les machines (Clients et serveur) sont disponibles et fonctionnelles.
2. Les différents programmes sont composés sur un ensemble de fichiers d'instance fixe
3. Le langage de programmation choisi est C, vu que les différentes bibliothèques et frameworks (**OpenMP**, **CUDA**, **MPI**, ...) fonctionnent correctement avec ce langage de programmation.

4 Bases méthodologiques

1. Les outils utilisés sont :
 - **Trello** : outil visuel et facile à utiliser pour gérer les différentes tâches du projet. Il est entièrement gratuit.
 - **ASTAH UML** : outil gratuit pour produire les différents diagrammes (use case diagram, séquentiel diagram, ...).
 - **Gantt Project** : outil utilisé pour ordonner et visualiser les diverses tâches dans le temps. Il permet de représenter graphiquement l'avancement du projet.
 - **Gitlab** : outil utilisé pour gérer les différentes versions de mon code.
 - **Gitlab-CI** : outil utilisé pour l'intégration continue. Cet outil est mis en place pour automatiser la compilation et l'exécution des différents tests.
2. Les étapes suivies pour la gestion de projet :
 - **Compréhension du projet** : Dans première phase d'étude, j'analyse le besoin du client et comprendre les différents points.
 - **Conception et planification** : Définir avec détails ce qui doit être fait, comment et avec quels moyens, et planifier dans le temps les étapes ainsi que les tâches.
 - **Réalisation du projet** : Mettre en oeuvre concrète des éléments planifiés, programmation ...
 - **Bilan final** : Rédiger un bilan de synthèse sur le déroulement du projet et les résultats attendus

Le modèle suivi pour la gestion de projet ainsi que la planification sera décrite dans la partie "**Planification**" dans l'Annexe.

2

Description générale

Dans ce chapitre, nous allons présenter l'environnement utilisé pour réaliser le projet, puis les caractéristiques des utilisateurs, ensuite les fonctionnalités du système et enfin la structure générale du système.

1 Environnement du projet

Ce projet dépend de l'environnement de développement ci-dessous :

- **Système d'exploitation** : Ubuntu 14.04 LTS
- **Langage de programmation** : C
- **API (Application Programming Interface)** : OpenMP, MPI, Cuda
- **IDE utilisé** : Clion pour la programmation de la partie en C. Pour les autres parties du projets, un éditeur de text sera utilisé pour la programmation. L'exécution est faite depuis le terminal, en exploitant un Makefile.
- **Caractéristiques de la machine de Développement** : 2,7 GHz Intel Core i5, 8 GB 1867 MHz DDR3

2 Caractéristiques des utilisateurs

Il n'y a qu'un type d'utilisateur pour l'utilisation du programme. Ses caractéristiques sont décrites dans le tableau [1](#).

	Connaissance de l'informatique	Expérience de l'application	Droits d'accès
Réponse	Oui	Oui	Tout droit
Commentaire	Savoir créer un fichier en suivant des règles, et avoir des connaissances en Linux, car pour lancer le programme on se base sur des lignes de commandes linux	Savoir utiliser un terminal pour lancer le programme	N'importe quel utilisateur peut lancer le programme en utilisant un terminal linux

Figure 1 – Tableau des caractéristiques des utilisateurs

3 Fonctionnalités du système

Le diagramme de cas d'utilisation (use case) est présenté dans la figure 2

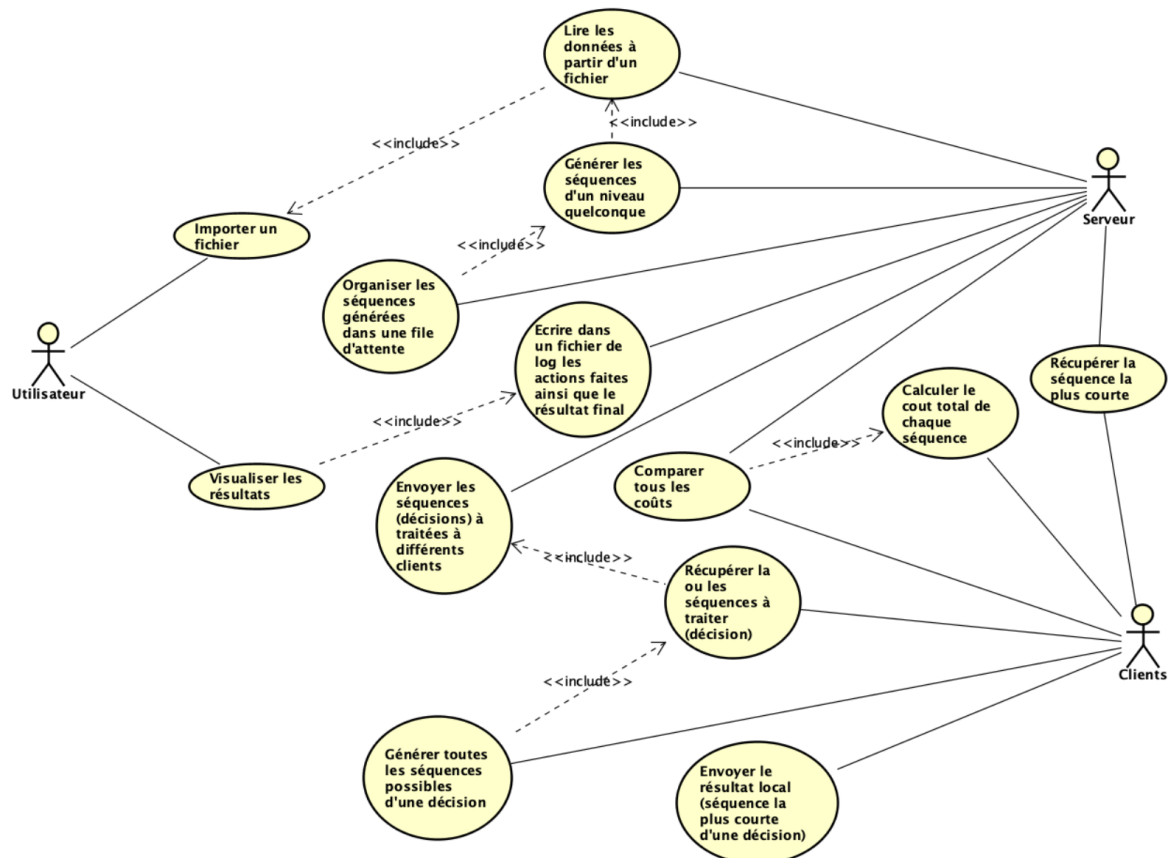


Figure 2 – Le diagramme de cas d'utilisation

Dans ce diagramme, trois rôles sont mis en place : l'utilisateur, le serveur et plusieurs clients. L'utilisateur peut importer un fichier contenant la matrice des distances, le serveur, quant à lui, récupère la matrice depuis ce fichier, génère des décisions (séquences temporaires) qu'il ajoutera dans une file d'attente, ensuite il envoie ces décisions à différents clients. Lorsqu'un client récupère la décisions à traiter, il génère toutes les séquences possibles de cette décision

puis renvoie la séquence la plus courte.

Le système est composé de deux grandes parties, un serveur qui traite les décisions et un client qui effectue des opérations sur ces décisions. L'interaction entre l'utilisateur, le serveur et les clients est présentée dans un diagramme de séquence (Figure 3)

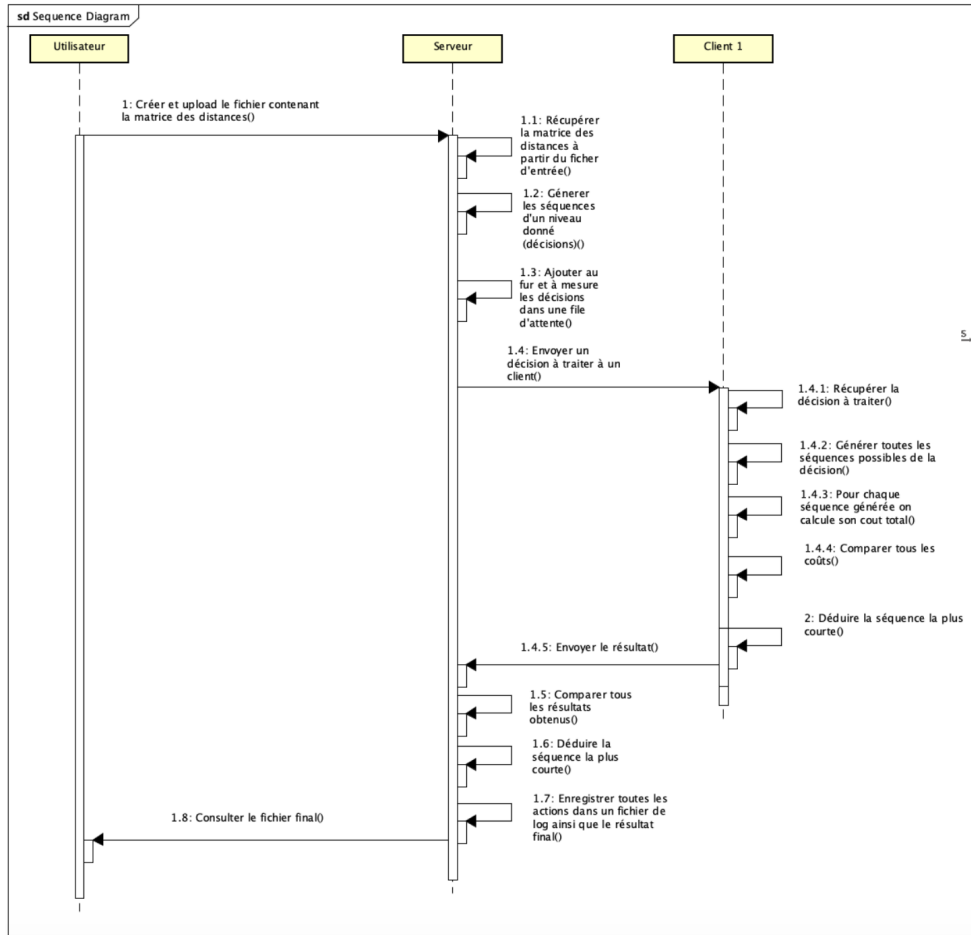


Figure 3 – Le diagramme de séquence

Dans le diagramme de séquence, le serveur récupère un fichier de données (contenant une matrice des distances), ensuite génère un ensemble de décisions (séquences temporaires) qui sont classés dans une file d'attente FIFO pour les envoyer par la suite à différents clients. Lorsqu'un client récupère une décision, il génère toutes les combinaisons possibles de cette décision puis calcule le coût de chaque combinaison générée (feuilles de l'arbre). la séquence la plus courte est ainsi envoyée au serveur, celui-ci enregistre les résultats envoyés par tous les clients pour ensuite les comparer et écrire dans un fichier de résultat la séquence la plus courte. Comme le diagramme suivant : Figure 3

4 Structure générale du système

Pour résoudre le problème de la séquence la plus courte en suivant l'architecture serveur / client, le système globale est découpé en plusieurs parties. Comme le montre la figure 4.

Le serveur récupère la matrice des distances depuis un fichier de données. Ensuite il génère des décisions qui seront stockées dans une file d'attente par la suite. Celle ci suit la stratégie FIFO

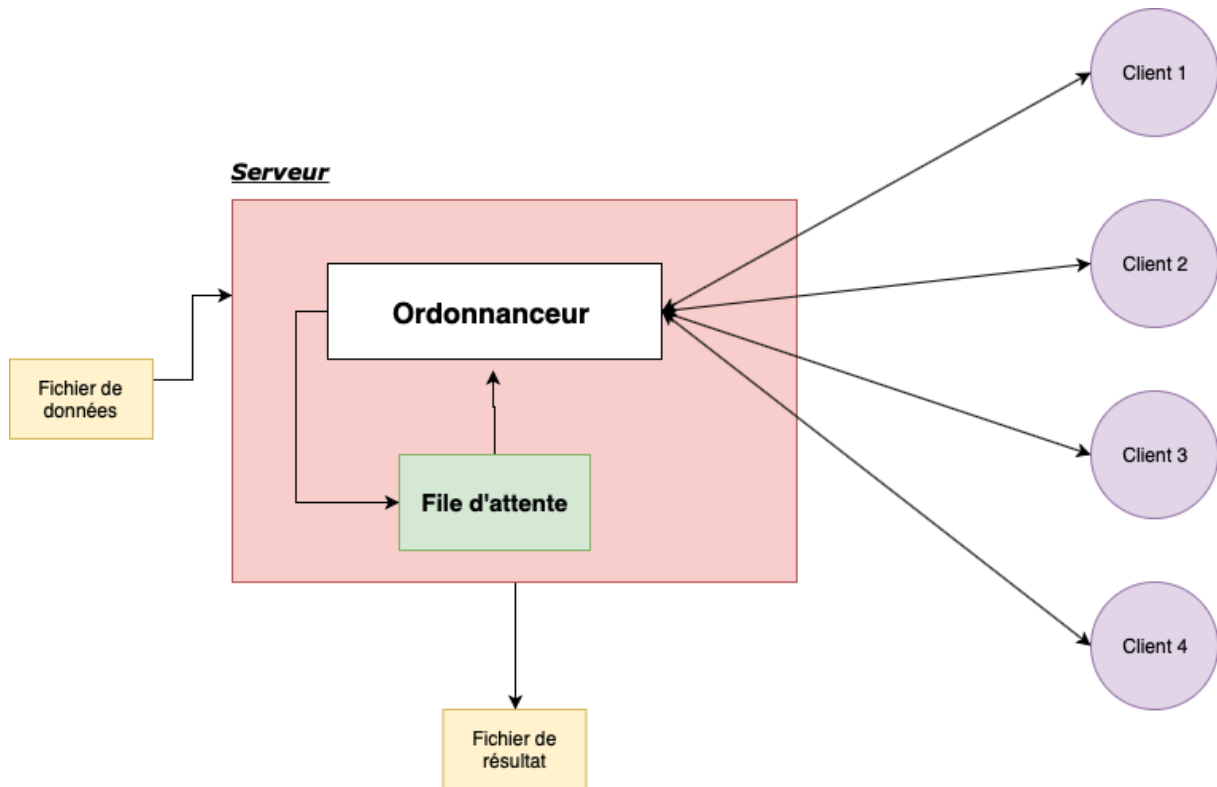


Figure 4 – Les différentes parties du système

(First In First Out). L'ordonnanceur, quant à lui envoie les décisions à traiter à plusieurs Clients.

Chaque client traite la décision, et génère toutes les séquences possibles puis en déduit la séquence la plus courte après avoir calculé son coût totale. Le résultat est ainsi envoyé au serveur. Tous les clients font les mêmes tâches mais sur des décisions différentes. Le serveur récupère tous les résultats obtenus par les clients puis en déduit la solution finale (la séquence la plus courte).

Cette architecture est valable pour tous les niveaux de parallélisme. La notion du Client qui change pour chaque technologie. Pour MPI par exemple, le client est un processus. Cependant pour CUDA, le client correspond à un thread sur le GPU.

3

État de l'art

1 Structure des Systemes de calcul parallèl

Les ordinateurs de hautes performances reliés à un réseau d'interconnexion ou même une grappe de calcul connectant plusieurs supercalculateurs par un réseau haut débit font partie de la majorité des systèmes de calcul d'aujourd'hui qui ont pour but, la résolution rapide des problèmes complexes. Pour arriver à réaliser cela, beaucoup d'efforts ont été déployés et par tous les moyens possibles pour accélérer l'exécution d'instruction au cœur même des processeurs. Ce qui, justifie les éléments de calcul d'aujourd'hui.

1.1 Amélioration des systèmes de calcul parallèl

Dans le cas général, la conception de la majorité des programmes informatiques pour la résolution des problèmes était par traitement d'un ensemble d'instructions en série d'un algorithme exécutées indépendamment, l'une après l'autre.

Cependant, dans le cas des traitements parallèles plusieurs processeurs sont utilisés simultanément pour trouver des solutions à un problème complexe donné. Celui ci est cassé en plusieurs parties indépendantes les unes des autres pour que chaque processeur puisse exécuter son bout de code en meme temps que les autres. Ici le terme processeur peut désigner plusieurs architectures telles qu'un seul ordinateur avec plusieurs microprocesseurs ou meme plusieurs ordinateurs en réseau.

Depuis les années 80, la croissance des processeurs avait pour but principal, l'augmentation de la fréquence. Certes, le nombre d'instructions multiplié par le temps moyen par instruction n'est autre que le bout d'exécution d'un bout de code.

L'équation de la consommation d'énergie d'une puce est donnée par :

$$P = C \times V^2 \times F$$

où :

P : puissance,

C : capacité commutée par cycle d'horloge,

V : tension du courant,

F : nombre des cycles d'un processeur par seconde.

Selon l'équation ci-dessus, la croissance de la fréquence du processeur est la cause de la croissance de l'énergie dissipée. L'évolution de la fréquence a pris fin vers la années 2000.

Il est indispensable de rappeler les lois de l'évolution des systèmes de traitement avant de d'entamer d'autres notions.

Loi de Moore

Les lois de Moore sont empiriques qui ont trait à l'évolution de la puissance de calcul des ordinateurs et de la complexité du matériel informatique. Les énoncés de Moore ne sont que des suppositions.

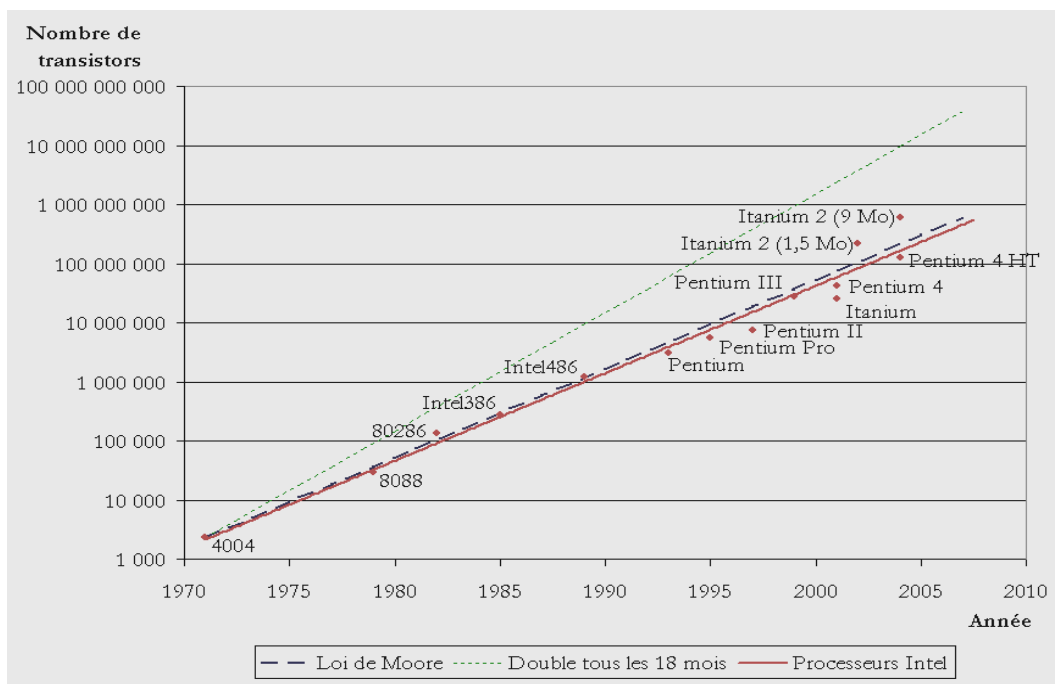


Figure 1 – Croissance du nombre de transistors dans les microprocesseurs Intel par rapport à la loi de Moore. En vert, la prédiction initiale voulant que ce nombre double tous les 18 mois

Afin que la loi de Moore reste valide, le recours au traitement parallèle est indispensable. Pendant des années, les fondeurs de processeurs étaient en mesure de fournir des augmentations de fréquence d'horloge et des améliorations de parallélisme au niveau des jeux d'instructions, pour que les applications n'utilisant qu'un seul thread s'exécutent beaucoup plus vite sur une nouvelle génération de processeurs sans aucun changement ou modification. Les fabricants des processeurs d'aujourd'hui préfèrent les architectures multicœurs, pour gérer la puissance dissipée à cause de l'horloge.

Les lois d'Amdahl et Gustafson

L'accélération de la parallélisation serait linéaire. En effet, ajouter deux fois le nombre d'éléments de traitement permettrait de diviser par deux le temps d'exécution, et l'ajouter une seconde fois devrait encore réduire le temps d'exécution de moitié. La majorité des algorithmes ont une vitesse d'accélération quasi-linéaire pour un petit nombre d'éléments de traitement, et

l'augmentation du nombre de ces éléments n'aura aucune valeur.

