Projet Systèmes Distribués
GNU Make distribué

Réalisé par :

ABAHAMID Soufiane

ATATRI Doaâ

BOUMEHDI Mohammed-Yassine

ENNOUR Nassim

Encadré par :

M. Grégory Mounié

Année Universitaire 2024-2025

Table des figures

Table des matières

Introduction générale

Le développement des systèmes distribués est devenu essentiel pour répondre aux besoins croissants en termes de calculs complexes et de gestion de données massives. Ce projet s'inscrit dans ce contexte et vise à analyser les performances d'un système distribué en utilisant la plateforme Grid5000.

L'objectif principal est d'explorer les performances des systèmes distribués, en mettant l'accent sur la gestion des ressources, l'optimisation des temps de calcul et l'évaluation de la résilience face à des pannes. En utilisant Julia, un langage adapté aux calculs distribués, nous avons mis en œuvre plusieurs expérimentations pour évaluer les performances d'un système distribué en conditions contrôlées.

Ce rapport présente les méthodologies utilisées, les résultats obtenus et les enseignements tirés de cette expérience.

Chapitre 1

Présentation de la technologie choisie

1.1 Introduction au Projet

L'objectif de ce projet était de développer une version distribuée de GNU Make en utilisant le langage Julia et de valider son fonctionnement à l'aide d'expérimentations sur la plateforme Grid5000. Ce projet met en œuvre des concepts de calcul distribué pour exploiter efficacement les ressources d'un cluster.

1.2 Choix du Langage : Julia

Nous avons choisi Julia pour ses performances proches du C/C++ et sa syntaxe intuitive. Julia est particulièrement adapté aux calculs scientifiques et parallèles, avec des bibliothèques natives pour le calcul distribué, comme le module Distributed. Ce choix reflète notre volonté d'allier simplicité de développement et performances élevées.

1.2.1 Bibliothèques Utilisées

- **Distributed :** Pour la gestion des tâches distribuées et la communication entre les nœuds.
- **Statistics :** Pour effectuer des analyses statistiques sur les données obtenues, comme la moyenne et la variance des temps d'exécution.

Un modèle Master-Worker a été adopté pour simplifier la gestion des communications. Le maître (master) assigne les tâches, et les nœuds travailleurs (workers) exécutent les calculs.

Chapitre 2

Travail réalisé

2.1 Parsing des fichiers Makefile : MakefileParser

Le module **MakefileParser** est chargé d'analyser les fichiers Makefile pour extraire les cibles, leurs dépendances, et leurs commandes associées. Chaque cible est représentée par la structure `Target`, qui inclut un nom, une liste de dépendances et les commandes nécessaires à son exécution. La fonction principale, `parse_makefile`, lit et interprète les fichiers Makefile en construisant un dictionnaire de cibles. Ce dictionnaire sert de base pour générer le graphe des dépendances, permettant de traduire la structure logique du Makefile en un modèle exploitable par les autres modules.

2.2 Construction du graphe de dépendances : GraphGenerator

Une fois les cibles et leurs relations extraites, le module **GraphGenerator** est utilisé pour construire un graphe de dépendances. Ce graphe est représenté sous la forme d'un dictionnaire, où chaque tâche (`MyTask`) est associée à ses dépendances. La fonction `build_dependency_graph` prend en entrée le dictionnaire de cibles généré par le module de parsing et le convertit en un graphe organisé. Les tâches sont encapsulées dans la structure MyTask, qui inclut le nom de la tâche, ses commandes, son statut (`NOT_STARTED`, `IN_PROGRESS`, ou `FINISHED`), et des métadonnées additionnelles. Ce graphe est essentiel pour assurer que l'exécution des tâches respecte les dépendances définies.

2.3 Gestion des nœuds de calcul : Master

Le module **Master** se charge de la gestion des nœuds de calcul disponibles. En utilisant les fichiers système fournis par la plateforme Grid5000, la fonction `get_worker_hosts` identifie et liste tous les nœuds de calcul, tout en excluant le nœud maître réservé à la coordination des tâches. Cette étape garantit une répartition efficace des tâches sur les ressources disponibles, en exploitant pleinement les capacités de parallélisme offertes par

l'infrastructure.