En 1966, La loi d'Amdahl a été annoncée, elle indique que l'accélération globale d'un programme est limitée par la partie séquentielle de celui-ci, car n'importe quel programme ne peut être parallélisé à 100%. En effet, un programme est consisté dans la majorité des cas des parties parallélisables et d'autres séquentielles. Voici l'équation de l'accélération maximale avec la parallélisation du programme :

$$\lim_{x \rightarrow 0} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

Si la partie séquentielle d'un programme atteint 10% du temps d'exécution ($\alpha = 0.1$), nous n'aurons qu'une accélération de 10 ×, quel que soit le nombre de processeurs ajoutés. La limite est atteinte, l'ajout des processeurs n'a aucun effet. L'ajout de plusieurs unités de traitement n'a aucune valeur ajoutée, quand un bout de code ne peut pas être divisé en raison de contraintes séquentielles. [1]

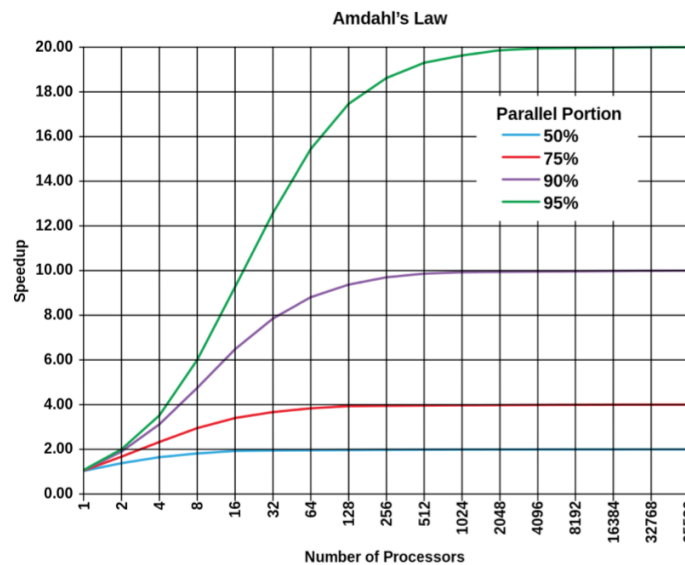


Figure 2 – Loi d'Amdahl

La Figure 2 représente le graphique de la loi d'Amdahl. L'accélération d'un programme par rapport à la parallélisations est limitée par le nombre de possibilités de parallélisations. Par exemple, si 90% du programme peut être parallélisé, quel que soit le nombre de processeurs utilisés, l'accélération maximale théorique en utilisant le calcul parallèle serait de 10 ×. [1]

La loi de Gustafson est une autre loi en informatique étroitement liée à la loi d'Amdahl. Il indique que l'accélération avec les processeurs P est :

$$S(P) = P - \alpha \times (P - 1) = \alpha + P \times (1 - \alpha)$$

On suppose qu'une tâche dans la figure 3 comporte deux parties indépendantes, A et B. B prend environ 25% du temps de l'ensemble du calcul. Avec un petit effort, un programmeur peut rendre cette partie cinq fois plus rapide, mais cela ne fait que réduire un peu le temps nécessaire au calcul. En revanche, il peut être nécessaire d'effectuer moins de travail pour rendre la partie A deux fois plus rapide. Cela rendra le calcul beaucoup plus rapide qu'avec l'optimisation de la partie B, même si B a eu une plus grande vitesse (5× contre 2×)

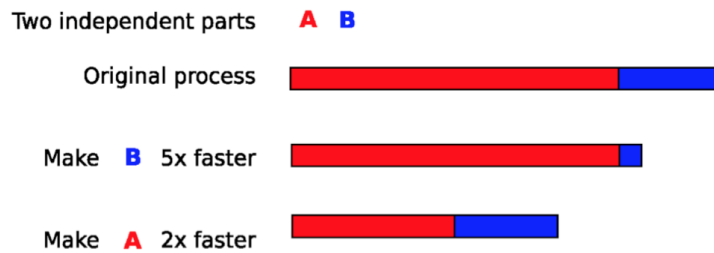


Figure 3 – Loi de Gustafson

La loi de Gustafson et la loi d'Amdahl supposent toutes les deux que la durée d'exécution de la partie séquentielle du programme est indépendante du nombre de processeurs. La loi de Gustafson suppose que la quantité totale de travail à effectuer en parallèle varie de manière linéaire avec le nombre de processeurs, tandis que la loi d'Amdahl suppose que l'ensemble du problème a une taille fixe, de sorte que la quantité totale de travail à effectuer en parallèle est également indépendante du nombre de processeurs.

Il était indispensable d'avoir une idée des trois lois (Moore, Amdahl et Gustafson), avant d'aborder l'évolution des systèmes parallèles qui ont poussé les technologies de se pencher vers les solutions parallèles.

1.1.1 Les Ordinateur hautes performances (HPC)

D'après Moore, chaque année le matériel de calcul haut performances évolue rapidement.

Les concepteurs des HPC (ordinateurs hautes performances) ont comme principal but le développement d'une machine qui a une puissance de calcul toujours plus importante que la génération de machines précédente afin de satisfaire les besoins croissants des applications scientifiques, c'est-à-dire toutes les tâches qui nécessitent une très forte puissance de calcul, comme les prévisions météorologiques

Un peu d'histoire

Dans les années 1961, les premiers HPC ont apparus.

Les ordinateurs les plus puissants du monde à un moment donné tendent à être égalés, puis dépassés, d'où le changement de la signification d'un HPC. Les premiers HPC étaient de simples ordinateurs monoprocesseurs mais possédant parfois jusqu'à dix processeurs périphériques pour les entrées-sorties, environ dix fois plus rapides que celui des concurrents. En 1970, la plupart des HPC adoptent un processeur vectoriel, qui effectue le décodage d'une instruction une seule fois pour l'appliquer à toute une série d'opérandes.

Vers la fin des années 1980, la technique des systèmes massivement parallèles est adoptée, avec l'utilisation dans un même HPC de milliers de processeurs. De nos jours, certains de ces HPC parallèles utilisent des microprocesseurs de type « RISC », conçus pour des ordinateurs de série, comme les PowerPC ou les PA-RISC. D'autres HPC utilisent des processeurs de moindre coût, de type « CISC », microprogrammes en RISC dans la puce électronique (AMD ou Intel) : le rendement en est un peu moins élevé, mais le canal d'accès à la mémoire, souvent un goulet d'étranglement, est bien moins sollicité.

Jusqu'au XXI^e siècle, les HPC sont souvent conçus comme des modèles uniques par des constructeurs informatiques.

Les composants des HPC

Les HPC sont plus performants que les ordinateurs ordinaires grâce à :

Premièrement, Leur architecture sous forme de « pipeline », c'est-à-dire l'exécution d'une instruction est faite sur une longue série de données, ou d'une manière parallèle en utilisant un nombre très élevé de processeurs fonctionnant chacun sur une partie du calcul simultanément, qui leur permet d'exécuter plusieurs tâches plus rapidement.

Ensuite, leurs composants électroniques très performants en utilisant souvent des structures de type serveurs lame contenant des processeurs multicœur ou des cartes graphiques dédiées au calcul scientifique de dernière génération, de la mémoire vive et des équipements de stockage de masse comme des disque dur SSD rapides qui sont reliés à la fibre optique et tout cela géré par un système d'exploitation dédié. Linux est généralement le logiciel le plus utilisé.

Afin d'exploiter au maximum sa puissance de calcul, l'architecture des mémoires au sein des HPC est analysée et plusieurs recherches sont faites la dessus. Les performances supérieures de la mémoire que ce soit en termes de composants ou même son architecture expliquent la supériorité des HPC sur les ordinateurs classiques en termes de performances. En effet, les HPC ont généralement des longues mémoires.

Quelques obstacles

D'une part, les HPC ont souvent besoin de plusieurs mégawatts de puissance électrique. Cette alimentation doit aussi être de qualité. En conséquence, ils produisent une grande quantité de chaleur et doivent donc être refroidis pour fonctionner normalement. Le refroidissement de ces ordinateurs pose souvent un problème important de climatisation.

D'une autre, les données ne peuvent pas circuler en dépassant la vitesse de la lumière entre deux parties d'un ordinateur. Le temps de latence entre certains composants n'est pas négligeable, quand la taille d'un HPC dépasse plusieurs mètres.

A nos jours, en très peu de temps, les HPC sont capables de traiter et d'envoyer de gros volumes de données. La puissance de calcul des processeurs serait sous exploitée, on aura en effet un goulot d'étranglement. Si, la conception doit assurer que ces données puissent être lues, transférées et stockées rapidement.

1.1.2 Les Clusters

Quant on parle de grappe de calcul, de cluster, ou de computer cluster en anglais, c'est pour désigner des techniques consistant à regrouper plusieurs ordinateurs indépendants appelés nœuds ou node en anglais, afin de permettre une gestion globale et de dépasser les limitations d'un ordinateur pour :

- Augmenter la disponibilité
- Faciliter la montée en charge
- Permettre une répartition de la charge

- Faciliter la gestion des ressources (processeur, mémoire vive, disques dur, bande passante réseau)

La création des grappes de calcul est beaucoup moins couteuse qu'un HPC. Les grappes de calcul regroupent plusieurs unités de traitements dans un réseau, mais qui fonctionnent comme un seul et unique énorme ordinateur avec plusieurs unités de calculs, d'espace de stockage, de GPU ...etc . Cette technique est utilisée pour résoudre le plus vite un problème de calcul complexe en dispatchant les tâches entre les nœuds du réseau.

Le principal atout de cette technologie est financier. L'utilisation d'une grappe de calcul est beaucoup moins coûteux qu'un HPC.

Le fonctionnement d'un cluster

Le terme cluster exprime l'idée de grappe. En outre, une grappe de calcul est un groupe de serveurs indépendants fonctionnant comme un seul et même système. Un client dialogue avec une grappe, comme s'il s'agissait d'une machine unique.

Dans les domaines scientifiques, les grappes sont habituellement constituées de nœuds de calcul/stockage et de un ou plusieurs nœuds frontaux. Parfois, on trouve des nœuds supplémentaires dédiés au monitoring.

Les nœuds peuvent être reliés entre eux par plusieurs réseaux. Par ailleurs, le réseau dont le débit est le plus lent est dédié aux tâches d'administration (chargement des systèmes sur les nœuds, suivi, mesure de charge...). À ce premier réseau vient généralement s'adjoindre un second réseau, avec une bande passante beaucoup plus importante. Des technologies de type Myrinet ou Infiniband peuvent être utilisées pour ce genre de réseau (Cf cours Mr SOUKHAL Ameer).

En fait, ces débits peuvent atteindre 40 gigabits par seconde.

Les programmes exécutés sur ce genre de machine se servent d'une API standard Message Passing Interface, utilisant la communication entre les divers processus répartis sur les nœuds avec des messages.

Lors de la défaillance d'un serveur, le logiciel de regroupement réagit en isolant le système défaillant. De même pour le partage des tâches d'un serveur surchargé avec un autre (dans le cas où les ressources sont partagées entre plusieurs tâches).

C'est une solution véritablement avantageuse en comparaison des HPC en raison du rapport performances/prix et des capacités d'extension avec des composants standards. L'ajout de nouveaux nœuds de calculs permet d'augmenter la puissance de calcul globale. La principale difficulté est de réussir à exploiter efficacement un tel système, là où les HPC disposent d'outils spécifiquement optimisés pour l'architecture. Malgré cela, les grappes se sont rapidement imposées dans le monde du calcul parallèle.^[4]

1.1.3 Grilles de calcul

Les grappes de calcul ont connu un véritable succès dans plusieurs domaines. Cette raison a poussé les constructeurs à franchir une nouvelle étape pour résoudre les problèmes les plus

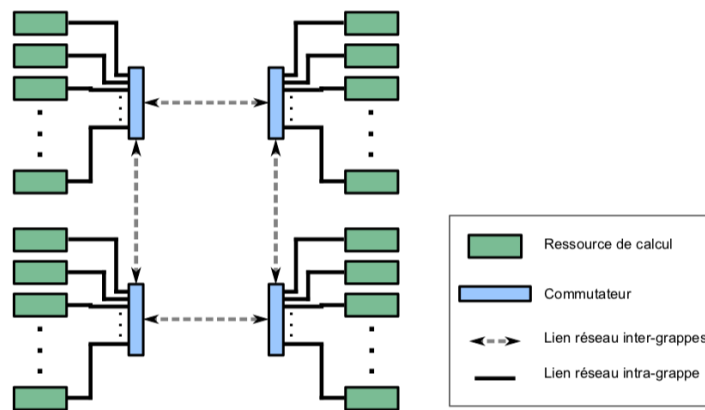


Figure 4 – Exemple d'Architecture d'une grille de calcul

complexes en peu de temps et avec des erreurs minables, n'importe quand et n'importe où. D'où l'apparition des grilles de calcul.

Cette technologie se base sur la liaison de plusieurs unités de traitements qui sont généralement dispatchés sur des distances pouvant atteindre des centaines de kilomètres voire plus. Les éléments liés peuvent être des simples ordinateurs, HPC, des grappes de calcul ou même tous ces composants mélangés. Le gros avantage de cette architecture est que de nouveaux composants peuvent être ajoutés à tout moment, certaines peuvent être retirées temporairement à cause d'une panne ou même pour toujours par exemple.

2 Programmation parallèle

Ce chapitre présente les 3 principaux modèles de programmation parallèle.

2.1 Les modèles de programmation parallèle

A l'utilisation de la parallélisation dans l'écriture du programme, il faut tout d'abord déterminer les blocs d'instructions qu'il est possible de rendre exécutables simultanément sur différents processeurs afin de découper les calculs complexes en fils d'exécution individuels. Après il faut choisir le modèle de programmation idoine, capable d'exprimer et d'exploiter le parallélisme de ce code. Ce choix est loin d'être évident, au vu du large panel de modèles existants. C'est pourquoi nous proposons ici de les classer en deux groupes correspondant à leur façon de gérer la mémoire entre ces différents fils d'exécution.^[4]

2.1.1 Mémoire partagée (paralléliser de données)

Un ensemble de fils d'exécution qui évoluent de façon concurrente dans un même espace d'adressage est la technique sur laquelle se base les modèles de programmation par mémoire partagée. A travers la mémoire de la machine, les communications (lectures/écritures) sont réalisées. Ainsi la tâche du programmeur est simplifiée en ce sens qu'il doit uniquement s'occuper du partage des tâches et potentiellement de la synchronisation en utilisant des verrous ou des sémaphores lors des accès à la mémoire partagée.

Ce type de modèle est clairement adapté à des machines de type SMP, où plusieurs processeurs vont partager une mémoire physique commune. Par ailleurs avec une sous-couche logicielle ou par un dispositif matériel particulier, il est possible d'employer ce modèle en faisant apparaître la mémoire physiquement distribuée comme une seule mémoire partagée. Cependant il est très difficile pour le programmeur d'obtenir des performances capables de concurrencer celles obtenues avec un modèle en mémoire distribuée, mieux adapté à de telles architectures.

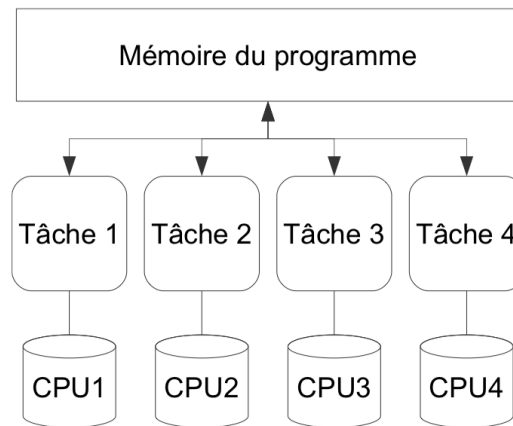


Figure 5 – Architecture d'un système à mémoire partagée

Faire appel à des processus légers ou threads est une des manières de paralléliser une exécution du programme. Ces processus sont appelés légers parce qu'ils partagent l'ensemble de leur espace mémoire, cette technique diminue les coûts de création ou de destruction associés à ces threads par rapport à des processus distincts, dits lourds.

Le partage des portions de mémoires entre processus indépendants est une autre manière de paralléliser une exécution, soit en recopiant les données d'un processus parent au moment de la création d'un processus fils (fork), soit en utilisant les primitives de mémoire partagée proposées par le système. Dans le domaine des services où un unique processus parent demeure en attente de traitements à confier à un ou plusieurs processus fils, cette technique est souvent utilisée. Ce fonctionnement est qualifié de maître-esclave.

La parallélisation en mémoire partagée n'implique pas nécessairement la création de plusieurs processus et peut également être réalisée au niveau de l'instruction. Dans ce cas, une même opération est appliquée à plusieurs données (SIMD) indiquées sous forme de vecteurs. Ces instructions sont pour cette raison dites vectorielles. Ce mode d'exécution est à la base de l'exécution sur GPU.^[3]

Le modèle de parallélisation le plus aisé à exploiter est la parallélisation en mémoire partagée, parce qu'il permet de préserver un seul espace mémoire pour toutes les tâches. Ce qui permet de faciliter l'adaptation d'un algorithme séquentiel avec un minimum de modifications, sans avoir recours à une répartition particulière des données. Il est nécessaire de gérer la cohérence des données mémoires puisque plusieurs tâches peuvent les modifier de manière simultanée et cela comme toute ressource partagée. Suivant le langage de programmation utilisé, cette synchronisation de l'accès aux données peut être intégrée au niveau des structures de données fournies (structure de données "thread-safe") ou être de la responsabilité du développeur.

L'obligation de conserver toutes les ressources sur une même machine rend difficile l'utilisation de ce type de parallélisation au-delà de quelques dizaines de cœurs et d'une centaine de gigaoc-

tets de mémoire vive avec des processeurs traditionnels. Ces limites correspondent de plus à des machines dédiées à ce type de parallélisation et sont donc en pratique beaucoup plus basses pour des machines de bureau ou des ordinateurs portables.[3]

L'utilisation de la parallélisation en mémoire partagée est grandement facilitée par des bibliothèques génériques telles que OpenMP, utilisé par de nombreuses simulations agents.[3]

2.1.2 Mémoire distribuée (paralléliser des tâches)

Le principe de cette seconde approche est basé sur la division de la mémoire totale du programme en plusieurs blocs de données et à les affecter aux fils d'exécution, chacun se chargeant d'effectuer ses calculs sur ses propres données de façon concurrente. Ce modèle connaît un succès grâce à la démocratisation des structures distribuées en lieu et place des machines massivement parallèles. Une des raisons majeures de ce changement est l'évolution des réseaux d'interconnexion.

Le point fort de cette technique est la réduction de la latence et l'augmentation de la bande passante d'une façon significative. En marge de cette évolution architecturale, les demandes toujours croissantes des applications, spécialement en termes de mémoire, ont favorisé l'utilisation de ce modèle car l'ensemble des processus offre un espace d'adressage plus grand permettant de traiter des données de taille plus importantes. Le modèle distribué est représenté par deux paradigmes majeurs : le passage de messages pour lequel le programmeur doit gérer explicitement les mouvements de données et les langages utilisant un espace d'adressage global partagé qui offrent un plus haut niveau d'abstraction en masquant les transferts liés à des accès distants.

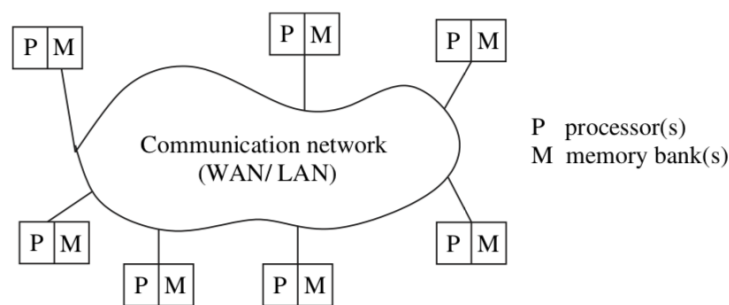


Figure 6 – *n* système distribué connecte les processeurs via un réseau de communication

La figure 6 représente un système distribué typique. Chaque ordinateur possède une unité de traitement de mémoire et les ordinateurs sont connectés par un réseau de communication. Les relations entre les composants logiciels exécutés sur chacun des ordinateurs et utilisant le système d'exploitation local et la pile de protocoles réseau pour fonctionner sont illustrées sur la figure 7. Le logiciel distribué porte le nom middleware. Une exécution distribuée est l'exécution de processus en utilisant le système distribué afin d'atteindre un objectif commun de manière collaborative. Une exécution est aussi parfois appelée calcul ou exécution.

Afin de décomposer la complexité de la conception du système, une architecture en couches est utilisée par le système distribué. Le middleware est le logiciel distribué qui pilote le système distribué, tout en offrant une transparence d'hétérogénéité au niveau de la plateforme.

Plusieurs primitives et appels à des fonctions définies dans diverses bibliothèques de la couche middleware sont incorporés dans le code du programme utilisateur. Il y a plusieurs bibliothèques qui servent à appeler des primitives pour les fonctions les plus utilisées.

Les versions commerciales de middleware actuellement déployées utilisent souvent les technologies CORBA, DCOM (modèle de composant distribué), Java et RMI (invocation de méthode à distance). L'**interface de passage de messages (MPI)** développée dans la communauté des chercheurs est un exemple d'interface pour diverses fonctions de communication.

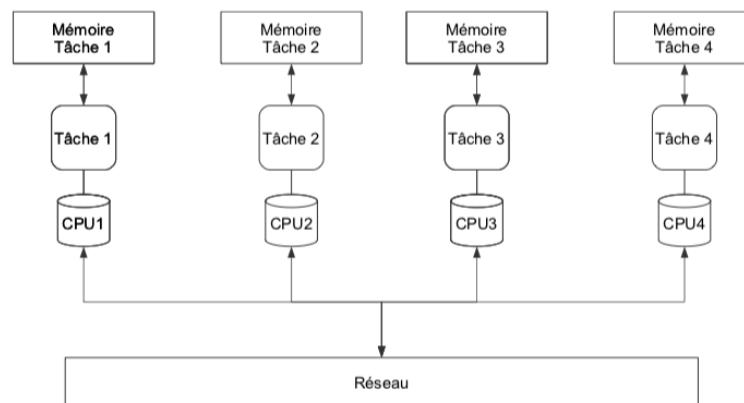


Figure 7 – Architecture d'un système distribué

Pourquoi utiliser les systèmes distribués ?

L'intérêt d'utiliser un système distribué réside dans la réponse aux exigences suivantes :

- **Augmentation du rapport performance / coût** Par partage des ressources et accès aux ressources géographiquement distantes, le rapport performance / coût est augmenté. Une telle configuration offre un meilleur rapport performance / coût que l'utilisation de machines parallèles spéciales.
- **Évolutivité** Etant donné que les processeurs sont généralement connectés à un réseau étendu, l'ajout de processeurs supplémentaires ne constitue pas un goulot d'étranglement direct pour le réseau de communication.
- **Fiabilité accrue** Un système distribué a le potentiel inhérent d'améliorer la fiabilité en raison de la possibilité de répliquer des ressources, ainsi que du fait que des ressources réparties géographiquement ne risquent pas de tomber en panne / de fonctionner en même temps dans des circonstances normales.
- **Accès aux données et aux ressources géographiquement distantes** Dans de nombreux scénarios, les données ne peuvent pas être répliquées sur tous les sites participant à l'exécution distribuée car elles peuvent être trop volumineuses ou trop sensibles pour être répliquées.
- **Calculs distribués de manière inhérente** Dans de nombreuses applications telles que le transfert de fonds dans le secteur bancaire ou la recherche d'un consensus entre des parties éloignées géographiquement, le calcul est distribué de manière inhérente.
- **Partage des ressources** Les ressources telles que les périphériques, les ensembles de données complets dans les bases de données, les bibliothèques spéciales, ainsi que les données (variables / fichiers) ne peuvent pas être entièrement répliquées sur tous les sites, car ils

ne sont souvent ni pratiques ni rentables. En outre, ils ne peuvent pas être placés sur un site unique, car l'accès à ce site peut s'avérer être un goulot d'étranglement.

2.1.3 Modèle hybride

L'apparition des parallélisations dites hybrides est favorisée par la récente popularisation des processeurs multi-coeurs et l'apparition de nouvelles solutions d'exécution comme les cartes graphiques. Ce modèle permet de mettre à contribution dans un même programme plusieurs modèles de programmation distincts comme OpenMP, MPI ou le CUDA.[3]

Il faut utiliser ces différentes méthodes d'une façon judicieuse afin d'exploiter l'ensemble des ressources présentes sur une même machine mais il faut toutefois prendre un certain nombre de précautions pour éviter tout conflit entre les modèles d'exécution. L'utilisation simultanée de MPI et d'OpenMP requiert en particulier une certaine vigilance pour éviter tout problème de cohérence de l'état des processus ou des données.

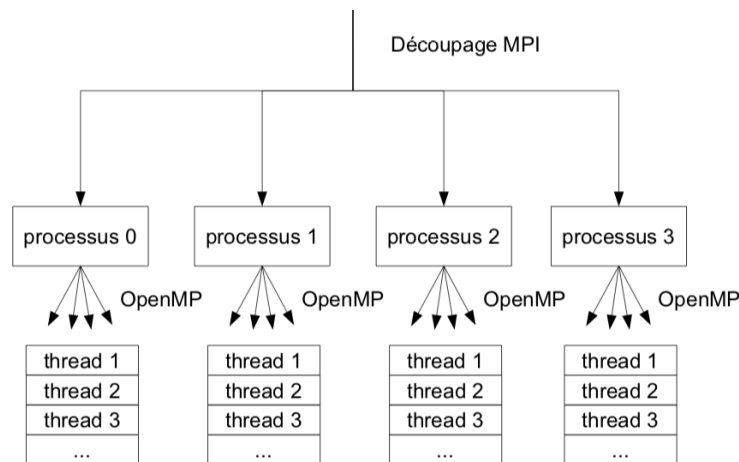


Figure 8 – Architecture Hybride (OPENMP + MPI)

L'utilisation des cartes Graphiques GPUs permet de profiter de leurs puissances en calcul. L'OpenACC est un des modèles de programmation qui nécessite l'utilisation d'un CPU et GPU simultanément pour exploiter au maximum possible l'architecture des processeurs traditionnelles et de cartes graphiques. Quasiment les mêmes syntaxes sont utilisées dans ce modèle.

Le modèle de programmation GPU

Le modèle de programmation GPU est caractérisé par l'utilisation de la plus large possible du découpage en threads en remplacement des boucles présentes dans l'algorithme. Cette démarche de parallélisation fine se justifie par les coûts d'exécution différents rencontrés sur CPU et sur GPU.

Un processeur traditionnel peut traiter un nombre limité de processus qui s'exécutent sur une longue durée à l'échelle du matériel : secondes, minutes, heures. La création et la destruction de processus est ainsi un traitement coûteux, car elle requiert l'allocation ou la libération d'un environnement mémoire et système complet. Les threads, ou processus légers, permettent de réaliser des traitements ponctuels en évitant cette allocation d'environnement, comme évoqué dans notre section sur la parallélisation en mémoire partagée.

Par contre, un GPU est conçu pour permettre à l'application d'exécuter un petit nombre d'opérations sur de grands volumes de données. Ce type d'exécution est caractérisé par des tâches brèves et remplacées très fréquemment de manière à assurer le remplissage des centaines de coeurs proposés par l'architecture. Les latences mémoires, importantes en regard du temps de traitement de chaque tâche, encourageant également la réallocation des ressources matérielles de calculs bloqués en attente d'opérations mémoire à d'autres traitements.

Dans ces circonstances, l'utilisation de nombres très importants de threads permet à l'ordonnanceur GPU de disposer d'un grand nombre de candidats pour optimiser le remplissage des ressources d'exécution fournies par la carte. Ces candidats sont regroupés en warp, ou paquet d'exécution, au moment de leur attribution à un multi-processeur matériel particulier.[3]

2.2 Les langages, API et frameworks

- **ACE (Adaptive Communication Environment)** Cet environnement de threads C++ est portable entre les plates-formes UNIX et Win32. Il intègre aux threads une gamme de mécanismes IPC tels que RPC, sockets et IPC System V. La principale force de ACE est son utilisation de nombreux modèles de conception de base pour les logiciels de communication simultanés. ACS fournit un ensemble complet d'encapsuleurs C++ réutilisables et de composants d'infrastructure qui exécutent les tâches courantes du logiciel de communication sur diverses plates-formes de système d'exploitation.
- **BSP (Bulk Synchronous Parallel model)** est un modèle de transition pour la conception d'algorithmes parallèles. Il a une fonction similaire à celle de la machine à accès aléatoire parallèle (PRAM). BSP diffère de PRAM en ne prenant pas la communication et la synchronisation pour acquis. Une partie importante de l'analyse d'un algorithme BSP repose sur la quantification de la synchronisation et de la communication nécessaires.
- **Calypso** Une implémentation du modèle BSP (bien que les auteurs ne le décrivent pas de cette façon). Un programme Calypso est un programme SPMD qui consiste en des processus maître et ouvrier répartis sur un réseau de postes de travail UNIX ou NT (OS homogène uniquement). Le processus maître s'occupe des opérations séquentielles et sert de serveur de tâches et de mémoire pour les processus des travailleurs. Les processus participent dynamiquement au calcul des sections parallèles. Le système offre un équilibre dynamique de la charge et un degré de tolérance aux pannes.
- **CC++ (Compositional C++, Caltech)** un petit ensemble d'extensions au C++ pour la programmation parallèle. CC++ fournit des constructions pour spécifier l'exécution simultanée, pour gérer la localité et pour la communication. Il permet de développer des programmes parallèles à partir de composants plus simples en utilisant une composition séquentielle, parallèle et simultanée. Ainsi, les algorithmes peuvent être traduits en programmes CC++ d'une manière simple et directe.
- **Charm++** est un langage de programmation parallèle orienté objet basé sur C++ et développé dans le Parallel Programming Laboratory de l'Université de l'Illinois à Urbana-Champaign. Charm++ a été conçu dans le but d'améliorer la productivité des programmeurs en fournissant une abstraction de haut niveau d'un programme parallèle tout en offrant de bonnes performances sur une grande variété de plateformes matérielles sous-jacentes. Les programmes écrits en Charm++ sont décomposés en un certain nombre

d'objets de messagerie coopérants appelés chares. Lorsqu'un programmeur invoque une méthode sur un objet, le système d'exécution Charm++ envoie un message à l'objet invoqué, qui peut résider sur le processeur local ou sur un processeur distant dans un calcul parallèle. Ce message déclenche l'exécution du code à l'intérieur de la chare pour traiter le message de manière asynchrone.

Les tâches peuvent être organisées en collections indexées appelées tableaux de chare et des messages peuvent être envoyés à des tâches individuelles dans un tableau de chare ou à l'ensemble du tableau de chare simultanément.

Les chares d'un programme sont mappés aux processeurs physiques par un système d'exécution adaptatif. Le mappage des chares aux processeurs est transparent pour le programmeur, et cette transparence permet au système d'exécution de modifier dynamiquement l'affectation des chares aux processeurs pendant l'exécution du programme pour prendre en charge des fonctionnalités telles que l'équilibrage de charge basé sur la mesure, la tolérance aux pannes, le pointage automatique et la possibilité de réduire et d'étendre l'ensemble des processeurs utilisés par un programme parallèle.

- **Chameleon** Une bibliothèque de transmission de messages . Chameleon est une interface stable de bas niveau pour les environnements de transmission de messages p4, PICL, PVM et spécifiques aux fournisseurs. Chameleon peut être utilisé dans un mode de développement où il fournit une grande variété d'informations de débogage ou dans un mode de production où il souligne l'efficacité parallèle.
- **CILK** Exécution efficace des calculs multithreads. Cilk est un langage algorithmique multithreading. La philosophie de Cilk est que les programmeurs doivent se concentrer sur la structuration de leurs programmes pour exposer le parallélisme et exploiter la localité, laissant au système d'exécution la responsabilité de planifier le calcul pour fonctionner efficacement sur une plate-forme donnée. Ainsi, le système d'exécution Cilk prend en charge des détails tels que l'équilibrage de charge, la téléavertissement et les protocoles de communication. Contrairement à d'autres langages multithreads, cependant, Cilk est algorithmique dans la mesure où le système d'exécution garantit des performances efficaces et prévisibles. Cilk fonctionne sur de nombreux systèmes SMP différents, y compris les PC sous Linux.
- **Concurrent C** étend le modèle CSP utilisé en Occam pour fournir un modèle de programmation simultanée plus général (et complexe). Concurrent C utilise un schéma synchrone de transmission de messages pour faciliter le parallélisme, les processus se déroulant indépendamment, sauf lorsque la communication a lieu. Le Concurrent C introduit un ensemble assez complexe de nouveaux opérateurs et de structures de programme pour prendre en charge la programmation parallèle asynchrone. Grâce à l'utilisation de pointeurs de transaction (une transaction est un mécanisme de communication bidirectionnelle structuré) et de variables de processus, les processus C simultanés peuvent communiquer directement avec d'autres processus, quelle que soit la position physique des processus dans le système. Le C simultané se mappe bien sur les machines à mémoire distribuée. (En fait, Concurrent C peut être implémenté sur un système de mémoire partagée, mais les constructions pour gérer la mémoire partagée ont été délibérément exclues du langage pour assurer la portabilité.)
- **GLU Granular Lucid** Système de programmation pour la construction d'applications parallèles et distribuées. Fonctionne sur un certain nombre de systèmes, y compris les PC sous Linux.
- **Haskell** Un langage de programmation fonctionnel . Haskell est un langage fonctionnel

d'ordre supérieur avec un riche système de type polymorphe et une sémantique non restrictive (c.-à-d. "évaluation paresseuse"). Son auteur décrit Haskell comme un langage para-fonctionnel pour faire comprendre qu'il s'agit d'une extension d'un langage purement fonctionnel et qu'il inclut des constructions pour représenter un parallélisme explicite. Il utilise un méta-langage pour distinguer la sémantique fonctionnelle (ce qui est calculé) de la sémantique opérationnelle (comment s'effectue le calcul).

"para-Haskell" supporte deux types d'annotations parallèles : les contraintes d'ordonnement et les expressions cartographiques. HPC+++ (High Performance C++) : Un modèle standard pour la programmation parallèle en C++. Un environnement de programmation du groupe de Dennis Gannon à l'Université de l'Indiana. Il combine son ancien système pC+++ et le système Caltech CC++. Il prend en charge une approche de base parallèle aux données, permettant ainsi une interaction compatible avec les directives de distribution de la HPF.

- **HPF High Performance Fortran** est un dialecte parallèle de données de Fortran 90. Des extensions ont récemment vu le jour pour soutenir le parallélisme au niveau des tâches, mais le cœur du langage et ses racines historiques se trouvent dans la programmation parallèle aux données. La plupart des HPF sont des directives et des constructions de langage pour partitionner et distribuer des tableaux entre les nœuds d'un ordinateur parallèle. La HPF n'a pas eu beaucoup de succès jusqu'à présent, car les algorithmes qui ne sont pas strictement parallèles aux données sont difficiles à mettre en œuvre avec la HPF.
- **Linda** La langue de coordination la plus connue est Linda. Chez Linda, la coordination s'effectue au moyen d'un petit ensemble d'opérations qui manipulent des objets dans une mémoire partagée distincte. La mémoire partagée supporte des algorithmes qui utilisent des constructions de haut niveau telles que les structures de données distribuées et la communication anonyme (c'est-à-dire que l'expéditeur et le destinataire ne connaissent pas l'identité de l'autre). Les fournisseurs commerciaux de Linda (Scientific Computing Associates) fournissent les versions Fortran et C du système. Voir aussi JADA, WWWinda, ISETL-Linda, ParLin, Eilean, P4-Linda, Glenda, POSYBL et Objective-Linda.
- **JADA** est un système de type Linda qui mélange Linda avec Java. Les tuples multiples sont supportés. Ceux-ci peuvent être locaux (pour la coordination entre les threads) ou distants (pour la coordination entre des applets distincts potentiellement distribués sur le WWW). La JADA a été créée dans le cadre du PageSpace (un projet financé par ESPRIT).
- **Lucide** Un langage fonctionnel parallèle basé sur la logique intentionnelle. Lucide est un langage implicitement parallèle. Lucid permet aux structures de données telles que les tableaux, les listes et les arbres d'être implémentés d'une manière facilement distribuable. Lucid est simple et élégant. Lucid n'est engagé dans aucun modèle particulier de calcul, de sorte que l'auteur d'un compilateur Lucid dispose d'une liberté considérable pour implémenter des fonctionnalités de langage d'une manière qui ne peut être interféré avec le programme utilisateur.
- **MPI Message passing interface** est une norme conçue en 1993-94, qui définit une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseur par passage de messages. Elle est devenue de facto un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. MPI a été écrite pour obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Elle est disponible sur de très nombreux matériels et systèmes d'ex-

exploitation. Ainsi, MPI possède l'avantage par rapport aux plus vieilles bibliothèques de passage de messages d'être grandement portable (car MPI a été implémentée sur presque toutes les architectures de mémoires) et rapide (car chaque implémentation a été optimisée pour le matériel sur lequel il s'exécute).

- **NESL** est un langage parallèle développé à Carnegie Mellon par le projet SCandAL. Il intègre diverses idées de la communauté théorique (algorithmes parallèles), de la communauté langagière (langages fonctionnels) et de la communauté du système (plusieurs des techniques de mise en œuvre). Les nouvelles idées les plus importantes derrière NESL sont les suivantes :
 1. **Parallélisme des données imbriquées** : cette fonctionnalité offre les avantages du parallélisme des données, un code concis, facile à comprendre et à déboguer, tout en étant bien adapté aux algorithmes irréguliers, tels que les algorithmes sur arbres, graphiques ou matrices clairsemées.
 2. **Un modèle de performance basé sur le langage** : cela donne un moyen formel de calculer le travail et la profondeur d'un programme. Ces mesures peuvent être liées au temps de fonctionnement sur des machines parallèles.

Le but principal dans la conception de NESL était de rendre la programmation parallèle facile et portable. Les algorithmes sont généralement beaucoup plus concis dans NESL que dans la plupart des autres langages de programmation parallèles. De plus, le code ressemble beaucoup à un pseudocode de haut niveau.

- **Occam** est l'un des premiers langages créés explicitement pour le calcul parallèle. Un programme Occam est un ensemble de processus qui sont composés séquentiellement ou en parallèle. Les processus interagissent par des canaux de communication explicites.
- **PAMS** Un environnement de programmation commercialement supporté de Myrias. PAMS est un système basé sur un compilateur. Le programme identifie les boucles parallèles et utilise des directives pour les faire exécuter en parallèle.
- **PVM Parallel Virtual Machine** Un système logiciel qui permet d'utiliser en parallèle un ensemble d'ordinateurs hétérogènes. Il inclut des bibliothèques de fonctions appelables par l'utilisateur et un programme démon qui coordonne l'activité inter-machine. Fonctionne sans : Ordinateurs hétérogènes. Langues : C et Fortran.
- **Sisal** est un langage de programmation fonctionnel. Il a été largement utilisé dans les environnements à mémoire partagée. Il y a eu du travail pour le déplacer vers des environnements à mémoire distribuée, mais ce travail n'a pas conduit à une implémentation robuste de la mémoire distribuée. Sisal extrait le parallélisme d'un programme à l'aide d'une analyse de dépendance aux données. Le langage n'a pas de constructions parallèles explicites. Sisal garantit des résultats reproductibles dans un environnement multiprocesseur.
- **Sthreads** Un environnement de programmation basé sur des threads de Caltech. Le système se compose d'un pragma et d'un ensemble de primitives de synchronisation. Si utilisé selon certaines règles étroitement définies, un programme Sthreads est garanti pour s'exécuter et produire le même résultat dans les modes séquentiels et multi-threads. La bibliothèque sous-jacente utilisée pour implémenter les pragmas est également fournie avec Sthreads et rendue visible pour l'utilisateur.

- **Strand** Strand est un langage parallèle basé sur la programmation logique simultanée. Il est très similaire au Parlog concurrent plat. Une discussion de son utilisation dans la programmation scientifique peut être trouvée dans [1]. Le langage a été développé pour la distribution commerciale, mais il est actuellement disponible gratuitement.
- **WinPar** Un environnement basé sur la transmission de messages (MPI et PVM) pour prendre en charge le calcul parallèle sur les stations de travail Intel Architecture. NT est la plate-forme de choix pour WINPAR. WINPAR est un environnement de développement logiciel intégré pour le calcul parallèle ciblant les ordinateurs personnels interconnectés par des réseaux locaux sous Windows NT.

Les objectifs techniques de WINPAR sont de fournir une couche de transmission de messages incluant MPI et PVM, de fournir un cadre de fonctionnalités de base nécessaires au calcul parallèle et de fournir un ensemble d'outils pour le développement de code, la simulation, la prévision des performances, le débogage graphique de haut niveau, la surveillance et la visualisation des applications parallèles. Les objectifs commerciaux de WINPAR sont d'offrir un environnement de développement parallèle abordable pour la formation et l'éducation dans les universités, les organismes de recherche et l'industrie, d'être conforme aux normes existantes sur le marché HPCN, et donc d'étendre ce marché actuellement uniquement UNIX et MPP aux ordinateurs Windows NT en réseau.

- **OpenMP (Open Multi-Processing)** est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée. Cette API est prise en charge par de nombreuses plateformes, incluant GNU/Linux, OS X et Windows, pour les langages de programmation C, C++ et Fortran. Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

OpenMP est portable et dimensionnable. Il permet de développer rapidement des applications parallèles à petite granularité en restant proche du code séquentiel.

- **CUDA** (initialement l'acronyme de Compute Unified Device Architecture) est une technologie de GPGPU (General-Purpose Computing on Graphics Processing Units), c'est-à-dire utilisant un processeur graphique (GPU) pour exécuter des calculs généraux à la place du processeur (CPU). En effet, ces processeurs comportent couramment de l'ordre d'un millier de circuits de calcul fonctionnant typiquement à 1 GHz, ce qui représente un potentiel très supérieur à un processeur central à 4 GHz, fût-il multicœurs et multi-threads, si et seulement si le calcul à effectuer est parallélisable.