2.4 Ordonnancement et exécution des tâches : Launcher

Le module **Launcher** orchestre l'exécution parallèle des tâches sur les différents nœuds. Il utilise le framework **Distributed.jl** pour distribuer les tâches en s'appuyant sur les directives `@everywhere` et `@spawnat`. La fonction `execute_tasks` attribue les tâches prêtes à être exécutées aux nœuds disponibles, en vérifiant systématiquement leurs dépendances avec la fonction `can_be_executed`. Une fois qu'une tâche est attribuée à un nœud, son statut passe de `NOT_STARTED` à `IN_PROGRESS`, puis à `FINISHED` une fois l'exécution terminée. Cette gestion fine assure que les ressources des nœuds sont exploitées de manière optimale, avec un suivi en temps réel des tâches en cours. De plus, l'utilisation de la macro `@time` permet de mesurer précisément les performances d'exécution.

Chapitre 3

Tests des performances

3.1 Tests de Latence et Débit (Ping-Pong)

Dans cette section, nous comparons les latences mesurées lors des tests de Ping Pong normal (sans interactions disque) et ceux avec I/O (lecture/écriture sur disque). Les résultats sont visualisés sous forme de courbes, permettant une interprétation claire des différences entre les deux configurations.

3.1.1 Méthodologie :

- **Ping Pong Normal :** Les messages sont échangés directement entre deux nœuds, sans interaction avec le disque.

- **Ping Pong avec I/O :** Les messages sont échangés, puis sauvegardés sur disque avant d'être renvoyés, simulant des scénarios d'application réelle où l'écriture disque est impliquée.

Les messages testés étaient de tailles croissantes (de 1 KB à 1 MB), avec plusieurs itérations par taille pour garantir des mesures fiables. Les latences mesurées ont été agrégées et représentées graphiquement.

3.1.2 Calcul de la latence :

Ces courbes comparent les latences mesurées pour les deux tests de Ping Pong : le Normal et celui avec IO.

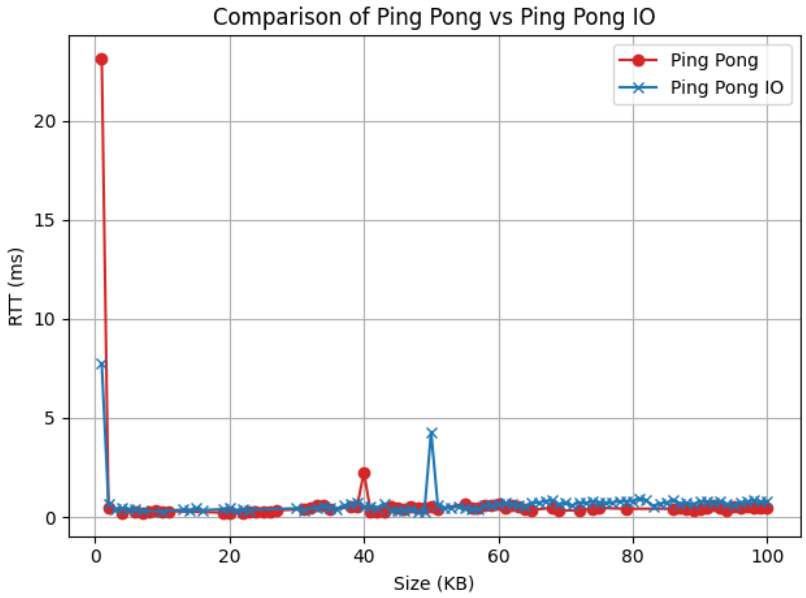

Fig. 3.1 : Comparaison latence Ping Pong normal VS IO

Ce graphique montre que la latence pour le test Normal reste faible et stable quelle que soit la taille des données, indiquant une communication fluide entre les nœuds sans surcharge notable. En revanche, la courbe correspondant au test IO met en évidence une augmentation plus importante de la latence. Cette augmentation reflète l'impact des opérations d'entrée/sortie sur les performances, entraînant des variations plus marquées et des pics de latence.