CUDA permet de programmer des GPU en C. Elle a été développée par Nvidia pour ses 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 le 15 février 2007.

L'exploitation des ressources GPU ne nécessite pas une connaissance des concepts ou des modèles de programmation GPU : de nombreuses bibliothèques de traitement vectoriel ou matriciel traditionnellement utilisées sur CPU déjà existantes pour cette architecture. Ces bibliothèques fournissent souvent une interface de programmation similaire à leur équivalent CPU, de manière à faciliter leur utilisation dans un programme parallélisé existant.

Ci-après quelques exemples de bibliothèques de ce type basées sur CUDA :

1. **cuBLAS** est une implémentation du standard d'algèbre linéaire BLAS. Certaines opérations deviennent ainsi 6x à 17x plus rapides que leur équivalent CPU. Cette bibliothèque fait partie des bibliothèques optimisées GPU fournies par la société NVIDIA.
 2. **cuFFT** pour CUDA Fast Fourier Transform library, permet le calcul de transformées rapides de Fourier sur GPU. Cette bibliothèque est également fournie par la société NVIDIA.
 3. **CUSP** (C++ Templated Sparse Matrix Library) est une bibliothèque d'algèbre linéaire à faible densité. Elle permet également la manipulation et le traitement de graphes. Son utilisation repose sur le mécanisme des templates C++ pour permettre la génération de code GPU parallélisé en fonction des traitements demandés par l'utilisateur.
 4. **cuSparse** est une bibliothèque de traitements matriciels fournie par NVIDIA. Les formats de représentations de matrice creuses les plus courants (COO, CSR, CSC, ELL/HYB) et leur manipulation sont gérés de manière native en CUDA.
- **OpenCL** (Open Computing Language) est la combinaison d'une API et d'un langage de programmation dérivé du C, proposé comme un standard ouvert par le Khronos Group. OpenCL est conçu pour programmer des systèmes parallèles hétérogènes comprenant par exemple à la fois un CPU multi-cœur et un GPU. OpenCL propose donc un modèle de programmation se situant à l'intersection naissante entre le monde des CPU et des GPU, les premiers étant de plus en plus parallèles, les seconds étant de plus en plus programmables.

Il existe également des alternatives basées sur OpenCL :

1. **clMath (anciennement AMD APPML)** permet de recouvrir à la fois les opérations proposées par BLAS et le traitement des transformées de Fourier. cette bibliothèque est souvent utilisée avec celle de **clMAGMA**, qui fournit de nombreux solveurs linéaires et solutions de factorisation, réduction ou transformation de matrices.
2. **clpp** est un autre projet qui fournit des primitives de traitement en parallèle de structures de données. Ces traitements incluent notamment la recherche par préfixe ("scan"), le tri, ou la réduction de valeurs, de manière à faciliter la parallélisation de traitements plus complexes sur des structures telles que des graphes ou des arbres.
3. **VexCL** facilite également le traitement de matrices et de vecteurs en OpenCL. Cette bibliothèque est plus particulièrement orientée vers la réduction de la quantité de code nécessaire à la préparation et à la gestion des traitements sur GPU, au moyen de l'architecture objet C++.

2.3 Les avantages du modèle hybride

Pourquoi mélanger les modèles de programmation ?

Les standards MPI, OpenMP et CUDA particulièrement et les modèles de programmation existant en général font l'objet de réflexions régulières qui visent à améliorer leur interface et la

façon de les implémenter.

Pour diminuer la granularité ainsi que d'exploiter au maximum chaque ressource la combinaison de deux modèles est indispensable. En effet, on cherche à offrir un mélange du parallélisme à gros grain du paradigme distribué comme le passage de messages avec le parallélisme à grain fin offert par une approche partagée. Le premier permet de paralléliser sur l'ensemble des nœuds et le second à l'intérieur de chaque nœud.

Bénéficier des avantages des deux types de modèles

La combinaison de deux modèles de programmation permet de tirer profit des performances de chaque modèle et de remédier aux problèmes rencontrés sur certaines architectures ou dans des conditions particulières. Ainsi, l'idée est de remédier les limites du passage à l'échelle des implémentations du standard MPI face à un trop grand nombre de processus sur des nœuds composés d'un grand nombre de cœurs. En utilisant un modèle mixte, le nombre de processus MPI diminue jusqu'à une borne inférieure correspondant au nombre de nœuds de calcul (un processus par nœud).

Actuellement la tendance est la multiplication des cœurs dans un même nœud de calcul, cette programmation hybride réduit le trafic réseau (notion de **Granularité**), particulièrement important lors des communications collectives, et par là-même diminue les risques de contention.

Gérer finement l'empreinte mémoire

Une des grandes difficultés est la gestion de l'espace mémoire lors de l'écriture d'un programme. Les développeurs d'applications parallèles tentent d'en optimiser autant que possible l'occupation, vu que cet espace est limité.

D'une part l'emploi d'un modèle hybride de programmation tel MPI + OpenMP permet d'atteindre ce résultat sans nécessiter de structures de données supplémentaires ou de mise en place de mécanismes d'interception. Ainsi en générant un processus MPI par nœud de calcul multicœurs, les données d'un même nœud sont partagées entre les threads OpenMP plutôt que d'être dupliquées dans les espaces d'adressage de plusieurs processus.

D'autre part, la réduction du nombre de processus par nœud qu'implique la combinaison de deux modèles de programmation réduit fortement la quantité de mémoire allouée pour les structures de données utilisées par l'implémentation MPI. Les communications collectives irrégulières sont alors plus efficaces.

3 Conclusion

D'après l'étude approfondie des différentes technologies ainsi que les avantages de chacune nous allons utiliser MPI pour la programmation distribuée, OpenMP pour la programmation partagée et enfin CUDA pour l'intégration de la puissance du GPU. En mettant en place ces trois technologies nous sommes sûrs d'exploiter nos ressources au maximum, ainsi que les trois différents niveaux de parallélisme.

Ci-dessous un résumé des trois outils qui seront utilisés pendant le développement des différentes parties de notre projet :

OpenMP

OpenMP est un modèle de programmation parallèle qui au début ciblait uniquement les architectures à mémoire partagée. Aujourd'hui, il cible aussi les accélérateurs, les systèmes embarqués et les systèmes à temps réel. Les tâches de calcul peuvent accéder à un espace de mémoire commun, ce qui limite la redondance des données et simplifie les échanges d'information entre les tâches. En pratique, le parallélisme repose sur l'utilisation de processus système légers ou threads.

À son lancement, un programme OpenMP est séquentiel. Il est constitué d'un processus unique, le thread maître dont le rang vaut 0, qui exécute la tâche implicite initiale. Ainsi OpenMP permet de définir des régions parallèles, c'est-à-dire des portions de code destinées à être exécutées en parallèle. Au début d'une région parallèle, de nouveaux processus légers sont créés ainsi que les nouvelles tâches implicites, chaque thread exécutant sa tâche implicite, en vue de se partager le travail et de s'exécuter concurremment. Donc un programme OpenMP est une alternance de régions séquentielles et de régions parallèles.

Chaque processus léger (thread) exécute sa propre séquence d'instructions, qui correspond à sa tâche implicite. C'est le système d'exploitation qui choisit l'ordre d'exécution des processus, à chaque fois il affecte aux unités de calcul disponibles (cœurs de processeur). Ainsi aucune garantie sur l'ordre global d'exécution des instructions n'est accordée. voir figure 9. [2]

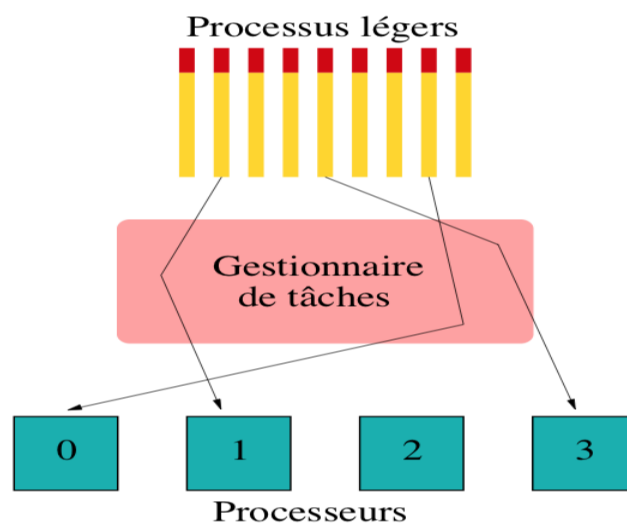


Figure 9 – Threads et leur affectation par l'OS

Les tâches d'un même programme partagent l'espace mémoire de la tâche initiale (mémoire partagée) mais disposent aussi d'un espace mémoire local : la pile (ou stack). Il est ainsi possible de définir des variables partagées (stockées dans la mémoire partagée) ou des variables privées (stockées dans la pile de chacune des tâches). Comme présenté dans la figure 10

En mémoire partagée, il est parfois nécessaire d'introduire une synchronisation entre les tâches concurrentes car elle permet d'éviter que deux threads ne modifient dans un ordre quelconque la valeur d'une même variable partagée

MPI

MPI est une norme qui définit un ensemble de fonctions de communication entre processus locaux ou distants. Des implémentations pour les langages C, C++ et Fortran sont disponibles sur

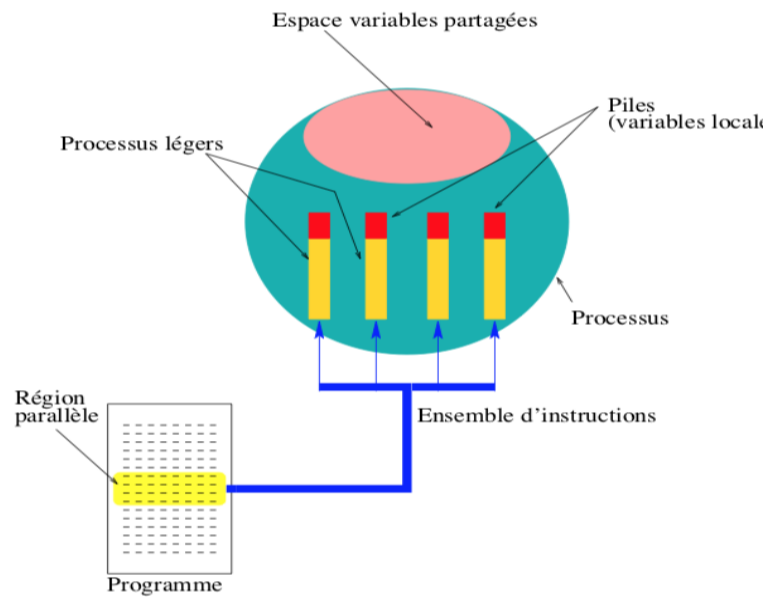


Figure 10 – Variables privées et partagée

de nombreuses plates-formes, ce qui en fait un standard pour la réalisation des parallélisations distribuées.

Ces fonctions permettent d'avoir de bonnes performances d'exécution aussi bien sur une même machine qu'entre des machines distantes. MPI repose pour cela sur un ensemble de primitives de communication de haut niveau susceptibles d'exploiter les mécanismes optimisés d'échange de données offerts par le système d'exploitation et le matériel.

Une exécution MPI est à base d'un ensemble de processus associés à des numéros de rang indépendants de leur localisation physique (Figure 12). Ces numéros de rang permettent à chaque processus d'adapter ses traitements en fonction de son rôle dans le groupe, en se comportant par exemple en maître distribuant des tâches ou en esclave traitant ces calculs.^[2]

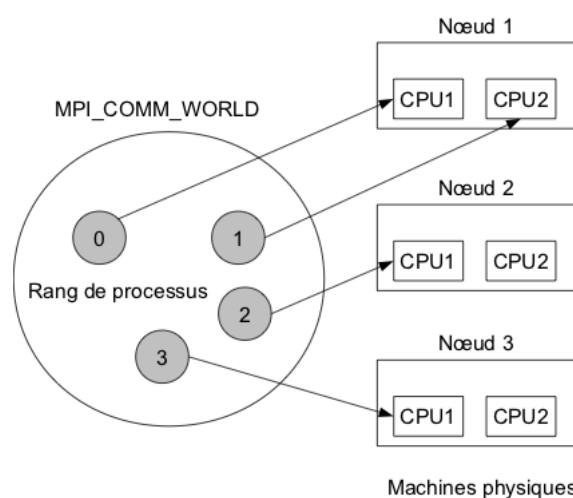


Figure 11 – Exemple d'association entre processus MPI et matériel physique

Deux catégories d'opérations de communications sont proposées par MPI :

- **Les opérations point-à-point** : ces communications mettent en jeu un unique émetteur et

destinataire dans le groupe de processus.

- **Les opérations de groupe**, également qualifiées de collectives ou de multicast, impliquent la participation d'une partie ou de tous les processus MPI pour réaliser un même traitement. Un exemple de tel traitement commun est la diffusion d'une donnée en début de calcul, ou la mise en commun de résultats partiels à la fin de l'exécution MPI.

La majorité de ces opérations de communication possèdent des variantes synchrones et asynchrones, de manière à faciliter la gestion du déroulement de l'exécution ou la poursuite de traitements en tâche de fond dans l'attente de communications.

Les implémentations libres les plus connues du standard MPI sont **MPICH** et **OpenMPI**.

CUDA

La carte graphique "GPU" ou "device" est utilisée comme coprocesseur de calcul pour le processeur de la machine hôte, le PC typiquement host ou CPU. La mémoire du CPU est distincte de celle du GPU mais on peut faire des copies de l'un vers l'autre mais c'est généralement coûteux. Les fonctions calculées sur le device est appelée kernel. Celui-ci est dupliqué sur le GPU comme un ensemble de threads qui est organisé de façon logique en Grid. Celle-ci est mappée physiquement sur l'architecture de la carte au runtime. Chaque clone du kernel connaît sa position dans la grid et peut calculer la fonction définie par le kernel sur différentes données.

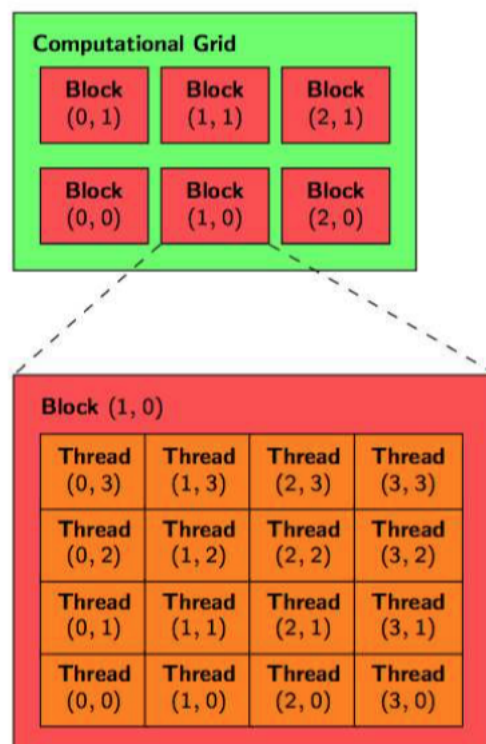


Figure 12 – Exemple d'une Grid

Une Grid est un tableau 1D, 2D ou 3D de **Threads blocks**. Chaque thread block est un tableau 1D, 2D ou 3D de threads, chacun exécutant un clone (instance) du kernel. Et chaque block a un unique BlockId, pareil pour les threads, chacun a son unique ThreadId dans un block donné.[2]

Quant à l'exécution d'un programme CPU, on réalise des transferts sur le GPU depuis le CPU : exécution de kernels. Afin d'être le plus efficace possible il faut absolument limiter les transferts de données. On peut exécuter les kernels en mode bloquant (synchrone) ou non bloquant (asynchrone) vis à vis du programme CPU . La figure 13 explique comment un programme est exécuté.

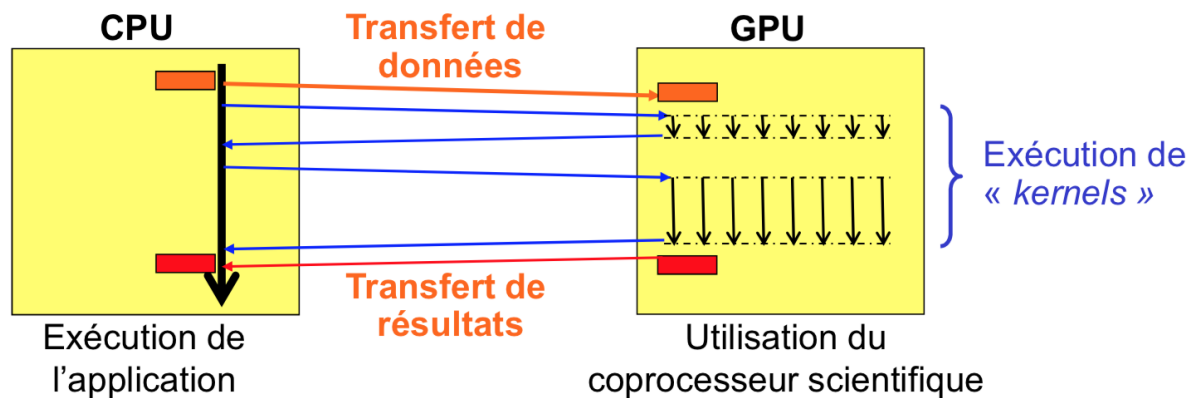


Figure 13 – Exécution d'un programme CPU - GPU

4

Analyse et conception

Dans ce chapitre, on analyse le problème de la séquence la plus courte afin de proposer une solution séquentielle et qu'on pourra rendre parallélisable.

1 Problème de la séquence la plus courte

1.1 Problème NP-difficile

Avant de proposer une solution au problème de la séquence la plus courte il est indispensable de connaître ce qu'est un problème NP-difficile.

un problème P est NP-difficile lorsque chaque problème L dans NP peut être réduit en temps polynomial à P , c'est-à-dire, en supposant qu'une solution pour P prend 1 unité de temps, nous pouvons utiliser la solution de P pour résoudre L en temps polynomial.

En conséquence, trouver un algorithme polynomial pour résoudre un problème NP-difficile donnerait des algorithmes polynomiaux pour tous les problèmes dans NP, qui est peu probable car beaucoup de ceux-ci sont difficiles.

1.2 Explication du problème de la séquence la plus courte

Le problème de la séquence la plus courte est l'un des problèmes les plus étudiés en mathématiques numériques.

Imaginons nous avons plusieurs points, et entre chaque point et les autres points nous avons une valeur qui correspond à la distance entre ces deux points. Le défi est de trouver le chemin le plus court pour visiter chaque point une seule fois et sans revenir au point de départ.¹

Le problème de la séquence la plus courte s'agit d'un problème NP-difficile, on ne peut pas le résoudre en temps polynomial.

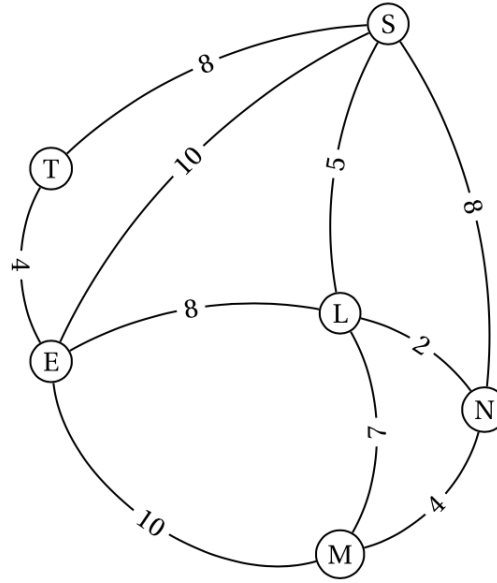


Figure 1 – Exemple des points de départ

1.3 Modélisation mathématique

Le problème de la séquence la plus courte peut être formulé comme un programme linéaire entier. Plusieurs modélisations mathématiques sont connues. Les plus notables sont la formulation Miller-Tucker-Zemlin (MTZ) et la formulation Dantzig-Fulkerson-Johnson (DFJ). La formulation DFJ est plus forte, bien que la formulation MTZ soit encore utile dans certains contextes.

1.3.1 Modélisation de Miller-Tucker-Zemlin

Supposons que nous avons des points numérotés de 1 à n .

$$x_{ij} = \begin{cases} 1 & \text{s'il existe un chemin entre le point } i \text{ et } j \\ 0 & \text{sinon.} \end{cases}$$

Soit u_i une variable fictive pour $i \in 1, \dots, n$, et soit c_{ij} la distance entre le point i et le point j . Ainsi le problème de la séquence la plus courte peut être modélisé comme suit :

$$\min \sum_{i=1}^n \sum_{j \neq i, j=1}^n c_{ij} \cdot x_{ij} \text{ tel que : } \forall i, j \in (1, \dots, n) 0 \leq x_{ij} \leq 1$$

$$\sum_{j \neq i, i=1}^n x_{ij} = x_{ij} \text{ tel que : } \forall j \in (1, \dots, n)$$

$$\sum_{j \neq i, j=1}^n x_{ij} = x_{ij} \text{ tel que : } \forall i \in (1, \dots, n)$$

$$u_i - u_j + n \cdot x_{ij} \leq n - 1 \text{ tel que : } 2 \leq i \neq j \leq n$$

$$0 \leq u_i \leq n - 1 \text{ tel que : } 2 \leq i \leq n$$

La première série d'égalités exige que chaque point provient d'un seule autre point, et la deuxième série d'égalités exige qu'il y ait un départ de chaque point vers un seule autre point. Les dernières contraintes imposent qu'il n'y ait qu'une seule tournée couvrant tous les points, et non deux ou plusieurs tournées disjointes qui ne couvrent que collectivement tous les points. Pour prouver ceci, il est montré ci-dessous que chaque solution réalisable ne contient qu'une seule séquence fermée de villes, et que pour chaque tour unique couvrant toutes les villes, il existe des valeurs pour les variables muettes u_i qui satisfont ces contraintes.