3.1.3 Calcul du débit :

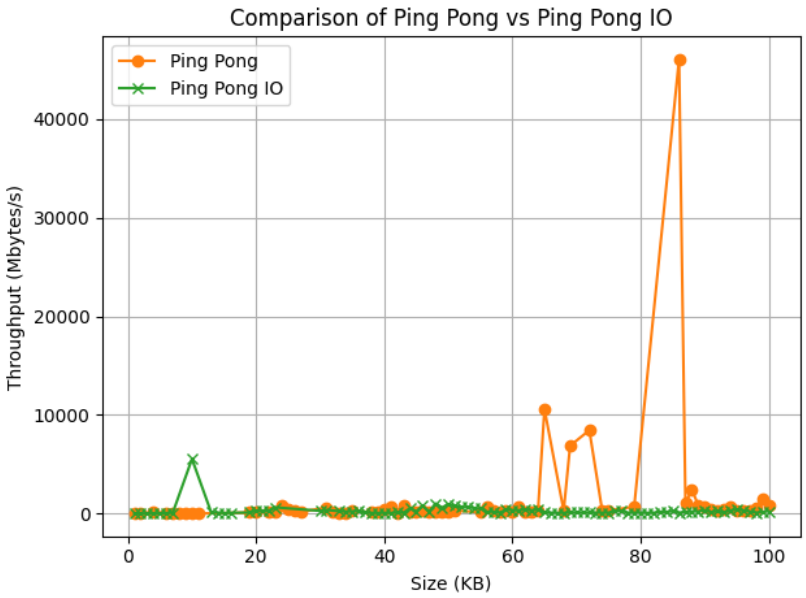

Fig. 3.2 : Comparaison débit Ping Pong normal VS IO

Pour le test Normal, le débit augmente régulièrement avec la taille des données, atteignant des valeurs plus élevées pour les grandes tailles, grâce à une utilisation optimale des ressources réseau. Cependant, quelques variations peuvent être observées pour certaines tailles, probablement dues à des fluctuations dans la latence initiale ou le traitement. Le test IO suit également une tendance à l'augmentation du débit avec la taille des données, mais avec des valeurs globalement inférieures à celles du test Normal. Cette différence s'explique par la surcharge introduite par les opérations d'entrée/sortie, qui limitent l'efficacité de la transmission des données.

3.1.4 Comparaison des Sites : Latence et Débit

Pour mieux visualiser les performances obtenues sur les différents sites, nous avons réalisé une comparaison des latences (en millisecondes) et des débits (en MBytes/s) mesurés. Les résultats sont présentés dans le graphique ci-dessous.

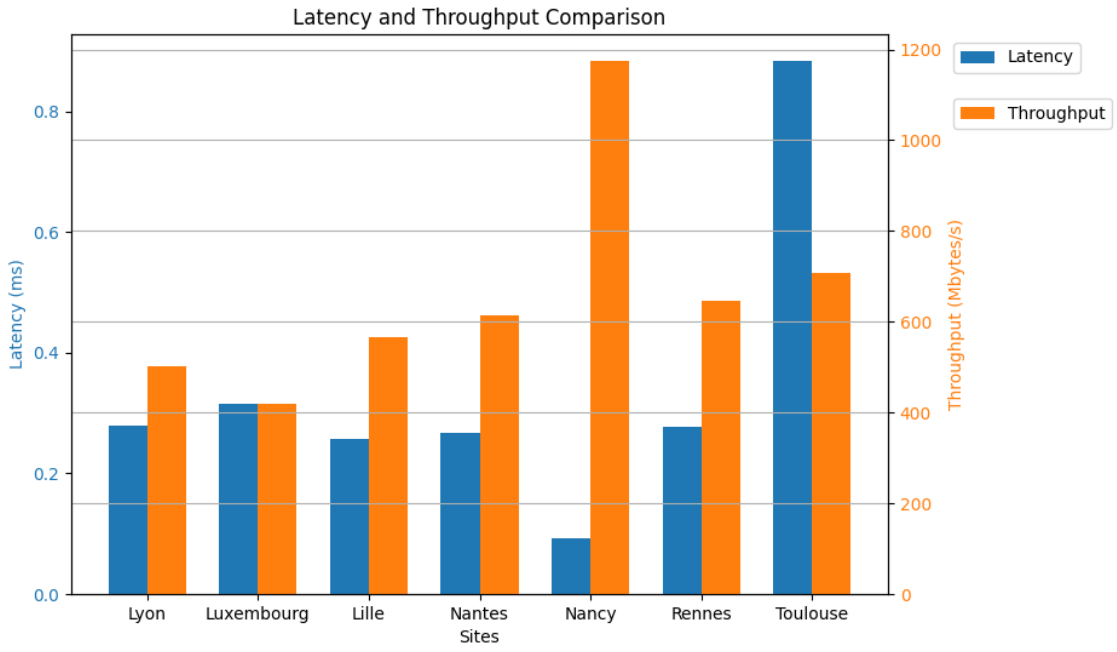

Fig. 3.3 : Comparaison entre les sites

- **Nancy** est clairement le site le plus performant, offrant une faible latence et un débit élevé, idéal pour les applications nécessitant des communications rapides et des transferts volumineux.