Pour prouver que chaque solution réalisable ne contient qu'une seule séquence de points, il suffit de montrer que chaque sous tour d'une solution réalisable passe par le point 1 (en notant que les égalités assurent qu'il ne peut y avoir qu'un seul tour). Car si l'on additionne toutes les inégalités correspondant à $x_{ij} = 1$ pour tout sous tour de k ne passant pas par la ville 1, on obtient :

$$n.k \leq (n-1).k$$

qui est bien sûr absurde.

Il faut maintenant montrer que pour chaque tournée couvrant tous les points, il existe des valeurs pour les variables muettes u_i qui satisfont les contraintes.

On définit la tournée comme commençant (et se terminant) à la ville 1. Choisir $u_i = t$ si la ville i est visitée à l'étape t alors :

$$u_i - u_j \leq n - 1$$

vu que u_i ne peut ni être supérieur à n ni inférieur à 1, les contraintes sont satisfaites même si $x_{ij} = 0$. Pour $x_{ij} = 1$ non a :

$$u_i - u_j + n.x_{ij} = (t) - (t+1) + n = n - 1$$

qui satisfait la contrainte.

1.3.2 Modélisation de Dantzig-Fulkerson-Johnson

Supposons que nous avons des points numérotés de 1 à n .

$$x_{ij} = \begin{cases} 1 & \text{s'il existe un chemin entre le point } i \text{ et } j \\ 0 & \text{sinon.} \end{cases}$$

Soit c_{ij} la distance entre le point i et le point j . Ainsi le problème de la séquence la plus courte peut être modélisé comme suit :

$$\min \sum_{i=1}^n \sum_{j \neq i, j=1}^n c_{ij}.x_{ij} \text{ tel que : } \forall i, j \in (1, \dots, n) 0 \leq x_{ij} \leq 1$$

$$\sum_{j \neq i, i=1}^n x_{ij} = x_{ij} \text{ tel que : } \forall j \in (1, \dots, n)$$

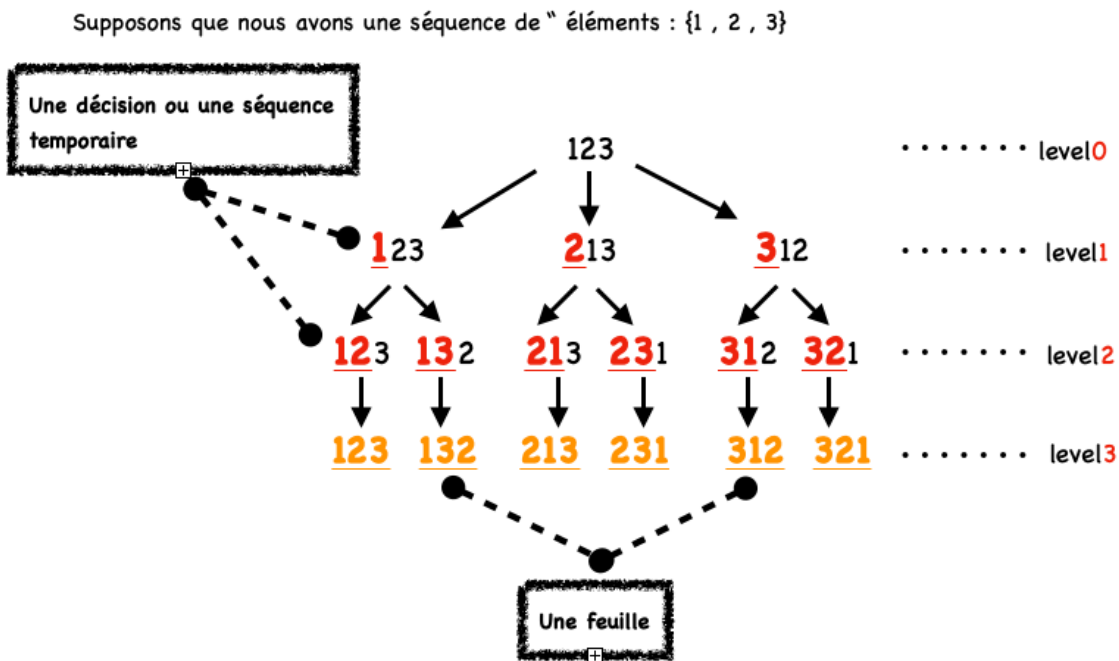
$$\sum_{j \neq i, j=1}^n = x_{ij} \text{ tel que : } \forall i \in (1, \dots, n)$$

$$\sum_{i=1}^n \sum_{j \neq i, j=1}^n x_{ij} \leq |Q| - 1 \text{ tel que : } \forall Q \subseteq \{2, \dots, n\}$$

La dernière contrainte de la formulation du DFJ garantit qu'il n'y a pas de sous tours parmi les sommets qui ne démarrent pas, donc la solution retournée est une seule tournée et non l'union de petites tournées.

1.4 Conception de l'algorithme de résolution

Pour résoudre le problème de la séquence la plus courte, avec mon tuteur, nous nous sommes mis d'accord sur la résolution par brute force. C'est-à-dire, générer toutes les séquences possibles pour ensuite calculer pour chaque séquence son coût afin de les comparer à la fin. La séquence la plus courte correspond à celle qui a le plus petit coût.



Voici quelques exemples des décisions (séquences temporaires) et leurs feuilles.

Une décision : **1**23

- séquence : {1, 2, 3}
- nombre d'éléments fixés : 1

Les feuilles de cette décision :

{1, 2, 3} & {1, 3, 2}

Une décision : **2**13

- séquence : {2, 1, 3}
- nombre d'éléments fixés : 2

Les feuilles de cette décision :

{2, 1, 3}

Figure 2 – L'arbre des décisions

La figure (2) représente l'algorithme que j'ai proposé et implémenté pour résoudre le problème. Cet algorithme a été bien réfléchi pour que l'on puisse le paralléliser facilement.

Dans la figure (2), supposons que nous avons une séquence de points : $\{1, 2, 3\}$. Pour chaque niveau on génère des décisions. Une décision est une séquence pour laquelle on a fixé un ou plusieurs éléments. Si tous les éléments de la décision sont fixés on obtient ainsi les feuilles de l'arbre. Toutes les feuilles correspondent à toutes les séquences possibles des éléments $\{1, 2, 3\}$. Dans la figure si nous prenons par exemple le niveau 1 et la décision $\{2, 1, 3\}$ on fixe le deuxième élément puis on fait un swap donc les éléments qu'on peut fixer sont 2 et 3. ainsi on obtient 2 nouvelles décisions : $\{2, 1, 3\}$ et $\{2, 3, 1\}$ avec les deux éléments sont fixés.

À la fin on remarque que toutes les combinaisons possibles de la séquence $1, 2, 3$ correspondent aux feuilles de l'arbre.

5

Mise en oeuvre

Dans cette partie je vais parler des librairies utilisées, le choix des versions, les différents types de risques rencontrés la réalisation de ce PRD, les choix techniques et enfin les différents résultats obtenus ainsi que leur explication.

1 Les librairies choisies

1.1 MPI

Pour la programmation "multiprocessus", j'ai choisi l'API OpenMPI que j'ai présenté dans la partie "**État d'art : Les langages, API et frameworks**". Quant à la version utilisée est la dernière version, à savoir **4.0.1**. Cette version est choisie car elle est la dernière version stable.

J'ai expliqué en détail, dans le manuel de configuration, les étapes nécessaires pour l'installation de cette API.

1.2 OpenMP

Pour la programmation "multithread", j'ai choisi l'API OpenMP que j'ai présenté aussi dans la partie "**État d'art : Les langages, API et frameworks**". En ce qui concerne à la version utilisée, j'ai choisi la version **4.0** qui n'est pas la dernière version. En effet, la dernière version est **5.0** qui est toujours en mode de production. Donc, pour éviter des problèmes dans les différentes fonction, j'ai préféré d'employer la version la plus stable.

La configuration de cette API est détaillé dans le cahier de configuration.

1.3 CUDA

Pour la programmation "GPU", j'ai choisi d'utiliser le Toolkit de NVIDIA CUDA. C'est une boîte à outils fournit par NVIDIA et qui fournit un environnement de développement pour créer des applications hautes performances accélérées par le GPU. Cette boîte à outils comprend des bibliothèques vraiment accélérées par le GPU, des outils de débogage et d'optimisation, un compilateur C et une bibliothèques d'exécution pour déployer notre application. J'ai choisi la

dernière version **10.1** qui est stable.

La configuration de l'environnement de travail est expliqué en détail dans le manuel de configuration.

1.4 Langage de programmation

Le langage choisi est le C. Ce langage de programmation a été choisi car il supporte les 3 technologies citées ci-dessus.

2 Framework CUNIT

CUnit est un système pour écrire, administrer et exécuter des tests unitaires en C. Il est construit comme une bibliothèque statique qui est liée au code de test de l'utilisateur.

CUnit utilise un cadre simple pour construire des structures de test et fournit un ensemble riche d'assertions pour tester des types de données communs.

En outre, plusieurs interfaces différentes sont fournies pour l'exécution des tests et la communication des résultats. Il s'agit notamment d'interfaces automatisées pour les tests et les rapports contrôlés par code, ainsi que d'interfaces interactives permettant à l'utilisateur d'exécuter des tests et de visualiser les résultats de manière dynamique.

Pour effectuer des tests unitaires et fonctionnels j'ai utilisé la dernière version stable (2.1-3) qui est sortie le 21-04-2014.

Ce framework est le plus populaire dans la communauté des développeurs en C.

3 Intégration continue (GITLAB-CI)

L'intégration continue (CI), permet d'intégrer le code source dans un dépôt Git. Lorsque je partage le nouveau code, une pipeline est déclenchée pour construire, tester et valider le nouveau code avant de fusionner les modifications dans le dépôt GIT.

Plusieurs étapes ont été automatisées, comme le BUILD et l'exécution des différents tests. En effet, lorsque je fais un PUSH sur gitlab, une image docker est créée puis lancée. Dans celle-ci j'installe toutes les dépendances nécessaires pour l'exécution et le build des différentes parties du projet.

Voici un exemple des différents pipelines créés :

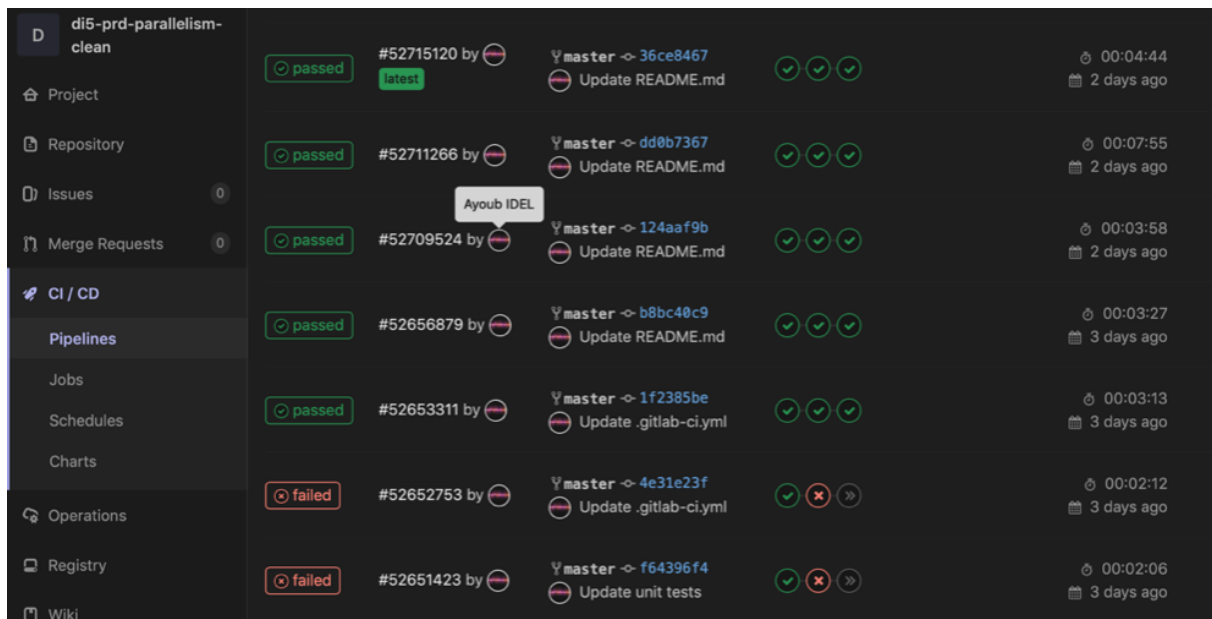


Figure 1 – Les pipelines

Ci-dessous les différents "stages" :

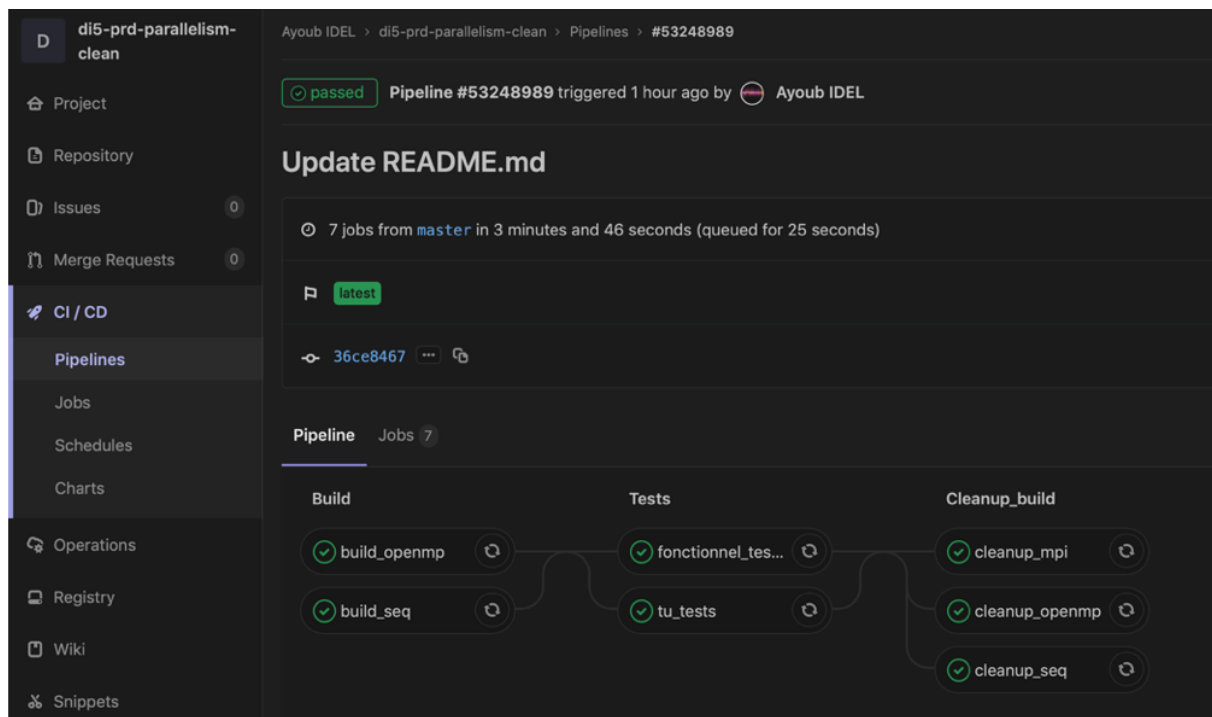


Figure 2 – Les pipelines

4 Versioning du code (GITLAB)

Tout au long de la phase développement j'ai utilisé plusieurs outils pour gérer les différentes version du projet. Quant au dépôt, j'ai un repo GITLAB. J'ai utilisé plusieurs branches pour le développement des différents projets comme présenté dans la prise d'écran ci-dessous.

Le logiciel GITKRAKEN a été utilisé pour mieux exploiter le dépôt Git.

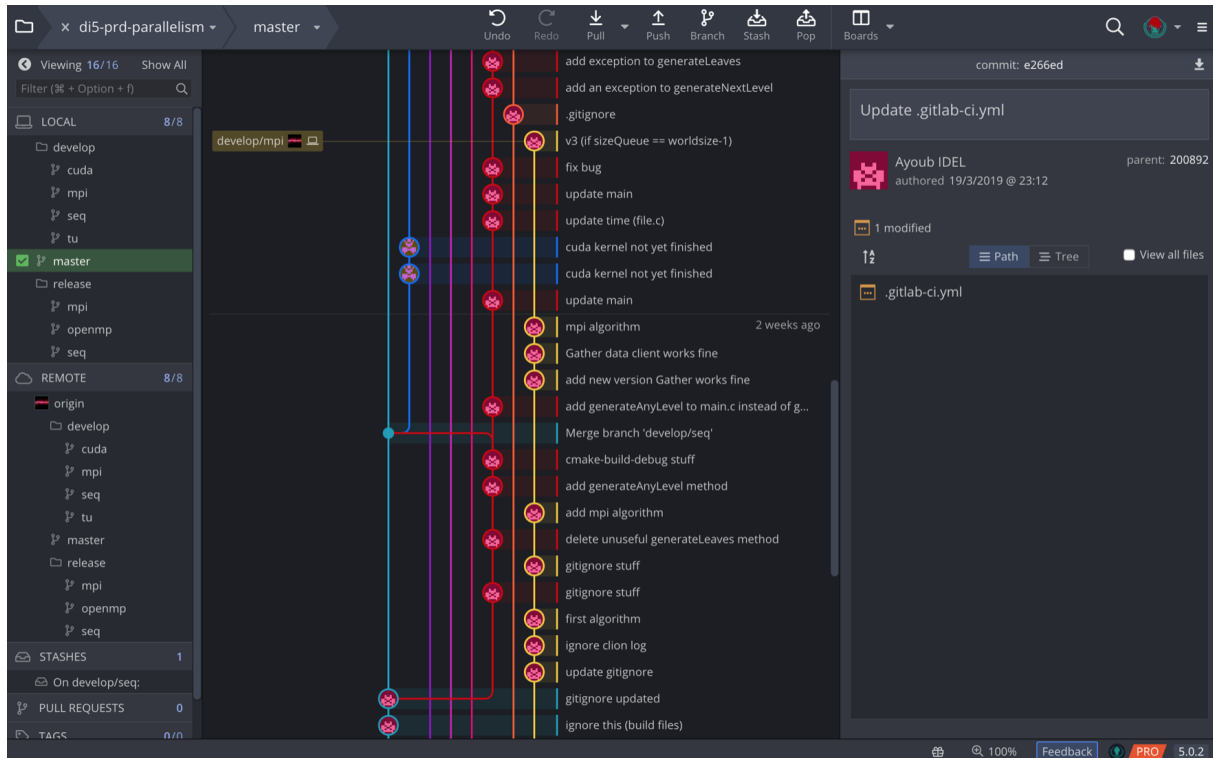


Figure 3 – Les pipelines

5 Revue du code (INDENT)

Le programme d'indentation peut être utilisé pour rendre le code plus facile à lire. Il peut également convertir d'un style d'écriture C à un autre.

indent comprend une grande partie de la syntaxe de C, mais il tente également de faire face à une syntaxe incomplète et mal formée.

Dans la version 1.2 et les versions plus récentes, le style d'indentation GNU est le style par défaut.

6 Documentation DOXYGEN

Doxygen permet de créer des documentations techniques pour notamment le C. Pour toutes les différentes parties de mon projet, la documentation DOXYGEN a été générée afin de faciliter la compréhension de l'implémentation de mon programme.

Shortest Sequence

Main Page
Data Structures ▾
Files ▾

code_source
seq

client_utils.c File Reference
Functions

Methods used in Client side. More...

```
#include "client_utils.h"
```

Include dependency graph for client_utils.c:

Functions

void	getArrayOfDataClients (MATRIX *all_leaves, MATRIX *mtx_distances, DATA_CLIENT **my_array)	Method that puts in a specific array of all DATA_CLIENTs with specific elements, distances and total cost after calculating them. More...
DATA_CLIENT *	getMinimumOfDataClients (DATA_CLIENT **all_data_clients, int num_elements, int SIZE)	Gives the minimum of DATA_CLIENT in a DATA_CLIENTS array (lowest cost). More...
int	deleteDataClient (DATA_CLIENT *data_client)	Deletes a specific instance of data_client. More...
void	createDataClientArray (DATA_CLIENT **my_array, int size)	Creates an array of data_clients. More...
void	addDataClient2Array (DATA_CLIENT **my_array, DATA_CLIENT *data_client, int SIZE, int pos)	Adds a specific data_client to the array of data_clients. More...
void	printDataClient (DATA_CLIENT *data_client, int size)	Prints the instance of data_client. More...

Detailed Description

Methods used in Client side.

Author
Ayoub IDEL

Version
1.1

Date
4 March 2019

Defines all fuctions used and executed in Client side.

Figure 4 – Exemple d'une documentation DOXYGEN

7 Identification et gestion des risques

Une analyse des risques a été faite pour identifier les différents risques qui peuvent influencer le déroulement de mon PRD. Cette étude m'a permis de mieux visualiser les raisons pour lesquels les objectifs du projet ne seront pas atteints. Cela m'a permis de bien gérer le projet afin de contourner les retards. La méthodologie suivi pour la gestion de projet est la méthode en cascade.

Pour l'analyse des risques j'ai suivi les étapes suivantes :

- Identification des différents types de risque.
- Analyser l'impact de chaque risque sur les objectifs finaux de mon projet, en termes de délais et de qualité.
- Mettre en place une stratégie pour contourner les risques étudiés.

8 risques humains

Pour cette partie j'ai analysé deux risques humains, à savoir : Maladie ou accident de travail. Ces risques peuvent entraîner des retards flagrants sur le déroulement de mon PRD.

9 risques sur les délais (deadline)

Ce risque est attaché à l'estimation faite au début sur les différentes tâches à accomplir. En effet, une tâche mal planifiée peut entraîner des dépassement au niveau des délais. En plus ma stratégie (méthode en cascade) est vraiment sensible. En effet, un petit retard sur une tâche peut entraîner des retards énorme à la fin. Donc pour éviter ce genre de problème pour chaque tâche j'ai laissé une marge au cas où un retard survient.

10 risques techniques

Les risques techniques que j'ai listé sont l'utilisation des différentes bibliothèques ainsi que la configuration de celles ci dans l'environnement Ubuntu. Il faut noter que j'ai intégré dans mon planning des créneaux pour mieux approfondir mes connaissances sur les technologies utilisées (MPI, OpenMP et CUDA).

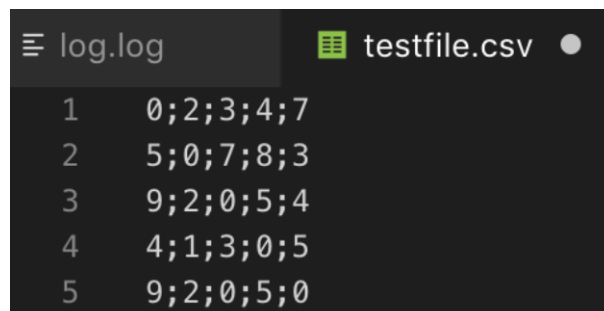
11 Résultats

Cette section présente d'abord le fichier d'entrée et de de sortie, ensuite une explication des résultats obtenus après les tests de performances.

11.1 Fichier entrée/sortie

Mon programme débute l'exécution on récupérant un fichier d'entrée CSV qui contient la matrice des distances, et en sortie un fichier de LOG est créé. Dans celui ci on stocke toutes les opérations faite par le serveur ainsi que le résultats final.

Voici un exemple simple d'un fichier d'entrée :



```

log.log
1 0;2;3;4;7
2 5;0;7;8;3
3 9;2;0;5;4
4 4;1;3;0;5
5 9;2;0;5;0
  
```

Figure 5 – Exemple d'un fichier d'entrée

Quant à la sortie on obtient le fichier LOG suivant :

Dans ce fichier on précise la date ainsi que l'heure de l'exécution de l'action, puis son type : (INFO) pour dire que c'est un résultat ou (ERROR) pour dire qu'une erreur est survenue.

```

8705508 [2019-03-24 14:41:59][INFO] : All leaves of actual node
8705509 3 2 1 4
8705510 3 2 4 1
8705511 3 1 2 4
8705512 3 1 4 2
8705513 3 4 1 2
8705514 3 4 2 1
8705515 [2019-03-24 14:41:59][INFO] : local min DATA_CLIENT
8705516 Elements : [ 3 2 1 4 ]
8705517 Distances : [ 2 5 4 ]
8705518 Total cost : 11
8705519 [2019-03-24 14:41:59][INFO] : This is node number : 3
8705520 [2019-03-24 14:41:59][INFO] : All leaves of actual node
8705521 4 2 3 1
8705522 4 2 1 3
8705523 4 3 2 1
8705524 4 3 1 2
8705525 4 1 3 2
8705526 4 1 2 3
8705527 [2019-03-24 14:41:59][INFO] : local min DATA_CLIENT
8705528 Elements : [ 4 2 1 3 ]
8705529 Distances : [ 1 5 3 ]
8705530 Total cost : 9
8705531 [2019-03-24 14:41:59][INFO] : global min DATA_CLIENT
8705532 Elements : [ 1 3 4 2 ]
8705533 Distances : [ 3 5 1 ]
8705534 Total cost : 9

```

Figure 6 – Exemple d'un fichier de sortie

11.2 Performances

J'ai fait une comparaison des résultats obtenus après les tests de performances. Pour chaque partie de mon projet, je note le temps d'exécution en fonction de la taille de séquence de départ.

Explications :

On remarque que la courbe qui correspond à l'exécution séquentielle du projet est quasiment superposée à la courbe de l'exécution OpenMP. Ceux-ci est dû à l'utilisation de 2 core dans la machine virtuelle et aux régions critiques exécutées les thread. En effet, les threads doivent attendre un moment pour la synchronisation.

Lorsque la taille de la séquence est égale à 7, l'exécution sur le GPU prend est supérieure à celle de la partie séquentielle. Cela est dû à la non exploitation suffisante du GPU où on utilisé que 7 threads pour traiter les différentes décisions. En plus, la communication entre le GPU et le CPU

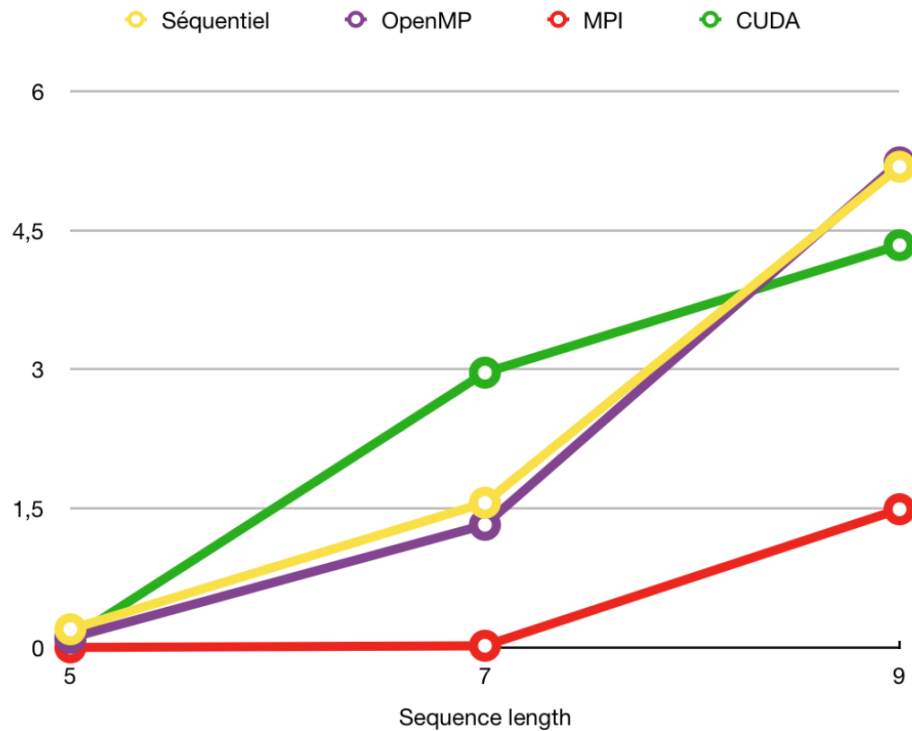


Figure 7 – Temps d'exécution en fonction de la longueur de la séquence

est vraiment couteuse, d'où cette différence de temps d'exécution. En revanche, lorsque la taille de la séquence est égale à 9, on remarque que le temps d'exécution de la partie CUDA est plus petit que l'exécution séquentielle, car on commence à mieux exploiter le GPU.

Quant à l'exécution MPI, on obtient le meilleur temps d'exécution possible. Quelque soit la taille de la séquence l'exécution sur MPI est toujours petite à celle de la partie séquentielle. Cela s'explique par le fait qu'on est sur une seule machine, donc les communication sont faites aussitôt. En effet, si on était sur un vrai réseau où les machines sont connectées par des câbles ethernet, on aurait un peu de latence. En plus chaque processus traite sa décision en parallèle que les autres processus.

6

Conclusion

1 Ce qui a été fait

Les tâches faites sur ce projet sont :

- L'étude de faisabilité des différentes parties du projet (Partie séquentielle, Partie parallèle OpenMP, Partie parallèle MPI, Partie parallèle CUDA).
- Analyse du problème de la séquence la plus courte.
- Etude des différentes technologies mises en place dans chaque partie (Etude approfondie des 3 technologies MPI, CUDA et OpenMP).
- Mise en place d'un algorithme séquentiel (et parallélisable) pour la résolution du problème de la séquence la plus courte. L'algorithme permet de générer le niveau suivant d'une décision.
- L'amélioration de l'algorithme proposé (generateNextLevel). Deux autres algorithmes améliorés ont été fait (GenerateAnyLevel et GenerateLeaves), le premier algorithme permet de générer n'importe quel niveau d'une décision, et le deuxième permet de générer directement le dernier niveau..
- La résolution séquentielle complète du problème de la séquence la plus courte en suivant l'architecture Client/Serveur.
- La résolution parallèle complète du problème de la séquence la plus courte en suivant l'architecture Client/Serveur, en utilisant CUDA.
- La résolution parallèle complète du problème de la séquence la plus courte en suivant l'architecture Client/Serveur, en utilisant OpenMP.
- La résolution parallèle complète du problème de la séquence la plus courte en suivant l'architecture Client/Serveur, en utilisant MPI.
- Tous les tests sont fait, à savoir : tests unitaires, tests fonctionnels, tests de configuration, tests d'installation et les tests de performances.
- L'intégration continue de la compilation et l'exécution des différents tests sur Gitlab-ci.
- Création des différents documents utiles pour la compréhension du projet : cahier des tests, manuel de configuration, manuel d'installation ainsi que le cahier du développeur.
- Le rapport final a été rédigé.

2 Ce qui reste à faire

Toutes les tâches fixées au début par mon tuteur ont été faites, mais ce projet peut être amélioré en choisissant une autre stratégie pour résoudre le problème de la séquence la plus courte. En effet, dans mon projet, le serveur génère n'importe quel niveau des décisions qui les stocke dans une file d'attente pour les envoyés aux clients libres. Ces clients génèrent toutes les feuilles et renvoient leur résultats locaux (la séquence la plus courte de chaque décisions). Le serveur ensuite récupère tous ces résultats locaux puis en déduit la séquence la plus courte globale.

Une autre stratégie peut consister sur le contrôle des échanges entre le client et le serveur. Autrement dit, le serveur peut envoyer au début une décision à un client, puis celui-ci génère d'autres décisions et les renvoie au serveur qui les stocke dans sa file d'attente pour ensuite les envoyer à d'autres clients. (Un jeu d'envoi et de récupérations des décisions peut être mis en place)

3 Planning

Toutes les tâches sont décrites en détail dans l'annexe D (Gestion de projet), ainsi que les diagrammes de Gantt & PERT du semestre 9 et 10. [5](#) (Annexe C) & [4](#) (Annexe C).

4 Bilan sur la qualité

La qualité des travaux est assurée par les aspects suivants :

1. **La qualité du code :** le code est bien structuré, en effet différents fichiers sont utilisés pour scinder les différentes parties du projet. Nous avons à chaque fois deux fichiers .c et .h. Le code source est bien commenté et les variables sont bien nommées.
2. **La qualité de la gestion du code :** L'outil (INDENT) est utilisé pour contrôler le code et les bonnes pratiques. Pour la gestion des différentes versions du projet un GITLAB a été utilisé avec plusieurs branches. Chaque nouvelle fonctionnalité a été faite sur une nouvelle branche puis mergée si elle est fonctionnelle. L'intégration continue (GITLAB-CI) a été faite aussi pour automatiser les différents tests et la compilation du projet. (Jenkins a été utilisé au début en local puis j'ai migré sur GITLAB-CI pour que n'importe quel personne travaillant dans le futur sur le même projet puissent profiter de cette fonctionnalité).
3. **La qualité de la documentation :** Plusieurs documents ont été faits pour faciliter la compréhension des différentes parties du projet, et pour qu'il soit aussi facile à reprendre pour une éventuelle mise à jour ou/et à niveau. Les documents fournis sont : manuel d'installation, manuel de configuration, cahier des tests et le cahier de développeur. La documentation DOXYGENE a été générée pour la compréhension de la structure du code et toutes les fonctions et structures employées dans les différentes parties du projet.

La qualité du code a été vérifiée par Mr. Jean-Yves RAMEL, ainsi que par mon tuteur Mr. Patrick MARTINEAU.

5 Bilan auto-critique

Dans la première partie de mon PRD (Semestre 9), j'avais passé plus de temps que prévu sur la modélisation (réfléchir à un algorithme qu'on pourra paralléliser par la suite), ce qui a causé un peu de retard dans le déroulement des tâches suivantes. Cependant, ce retard a été rattrapé dans le deuxième semestre (S10), et toutes les tâches prévues ont été faites comme il faut.

Dans la deuxième partie de mon PRD, et plus précisément dans le développement de la solution en utilisant CUDA, j'ai un problème d'allocation mémoire. En effet, j'ai effectué plusieurs tests de performances qui m'a permis de constater l'erreur.

Sur la machine (UNIX B), le programme renvoie une erreur lorsque la longueur de la séquence est supérieure à 10. Par contre sur ma machine personnelle, le programme arrête de fonctionner lorsque la séquence est supérieure à 12. (GPU de ma machine personnelle est un peu plus puissant que celui des machines UNIX B)

Ce bug est dû à l'allocation mémoire dans le GPU. J'ai pas eu le temps pour résoudre ce problème imprévu. Par contre, pour les autres parties du projet, à savoir : OpenMP, MPI et la partie Séquentiel fonctionnent très bien, mais ils peuvent être améliorés en terme d'allocation mémoire. (Tests Valgrind montrent que quelques espaces mémoires sont alloués mais pas désalloués) .

Sinon pour le reste tout fonctionne bien.

Tout au long, de cette année j'ai pu approfondir et maîtriser les différentes technologies de parallélisme (CUDA, OpenMP et CUDA), qui n'étaient pas mon point fort pendant l'année DI4. En effet, pour chaque technologie je sais l'utiliser proprement et dans quel genre de situation (**notion de Granularité**). J'ai aussi approfondi mes connaissances dans le langage C, que j'utilise très peu pendant le développement de mes projets personnels.

Annexes

A

Spécifications fonctionnelles

Voici les fonctions principales du projet, et qui seront employées pour la résolution du problème de la séquence la plus courte, comme décrites dans le diagramme des cas d'utilisation. Quelques fonctions seront susceptibles à la modification.

1 Diagramme de structure

Le Diagramme de structure qui a été mis en place se présente ainsi :

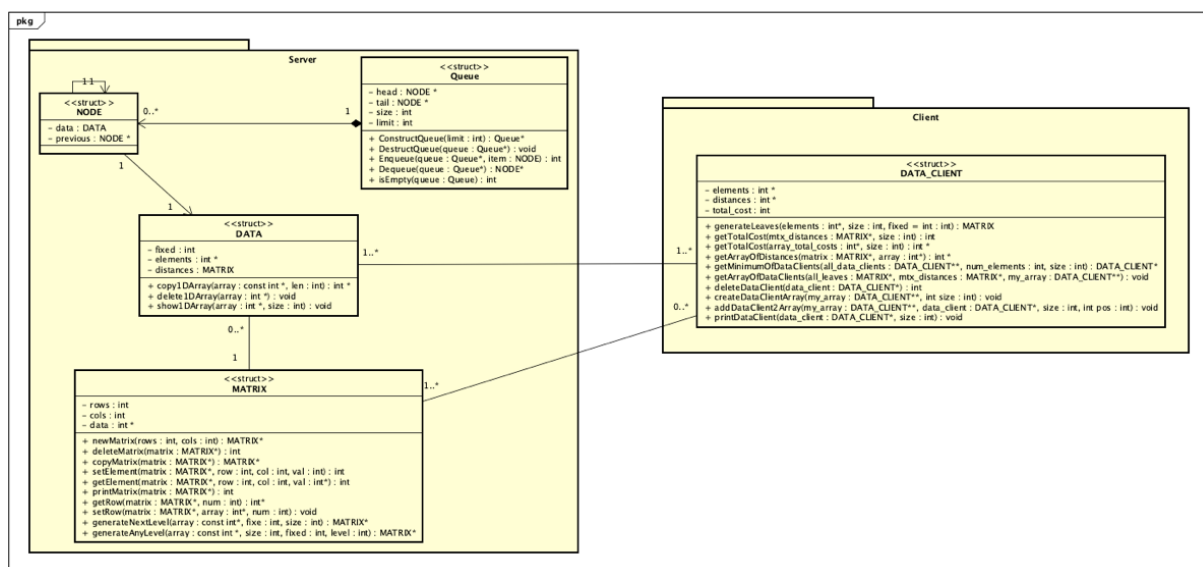


Figure 1 – Temps d'exécution en fonction de la longueur de la séquence

Nous avons une architecture Serveur / Clients. Dans la partie client , le serveur récupère la matrice des distances et il la stocke dans une structure MATRIX. Ensuite, il génère toutes un niveau donné des décisions. Chaque décision est ainsi stockée dans une structure DATA. Toutes les structures générées sont empilées dans une file d'attente de type FIFO(First In First Out).

Dans un deuxième temps le serveur envoie les différentes décisions à traiter à plusieurs clients. Lorsque celui-ci reçoit une décision il génère tous les résultats et en déduit la séquence la plus courte local qui stockera dans une structure DATA CLIENT. Cette solution locale sera renvoyée au serveur. Celui ci récupère toutes les solutions locales des différents clients pour en déduire la solution globale, qui correspond à la séquence la plus courte.

Chaque Structure a ses propres attributs et des méthodes. La documentation Doxygen générée explique en détail ce que fait chaque fonction et à quoi correspond chaque attribut.

2 Définition de la fonction 1 : get data from file

Identification de la fonction 1 :

Cette fonction permet de récupérer la matrice des distances à partir d'un fichier.

Présentation de la fonction 1 :

- Nom : getDataFromFile
- Priorité : Primordiale

Description de la fonction 1 :

Cette fonction va lire un fichier .csv puis va récupérer la matrice des distances (les distances entre tous les points)

Description précisée de la fonction 1 :

- Entrées : le nom du fichier
- Sorties : Une structure de type MATRIX qui contient toutes les distances
- : Exceptions : Si le fichier n'existe pas, ou le fichier n'est pas en bon format

3 Définition de la fonction 2 : write into log file

Identification de la fonction 2 :

Cette fonction permet d'enregistrer les résultats du problème de la séquence la plus courte dans un fichier, ainsi que toutes les actions faites par le serveur.

Présentation de la fonction 2 :

- Nom : writeIntoLogFile
- Priorité : Primordiale

Description de la fonction 2 :

Cette fonction va pouvoir enregistrer toutes les actions menés par le serveur : Générations des décisions, envoi des décisions à différents clients ... etc.

Description précisée de la fonction 2 :

- Entrées : Les séquences et leur cout total
- Sorties : une variable de type int : 1 si le fichier est créé et toutes les informations sont enregistrées, 0 sinon.

4 Définition de la fonction 3 : compare costs

Identification de la fonction 3 :

Cette fonction permet de comparer les couts de toutes les séquences.

Présentation de la fonction 3 :

- Nom : CompareCosts
- Priorité : Primordiale

Description de la fonction 3 :

Cette fonction va permettre de comparer les couts de toutes les séquences et récupérer les/la séquence qui a le plus petit cout.

Description précisée de la fonction 3 :

- Entrées : Toutes les séquences et leur cout total
- Sorties : la séquence la plus courte et son cout.

5 Définition de la fonction 4 : Create a queue

Identification de la fonction 4 :

Cette fonction permet de créer une file d'attente.

Présentation de la fonction 4 :

- Nom : CreateQueue
- Priorité : Primordiale

Description de la fonction 4 :

Cette fonction crée une file d'attente en choisissant sa capacité totale

Description précisée de la fonction 4 :

- Entrées : n : nombre entier
- Sorties : une file d'attente avec une capacité n

6 Définition de la fonction 5 : enqueue an element

Identification de la fonction 5 :

Cette fonction permet d'ajouter un élément à une file d'attente.

Présentation de la fonction 5 :

- Nom : EnqueueElement
- Priorité : Primordiale

Description de la fonction 5 :

Cette fonction ajoute dans la file d'attente un nouveau élément

Description précisée de la fonction 5 :

- Entrées : l'élément à ajouter
- Sorties : true si l'élément est ajouté , false sinon

7 Définition de la fonction 6 : dequeue an element**Identification de la fonction 6 :**

Cette fonction permet d'enlever un élément d'une file d'attente.

Présentation de la fonction 6 :

- Nom : DequeueElement
- Priorité : Primordiale

Description de la fonction 6 :

Cette fonction permet d'enlever un élément de la file d'attente en suivant la méthode FIFO (first in first out)

Description précisée de la fonction 6 :

- Entrées : NONE
- Sorties : true si l'élément est enlevé , false sinon

8 Définition de la fonction 7 : Generate next level**Identification de la fonction 7 :**

Cette fonction permet de générer le niveau suivant d'une décision

Présentation de la fonction 7 :

- Nom : GenerateNextLevel
- Priorité : Primordiale

Description de la fonction 7 :

Cette fonction va recevoir une décision , puis elle va générer toutes les décisions du niveau suivant.

Description précisée de la fonction 7 :

- Entrées : une décision
- Sorties : structure de type MATRIX contenant toutes les décisions générées

9 Définition de la fonction 8 : Generate any level**Identification de la fonction 8 :**

Cette fonction permet de générer n'importe quel niveau d'une décision

Présentation de la fonction 8 :

- Nom : GenerateAnyLevel
- Priorité : Primordiale

Description de la fonction 8 :

Cette fonction va recevoir une décision , puis elle va générer toutes les décisions du niveau voulu.

Description précisée de la fonction 8 :

- Entrées : une décision
- Sorties : structure de type MATRIX contenant toutes les décisions générées du niveau voulu

10 Définition de la fonction 9 : Generate leaves**Identification de la fonction 9 :**

Cette fonction permet de générer les feuilles d'une décision

Présentation de la fonction 9 :

- Nom : GenerateLeaves
- Priorité : Primordiale

Description de la fonction 9 :

Cette fonction va recevoir une décision , puis elle va générer toutes les décisions du dernier niveau.

Description précisée de la fonction 9 :

- Entrées :une décision
- Sorties : une structure de type MATRIX contenant toutes les séquences possibles d'une décision

11 Définition de la fonction 10 : Calculate Total Cost**Identification de la fonction 10 :**

Cette fonction permet de calculer le cout total d'une séquence

Présentation de la fonction 10 :

- Nom : CalculateTotalCost
- Priorité : Primordiale

Description de la fonction 10 :

Cette fonction permet de calculer et renvoyer le cout total d'une séquence, le cout est égale à la somme des distances entre deux sommets qui se suivent.

Description précisée de la fonction 10 :

- Entrées : une séquence et la matrice des distances
- Sorties : le cout total de la séquence

12 Définition de la fonction 11 : Compare All Costs**Identification de la fonction 11 :**

Cette fonction permet de comparer tout les coûts.

Présentation de la fonction 11 :

- Nom : CompareAllCosts
- Priorité : Primordiale

Description de la fonction 11 :

Cette fonction compare tous les cout des séquences et renvoie la séquence la plus courte

Description précisée de la fonction 11 :

- Entrées : un tableau des séquences et leur cout
- Sorties : La séquence la plus courte et son cout.

B

Spécifications non fonctionnelles

1 Contraintes de développement et conception

- **IDE** : Tous les résultats seront visualisés en console et des fichiers de log/résultats en utilisant un terminal. En effet un fichier Makefile a été configuré pour chaque partie du projet pour faciliter la compilation et l'exécution des différentes tâches.
- **Langage de programmation** : C
- **Bibliothèques** : CUDA, OpenMP et MPI

2 Contraintes de fonctionnement et d'exploitation

Dans cette partie nous allons voir les 4 enjeux qui décrivent les Contraintes de fonctionnement et d'exploitation.

2.1 Performances

L'algorithme qui sera proposé contiendra plusieurs itérations, partage de mémoire entre différents processus, Communication entre plusieurs CPU donc tous ces paramètres doivent être considéré pour avoir le meilleur résultat en terme de performance. Le choix du modèle de parallélisation compte aussi.

Quant à la gestion des fichier, seul le serveur s'occupera de l'enregistrement des différentes opérations ainsi que des résultats obtenus.

Pour comparer la performance de chaque implémentation (séquentielle ou parallèle), nous allons se baser sur le temps d'exécution de l'ensemble des méthodes. Donc nous allons mettre en place des éléments pour le calcul de temps d'exécution, pour ensuite en déduire la différence.

L'utilisation des ressources va être aussi contrôlée. (allocation et dés allocation mémoire) La vérification sera faite en utilisant VALGRIND.

2.2 Capacités

Nous avons supposé au début que toutes les machines existent, donc nous n'avons pas de problème concernant la capacité des clients. Nous allons changer ce paramètre dans notre programme pour voir ce qui se passe au niveau des performances. Les seules choses qui nous limitent sont les caractéristiques de la machine sur laquelle on exécute le projet.

2.3 Modes de fonctionnement

- Pour lancer le programme, il faut utiliser des lignes de commandes du Makefile pour lancer le projet voulu. (Le manuel d'installation et de configuration expliquent en détailles les différents commandes).
- L'arrêt du programme se produit lorsque le résultat final est donné. Si une erreur est produite le programme devra reprendre les calculs en se basant sur le fichier de log.

2.4 contrôlabilité

- Afin de suivre l'exécution du programme un fichier de log ainsi q'un fichier de résultat seront créés. En cas d'erreur, le système peu reprendre à partir de ces deux fichiers. Pour les tests des les résultats peuvent être afficher sur la console par le biais de la fonction : **printf()**.
- Afin de faire un suivi des calculs faits en tache de font, un fichier de log sera crée avec la date de chaque opération.

2.5 Sécurité

Il n'y a pas de demande de sécurité par l'encadrant. Le système est développé juste pour résoudre le problème de la séquence la plus courte en exploitant les différents niveaux de parallélisme. Le but n'est pas produire un programme commercial.

Le modèle suivi pour la gestion de projet est **Le modèle en Cascade**. Dans le premier semestre S09, ces trois cycles sont faits : l'étude de faisabilité, les spécifications (Fonctionnelles et NON fonctionnelles) ainsi que l'analyse. Les autres cycles seront faits pendant le second semestre S10, à savoir : la conception détaillée des premiers algorithmes, l'implémentation des différentes parties de notre projet et en fin la mise en place des tests.

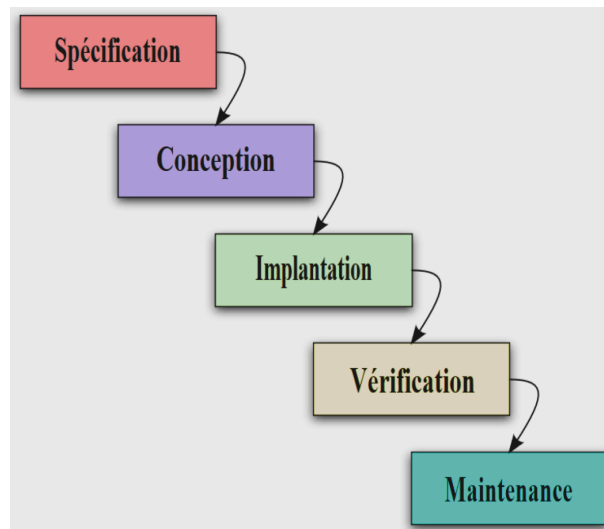


Figure 1 – Modèle en cascade

1 Aperçu de gestion de projet

Pour la planification du projet, j'ai créé le diagramme de Gantt suivant (Figure 2).

Le diagramme PERT du projet est présenté dans la figure (3).

Quant au semestre S09, le Projet de Réalisation et Développement commence le 11/09/2018 et prend fin le 12/12/2018 (Date de la soutenance). Le S10 (Partie développement) commence juste après la soutenance c'est-à-dire le 13/12/2018, et se termine le 27/04/2019. Comme vous le voyez dans le diagramme j'ai essayé de gagner un peu de temps pour entamer la modélisation

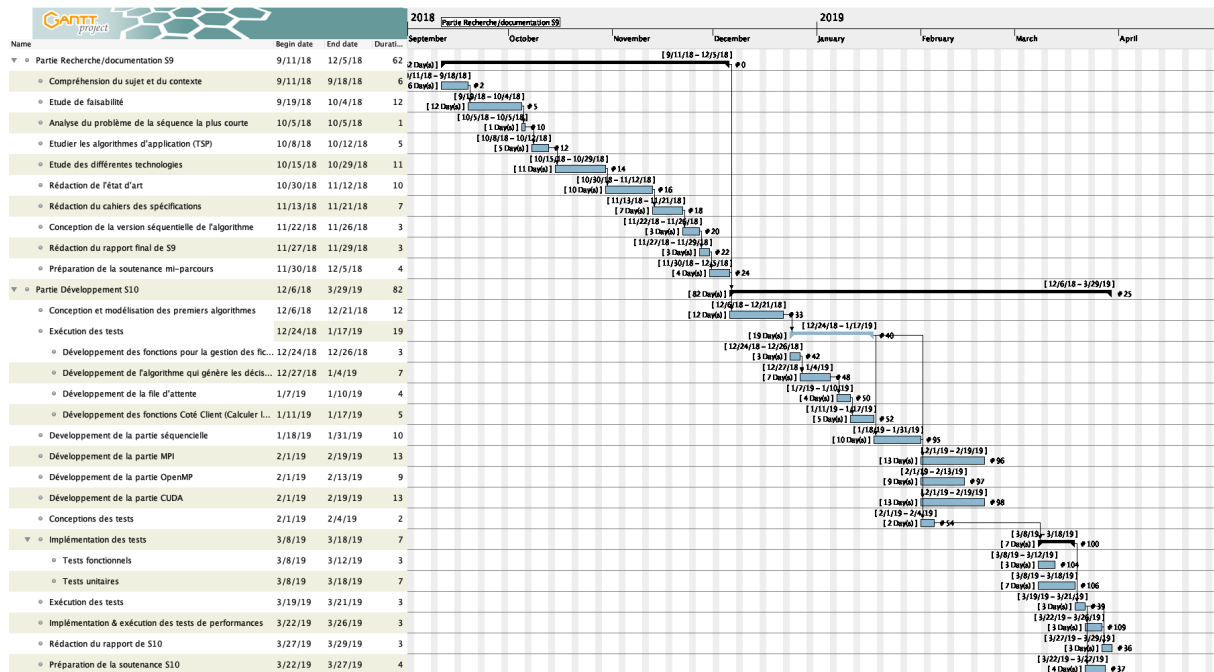


Figure 2 – Diagramme de Gantt

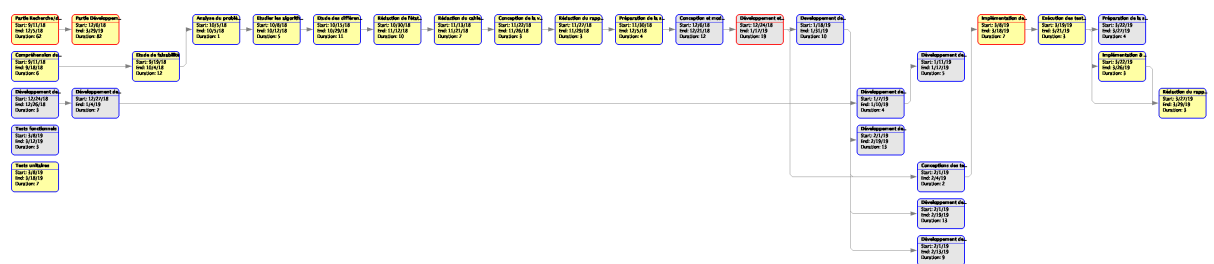


Figure 3 – Diagramme de Gantt

des différents algorithmes et commencer à programmer les différentes parties pour avoir un peu de temps à la fin du projet pour notamment améliorer les différents algorithmes.

La figure 4, affiche clairement les différents phases ainsi que les tâches du projet durant le semestre S9.

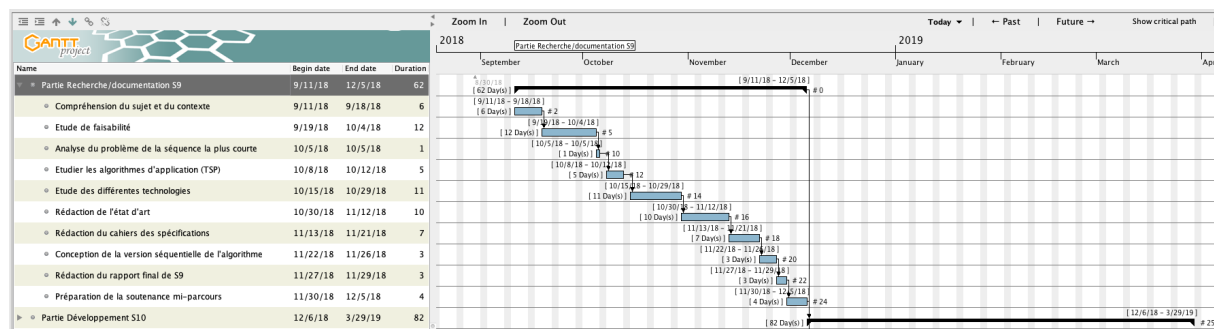


Figure 4 – Diagramme de Gantt : Les différentes tâches

La figure 5, affiche clairement les différentes phases ainsi que les tâches du projet durant le semestre S10.

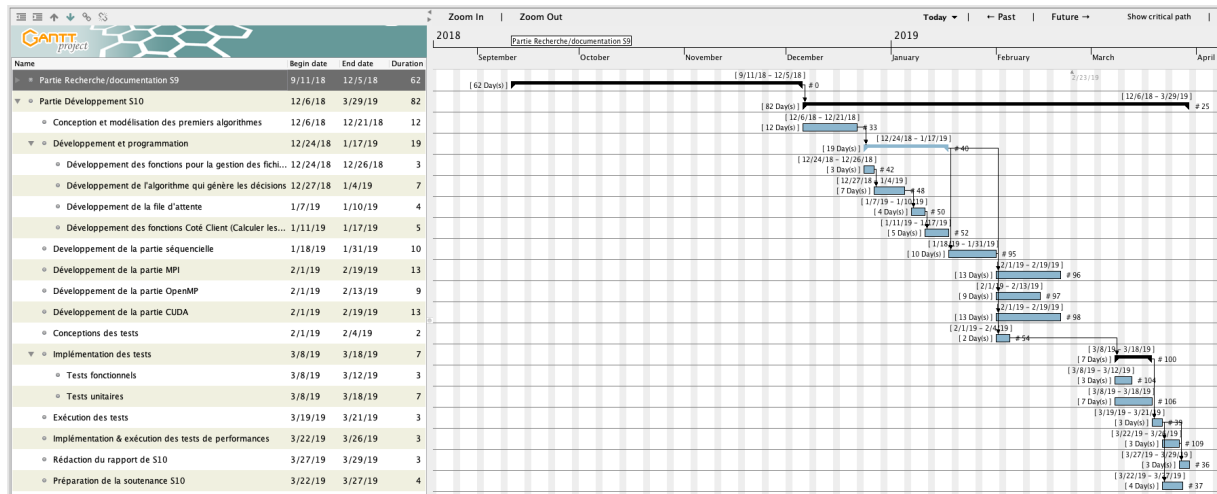


Figure 5 – Diagramme de Gantt : Les différentes tâches

2 Découpage en tâches

Partie Recherche et Documentation S09

2.1 Tâche 1 : Compréhension du sujet et du contexte

1. **Date de début** : Le 11 Septembre 2018
2. **Date de fin** : Le 18 Septembre 2018
3. **Durée** : 6 jours/H
4. **Description** : Comprendre le sujet et connaître l'objectif et l'idée générale du sujet.

2.2 Tâche 2 : Etude de faisabilité

1. **Date de début** : Le 19 Septembre 2018
2. **Date de fin** : Le 04 Octobre 2018
3. **Durée** : 12 jours/H
4. **Description** : Etudier la faisabilité technique et commencer à mettre en place les spécifications.

2.3 Tâche 3 : Analyse du problème de la séquence la plus courte

1. **Date de début** : Le 05 Octobre 2018
2. **Date de fin** : Le 05 Octobre 2018
3. **Durée** : 1 jour/H
4. **Description** : Comprendre le problème qui sera résolu en utilisant les techniques de parallélisation.

2.4 Tâche 4 : Etudier quelques algorithmes d'application

1. **Date de début** : Le 08 Octobre 2018
2. **Date de fin** : Le 12 Octobre 2018
3. **Durée** : 5 jours/H
4. **Description** : Etudier quelques exemples du problème de la séquence la plus courte, comme le problème du TSP (Traveling Sales Man)

2.5 Tâche 5 : Etude des différentes technologies

1. **Date de début** : Le 08 Octobre 2018
2. **Date de fin** : Le 29 Octobre 2018
3. **Durée** : 11 jours/H
4. **Description** : - Etudier les différentes technologies pour paralléliser les tâches ainsi que leurs avantages.
- Etudier les différents modes de parallélisation.

2.6 Tâche 6 : Rédaction de l'état d'art

1. **Date de début** : Le 30 Octobre 2018
2. **Date de fin** : Le 12 Novembre 2018
3. **Durée** : 10 Jours/H
4. **Description** : Rédiger les différentes parties de l'états d'art, en expliquant en détails les technologies utilisées et les modèles existants.
5. **Livrables** : Rendre l'état d'art sous forme d'un PDF propre au Tuteur.

2.7 Tâche 7 : Rédaction du cahier des spécifications

1. **Date de début** : Le 13 Novembre 2018
2. **Date de fin** : Le 21 Novembre 2018
3. **Durée** : 7 jours/H
4. **Description** : Rédiger au propre le cahier des spécifications détaillé.
5. **Livrables** : Le Cahier des spécification à rendre pour qu'il soit validé par le tuteur.

2.8 Tâche 8 : Conception de la première version séquentielle de l'algorithme

1. **Date de début** : Le 22 Novembre 2018
2. **Date de fin** : Le 26 Novembre 2018
3. **Durée** : 3 jours/H
4. **Description** : Mettre en place une stratégie pour la génération de toutes les combinaisons possibles et calculer le cout total de chacune.

2.9 Tâche 9 : Rédaction du rapport final pour S09

1. **Date de début** : Le 27 Novembre 2018
2. **Date de fin** : Le 29 Novembre 2018

3. **Durée** : 3 jours/H
4. **Description** : Rédiger tout les livrables en latex, à savoir : Cahier des spécifications, Etat d'art et Gestion de projet.
5. **Livrables** : Un fichier Pdf qui contient les livrables.

2.10 Tâche 10 : Préparation de la soutenance mi-parcours

1. **Date de début** : Le 30 Novembre 2018
2. **Date de fin** : Le 05 Décembre 2018
3. **Durée** : 4 jours/H
4. **Description** : Préparer les diapos pour la soutenance. Un fichier PowerPoint (Slides) doit être livré.

Partie Programmation et développement S10

2.11 Tâche 11 : Conception et modélisation de premiers algorithmes

1. **Date de début** : Le 06 Décembre 2018
2. **Date de fin** : Le 21 Décembre 2018
3. **Durée** : 8 jours/H
4. **Description** : Modéliser les différents algorithmes pour résoudre le problème de la séquence la plus courte (Algorithme de génération des décisions par exemple)
5. **Livrables** : Différents Algorithmes, le livrable sera sous forme d'une présentation.

2.12 Tâche 12 : Développement des méthodes pour la gestion des fichiers

1. **Date de début** : Le 24 Décembre 2018
2. **Date de fin** : Le 26 Décembre 2018
3. **Durée** : 3 jours/H
4. **Description** : Développer toutes les méthodes pour gérer les différents fichiers.
 - Gérer des fichiers de log
 - Gérer des fichiers d'erreur.
 - Ecrire les résultats dans un fichier.
 - Récupérer les données dans un fichier...
5. **Livrables** : Des fichier .c (code source), qui seront push sur le gestionnaire de version : Github, dont l'accès est donné au tuteur.

2.13 Tâche 13 : Développement de l'algorithme qui génère les décisions

1. **Date de début** : Le 27 Décembre 2019
2. **Date de fin** : Le 4 Janvier 2019
3. **Durée** : 16 jours/H
4. **Description** : Développer la partie du Scheduler, c'est-à-dire mettre en place l'algorithme qui permet de distribuer les lots des séquences au différents Clients.
5. **Livrables** : Des fichier .c (code source), qui seront push sur le gestionnaire de version : Github, dont l'accès est donné au tuteur.

2.14 Tâche 14 : Développement de la file d'attente

1. **Date de début** : Le 7 Janvier 2019
2. **Date de fin** : Le 10 Janvier 2019
3. **Durée** : 13 jours/H
4. **Description** : Programmer l'algorithme qui permet de faire la gestion d'une file d'attente selon des critères spécifiques.
5. **Livrables** : Des fichier .c (code source), qui seront push sur le gestionnaire de version : Github, dont l'accès est donné au tuteur.

2.15 Tâche 15 : Développement des fonctions coté Client

1. **Date de début** : Le 11 Janvier 2019
2. **Date de fin** : Le 17 Janvier 2019
3. **Durée** : 15 jours/H
4. **Description** : Programmer les différentes méthodes qui permettent de générer toutes les combinaisons possibles, calculer le cout de chaque combinaison.
5. **Livrables** : Des fichier .c (code source), qui seront push sur le gestionnaire de version : Github, dont l'accès est donné au tuteur.

2.16 Tâche 16 : Développement de la partie séquentielle

1. **Date de début** : Le 18 Janvier 2019
2. **Date de fin** : Le 31 Janvier 2019
3. **Durée** : 7 jours/H
4. **Description** : Développer toute la partie séquentielle
5. **Livrables** : Des fichier .c (code source), qui seront push sur le gestionnaire de version : Github, dont l'accès est donné au tuteur.

2.17 Tâche 17 : Développement de la partie MPI

1. **Date de début** : Le 1 Février 2019
2. **Date de fin** : Le 19 Février 2019
3. **Durée** : 3 jours/H
4. **Description** : Développer toute la partie parallèle en utilisant MPI.
5. **Livrables** : Le rapport final sous forme d'un PDF.

2.18 Tâche 18 : Développement de la partie OpenMP

1. **Date de début** : Le 1 Février 2019
2. **Date de fin** : Le 13 Février 2019
3. **Durée** : 4 jours/H
4. **Description** : Développer toute la partie parallèle en utilisant OpenMP.

2.19 Tâche 19 : Développement de la partie CUDA

1. **Date de début :** Le 1 Février 2019
2. **Date de fin :** Le 19 Février 2019
3. **Durée :** 4 jours/H
4. **Description :** Dans cette tâche je développerais toute la partie parallèle en utilisant CUDA.

2.20 Tâche 20 : Conception des tests

1. **Date de début :** Le 1 Février 2019
2. **Date de fin :** Le 4 Février 2019
3. **Durée :** 4 jours/H
4. **Description :** Mettre en place une stratégie pour faire les tests

2.21 Tâche 21 : Implémentation des tests

1. **Date de début :** Le 8 Mars 2019
2. **Date de fin :** Le 18 Mars 2019
3. **Durée :** 4 jours/H
4. **Description :** Tester toutes les fonctions du projet (Tests unitaires , tests fonctionnels , tests d'installation et de configuration)

2.22 Tâche 22 : Exécution des tests

1. **Date de début :** Le 19 Mars 2019
2. **Date de fin :** Le 21 Mars 2019
3. **Durée :** 4 jours/H
4. **Description :** Exécuter les tests et corriger les anomalies

2.23 Tâche 23 : Rédaction du rapport final

1. **Date de début :** Le 27 Mars 2019
2. **Date de fin :** Le 29 Mars 2019
3. **Durée :** 4 jours/H
4. **Description :** Rédiger le rapport final

2.24 Tâche 24 : Préparation de la soutenance S10

1. **Date de début :** Le 22 Mars 2019
2. **Date de fin :** Le 27 Mars 2019
3. **Durée :** 4 jours/H
4. **Description :** Préparer des slides pour la présentation finale.

Remarques


Pour certaines sections notamment dans la partie développement la totalité des fonctions seront programmées en utilisant qu'une seule version de parallélisme. Par exemple la première version sera de programmer toutes les fonctions en utilisant OpenMP, la deuxième en utilisant CUDA et la troisième en utilisant MPI. Cela sera fait afin d'avoir un programme fonctionnel. Si tout se passe bien, un programme en utilisant les trois technologies au même temps sera mis en place. Dans ce projet c'est évidemment l'idéal mais par contrainte de temps cela sera un peu plus complexe à mettre dans une durée assez courte (après voir discuter avec le tuteur). Mais cela reste faisable.

Des durées (partie développement des fonctions) peuvent être changées un peu en fonction de l'avancement des différentes versions.

3 Planning

3.1 Le planning de recherche

Voici le planning (6) de la partie Recherche et documentation



Name	Begin date	End date	Duration
▼ * Partie Recherche/documentation S9	9/11/18	12/5/18	62
• Compréhension du sujet et du contexte	9/11/18	9/18/18	6
• Etude de faisabilité	9/19/18	10/4/18	12
• Analyse du problème de la séquence la plus courte	10/5/18	10/5/18	1
• Etudier les algorithmes d'application (TSP)	10/8/18	10/12/18	5
• Etude des différentes technologies	10/15/18	10/29/18	11
• Rédaction de l'état d'art	10/30/18	11/12/18	10
• Rédaction du cahiers des spécifications	11/13/18	11/21/18	7
• Conception de la version séquentielle de l'algorithme	11/22/18	11/26/18	3
• Rédaction du rapport final de S9	11/27/18	11/29/18	3
• Préparation de la soutenance mi-parcours	11/30/18	12/5/18	4

Figure 6 – Diagramme de Gantt : Planning de recherche et documentation

3.2 Le planning de développement

Voici le planning (7) de la partie développement et programmation.



Name	Begin date	End date	Duration
► • Partie Recherche/documentation S9	9/11/18	12/5/18	62
▼ • * Partie Développement S10	12/6/18	3/29/19	82
• Conception et modélisation des premiers algorithmes	12/6/18	12/21/18	12
▼ • Développement et programmation	12/24/18	1/17/19	19
• Développement des fonctions pour la gestion des fichi...	12/24/18	12/26/18	3
• Développement de l'algorithme qui génère les décisions	12/27/18	1/4/19	7
• Développement de la file d'attente	1/7/19	1/10/19	4
• Développement des fonctions Coté Client (Calculer les...	1/11/19	1/17/19	5
• Développement de la partie séquentielle	1/18/19	1/31/19	10
• Développement de la partie MPI	2/1/19	2/19/19	13
• Développement de la partie OpenMP	2/1/19	2/13/19	9
• Développement de la partie CUDA	2/1/19	2/19/19	13
• Développement de la partie CUDA	2/1/19	2/4/19	2
▼ • Implémentation des tests	3/8/19	3/18/19	7
• Tests fonctionnels	3/8/19	3/12/19	3
• Tests unitaires	3/8/19	3/18/19	7
• Exécution des tests	3/19/19	3/21/19	3
• Implémentation & exécution des tests de performances	3/22/19	3/26/19	3
• Rédaction du rapport de S10	3/27/19	3/29/19	3
• Préparation de la soutenance S10	3/22/19	3/27/19	4

Figure 7 – Diagramme de Gantt : Planning de développement

D

Manuel d'installation

Toutes les étapes d'installation des différentes dépendances et bibliothèques sont détaillé dans un manuel d'installation joint à ce rapport final.

cf (**Manuel d'installation**)

E

Manuel de configuration

Toutes les étapes de configuration des différentes parties du projet (OpenMP, séquentielle, MPI et CUDA) sont expliquées en détail dans un manuel de configuration joint à ce rapport final.

cf (**Manuel de configuration**)

F

Cahier de développeurs

Un cahier de développeurs est joint à ce rapport final, où tous les diagrammes, algorithmes et la structure du projet sont expliqués en détail .

cf (**Cahier de développeurs**)



Cahier de tests

Un Document Cahier de tests est joint avec ce rapport final. Il contient :

- Toutes les exigences à tester
- La stratégie employée pour les tests
- Les outils utilisés pour les tests
- Les différents livrables
- Le plan de tests
- Les résultats de chaque tests

cf (**Cahier de tests**)

- [1] :M. Ameer SOUKHAL. "Cours Big-Data" Ecole polytechnique de TOURS.
- [2] :M. Patrick MARTINEAU. "Mise en oeuvre GPU" et "Mise en oeuvre multi-core", Ecole polytechnique de TOURS.
- [3] :Guillaume Laville. Exécution efficace des systèmes multi agents sur GPU. Calcul parallèle, distribué et partagé. Université de Franche Comté 2014.
- [4] :Jérôme Clet Ortega. Exploitation efficace des architectures parallèles de type grappes de NUMA à l'aide de modèles hybrides de programmation. Calcul parallèle, distribué et partagé. Université Sciences et Technologies Bordeaux 1, 2012.



Comptes rendus hebdomadaires

Compte rendu n°1 du 17/09/2018

Je vous envoie ce mail pour vous tenir au courant de l'avancement de mon PR&D. Je cherche toujours une solution pour le problème de la séquence la plus courte. Actuellement, je suis entrain d'analyser deux algorithmes du plus court chemin, à savoir : Johnson's algorithm et Floyd-Warshall algorithm.

Pour Mercredi prochain, je souhaite arriver à la séance en ayant une première version d'algorithme qui sera adaptable au problème.

Compte rendu n°2 du 24/09/2018

J'ai essayé de trouver une bonne approximation au problème. En effet, j'ai implémenté l'algorithme de Floyd-warshall avec une complexité de $O(n^3)$. L'algorithme va me permettre d'évaluer toutes les combinaisons possibles, puis, en déduire une matrice des distances des plus courts chemins entre toutes les paires de sommets.

Celle-ci sera exploitée afin de définir la séquence la plus courte en se basant sur des algorithmes d'approximation du TSP (voyageur de commerce).

Compte rendu n°3 du 01/10/2018

J'ai créé un répertoire GitHub pour que je puisse bien gérer mon projet et que vous ayez une meilleure visualisation de mon état d'avancement. Voici le lien Git : <https://github.com/DevPsksh/PR-D.git>. Quant au projet, j'ai quelques questions à vous poser. J'aimerais bien convenir d'un Rendez-vous. Je vous remercie d'avance.

Compte rendu n°4 du 08/10/2018

Comme nous avons pu échanger la semaine dernière, j'ai fait des recherches sur les différentes technologies en détails afin d'obtenir les meilleurs résultats en performance. J'ai essayé d'analyser les notions de parallélisme afin de trouver un compromis entre équilibrage des calculs intensifs et minimisation des communications entre différents processus (granularité).

Quant au problème de la séquence la plus courte, je suis parti sur la résolution par brute force, c'est-à-dire mettre en place un algorithme qui génère toutes les permutations possibles, déduit le coût total de chaque permutation, puis renvoie la séquence la plus courte. La complexité est $O(n!)$.

Je mettrais en place les différentes technologies de parallélisme pour avoir le meilleur temps d'exécution possible.

Compte rendu n°5 du 15/10/2018

La semaine dernière, j'ai commencé à éditer mon cahier des spécifications et à créer quelques diagrammes de modélisation. En parallèle j'ai implémenté un algorithme qui génère toutes les permutations possibles.

Cette semaine j'essayerais de finir la partie consacrée à l'état d'art pour que nous puissions discuter ensemble à propos des différents points à améliorer.

Compte rendu n°6 du 22/10/2018

J'ai quasiment fini l'état d'art de mon rapport. En effet, dans celui-ci j'ai abordé les points suivants : l'architecture de calcul parallèle, la programmation parallèle (les différents modèles et les avantages de chaque modèle) ainsi que les technologies utilisées. Quant au dernier point, j'aimerais bien savoir s'il faut citer uniquement les technologies que j'utiliserais dans la partie développement à savoir ;Cuda, OpenMP, MPI ou donner une vue globale des technologies qui existent aujourd'hui comme ThreadBuildingBlocks(TBB), OpenCL, PVM,

Cette semaine, je finirai l'état d'art puis j'aborderai la partie modélisation. Normalement après les vacances je commencerais à développer mon algorithme et mettre en place les différentes technologies.

Compte rendu n°7 du 13/11/2018

La semaine dernière, j'ai commencé à mettre en place mon algorithme. En effet, comme prévu, j'ai commencé à résoudre le problème de la séquence la plus courte en utilisant la force brute (en séquentiel). Autrement dit, lister toutes les combinaisons possibles puis calculer le coût total de chaque combinaison.

Cette semaine, je compte finir mon code en séquentiel pour ensuite entamer l'implémentation parallèle en utilisant les différentes technologies (MPI, Cuda, OpenMP).

Compte rendu n°8 du 19/11/2018

Quant à la partie séquentielle, il manque juste une méthode qui permet de calculer la valeur totale d'une combinaison.

La semaine dernière, j'ai pas du tout touché au code, par contre j'ai élaboré une stratégie pour la résolution du problème de la séquence la plus courte en utilisant les technologies (OpenMP, Cuda et MPI). J'ai passé la plupart de mon temps à travailler sur mon cahier de spécifications.

Ce Mercredi, j'ai un RDV avec Mr RAMEL pour discuter à propos des grandes parties de mon cahier des spécifications. Je vous transmettrai, la fin de cette semaine, la dernière version de mon rapport, afin d'avoir votre retour concernant mon livrable de ce semestre.

Compte rendu n°9 du 26/11/2018

Je vous présente d'avance mes excuses pour le retard d'envoi du rapport lié à un empêchement. J'ai mis au propre l'état d'art et j'aimerais bien avoir votre retour concernant celui-ci, notamment s'il faut détailler plus ou ajouter une partie omise. J'ai essayé d'être concis le maximum possible.

Cette semaine sera consacré à ajouter dans le rapport la partie spécifications fonctionnelles et la gestion de projet (planification). Si tout se passe bien, mon livrable sera prêt avant Jeudi prochain. J'essayerai de vous envoyer les livrables avant Jeudi, et sans faute cette fois, afin d'avoir votre confirmation pour ensuite commencer à programmer les différentes parties du projet.

Compte rendu n°10 du 29/11/2018

Veillez trouver ci-joint mon rapport avec les parties spécifications et Gestion de projet qui sont ajoutées. La version finale vous sera transmise la fin de cette semaine (quelques retouches doivent être faites).

J'aimerais bien savoir s'il faut détailler plus la partie "spécifications fonctionnelles" (ajouter par exemple les fonctions proposées par CUDA (cudaMemcpy, ...) et MPI (send, receive, scatter, gather, ...) ou même détailler plus la partie serveur ainsi que la partie client).

Serait-il possible de convenir d'un rendez-vous la semaine prochaine?

Compte rendu n°11 du 01/12/2018

Ci-joint la nouvelle version de mon rapport du PRD. (Les différents diagrammes ont été ajouté).

Le créneau du Mercredi 5/12 à 9h00 me convient parfaitement.

Compte rendu n°12 du XX/YY/2018

Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile : Problème de la séquence la plus courte

Ayoub IDEL

Encadrement : Patrick MARTINEAU

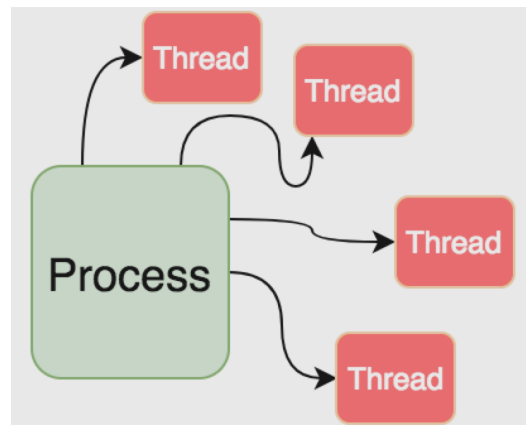


En collaboration avec Polytech'Tours

Objectifs

Dans ce projet nous allons résoudre un problème NP-difficile en utilisant les différents niveaux de parallélisme, à savoir :

- Parallélisme multiprocessus (MPI)
- Parallélisme multithreads (OpenMP)
- Parallélisme GPU (CUDA)



Parallélisation processus / threads

Mise en œuvre

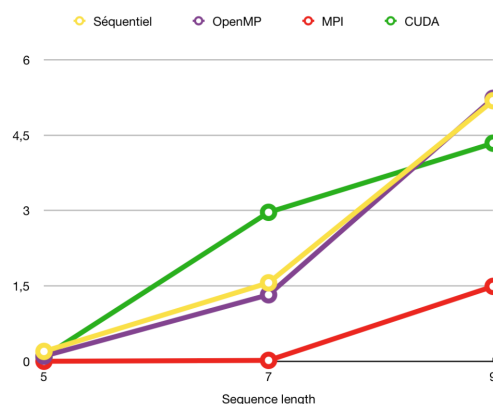
Dans un premier temps il faut proposer un algorithme séquentiel pour résoudre le problème de la séquence la plus courte. Ensuite cet algorithme doit être parallélisé en utilisant les différentes technologies, à savoir OpenMP, CUDA et MPI.



Les différentes technologies de parallélisme utilisées

Résultats attendus

La résolution du problème de la séquence la plus courte en exploitant les différentes technologies du parallélisme.



La différence entre les trois technologies en temps d'exécution en fonction de la taille de la séquence

Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile : Problème de la séquence la plus courte

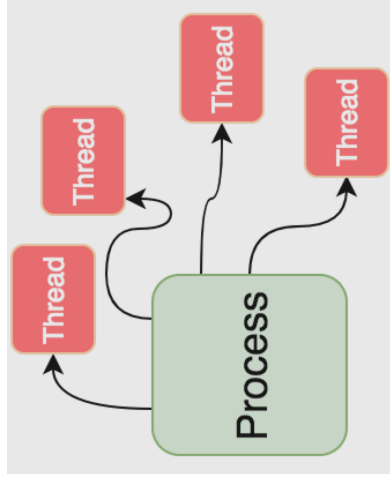
Ayoub IDEL

Encadrement : Patrick MARTINEAU

Objectifs

Dans ce projet nous allons résoudre un problème NP-difficile en utilisant les différents niveaux de parallélisme, à savoir :

- Parallélisme multiprocessus (MPI)
- Parallélisme multithreads (OpenMP)
- Parallélisme GPU (CUDA)



Parallélisation processus / threads

Mise en œuvre

Dans un premier temps il faut proposer un algorithme séquentiel pour résoudre le problème de la séquence la plus courte. Ensuite cet algorithme doit être parallélisé en utilisant les différentes technologies, à savoir OpenMP, CUDA et MPI.



OpenMP
Enabling HPC since 1997

Les différentes technologies de parallélisme utilisées



La différence entre les trois technologies en temps d'exécution en fonction de la taille de la séquence

Résultats attendus

La résolution du problème de la séquence la plus courte en exploitant les différentes technologies du parallélisme.

Mise en oeuvre des outils de parallélisme pour résoudre un problème NP-difficile

Problème de la séquence la plus courte

Résumé

Année après année, le traitement parallèle est considéré comme la meilleure méthode rentable, si ce n'est la seule, pour résoudre rapidement des problèmes de calcul complexes et gourmands en données. L'apparition d'ordinateurs parallèles puissants et moins cher, tels que les multi-processeurs de bureau et les clusters de postes de travail ou des grappes de calcul, a rendu ces méthodes parallèles généralement applicables, tout comme les normes logicielles pour la programmation parallèle portable. D'une part, les applications à forte intensité de données telles que le traitement des transactions et la recherche d'informations, l'exploration et l'analyse de données et les services multimédias ont constitué un nouveau défi pour la génération moderne de plateformes parallèles. D'autre part, les changements dans les architectures, les modèles de programmation et les applications ont des implications sur la façon dont les plates-formes parallèles sont mises à la disposition des programmeurs. L'interface MPI (Message Passing Interface), les threads OpenMP, ainsi que la programmation GPU en utilisant CUDA, ont été sélectionnés comme modèles de programmation les plus utilisés dans le calcul parallèle grâce à leur puissance, portabilité et surtout leur compatibilité. Dans ce projet, nous allons utiliser ces différentes technologies pour résoudre le problème de la séquence la plus courte. Celui-ci est un problème extrêmement difficile à résoudre. En effet, le problème est NP-difficile et sa complexité est de l'ordre de $O(n!)$.

Mots-clés

traitement parallèle, gourmands en données, le problème de la séquence la plus courte

Abstract

Year after year, parallel processing is considered the best, if not the only, cost-effective method for quickly solving complex and data-intensive computing problems. The emergence of powerful and cheaper parallel computers, such as desktop multiprocessors and desktop or cluster workstations, has made these parallel methods generally applicable, as have software standards for portable parallel programming. On the first hand, data-intensive applications such as transaction processing and information retrieval, data mining and analysis and multimedia services have presented a new challenge for the modern generation of parallel platforms. On the other hand, changes in architectures, programming models and applications have implications for how parallel platforms are made available to programmers. The MPI (Message Passing Interface), OpenMP threads, as well as GPU programming using CUDA, have been selected as the most commonly used programming models in parallel computing due to their power, portability and most importantly their compatibility. In this project, we will use these different technologies to solve the problem of the shortest sequence. This is an extremely difficult problem to solve. Indeed, the problem is NP-difficult and its complexity is in the order of $O(n!)$.

Keywords

parallel processing, data-intensive computing, the shortest sequence problem

Entreprise
Polytech'Tours



Tuteur entreprise
Patrick MARTINEAU

Étudiant
Ayoub IDEL (DI5)

Tuteur académique
Patrick MARTINEAU