- **Toulouse**, bien qu'affichant une latence élevée, peut être adapté aux scénarios nécessitant un bon débit malgré une réactivité moindre.

- Les sites intermédiaires comme **Rennes**, **Lille** et **Nantes** montrent des performances équilibrées et pourraient constituer des choix polyvalents.

3.2 NFS vs SCP

3.2.1 Latence

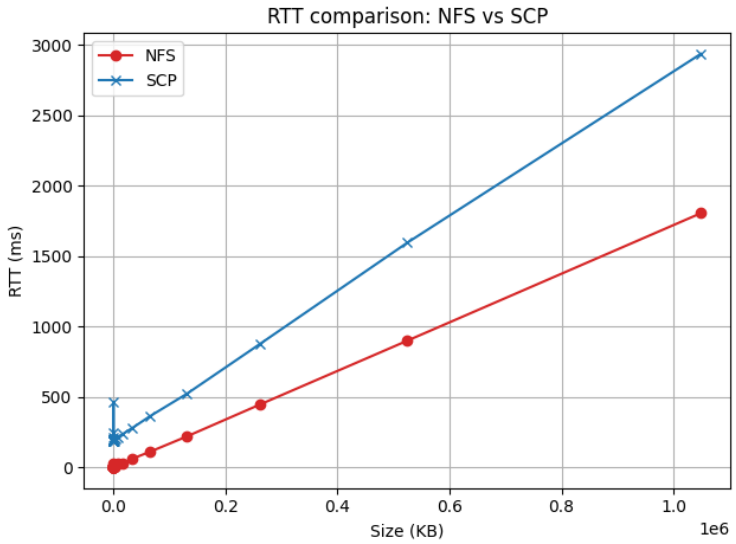

FIG. 3.4 : Latence : Comparaison entre NFS et SCP

- **NFS** a montré une latence légèrement plus faible, car les fichiers sont transférés de manière asynchrone.
- **SCP**, avec son chiffrement intégré, a enregistré une latence plus élevée, particulièrement notable pour les petites tailles de fichiers.

3.2.2 Débit

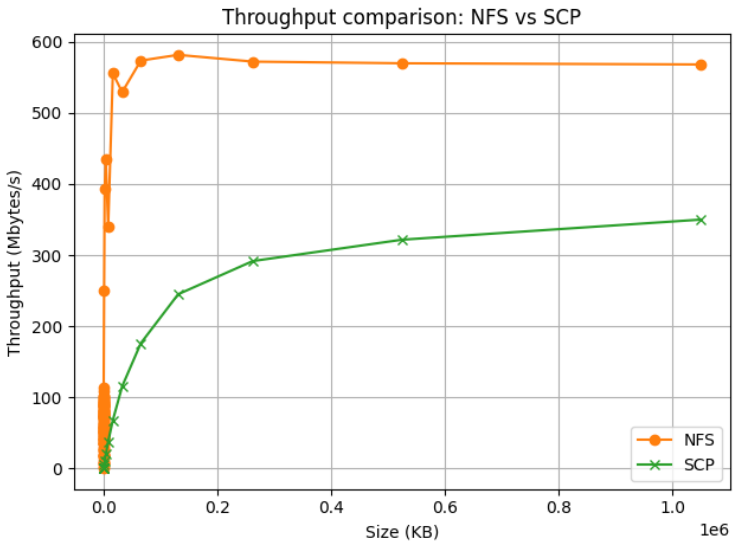

FIG. 3.5 : Débit : Comparaison entre NFS et SCP

- **NFS** a obtenu de meilleures performances pour les fichiers de grande taille (>10 MB), grâce à un transfert continu sans surcharge liée au chiffrement.
- **SCP** a montré des performances acceptables pour les petites tailles de fichiers, mais son débit est resté limité pour les fichiers plus volumineux, en raison de la surcharge cryptographique.

3.2.3 Choix de NFS

NFS se révèle être une solution plus performante pour les systèmes nécessitant des transferts fréquents et volumineux dans un environnement distribué.

3.3 Performances de make distribué

3.3.1 Description de l'expérience

Nous avons évalué les performances de notre système make distribué en testant le makefile "premier" (22 tâches) sur un nombre de machines variant de 1 à 20. Les métriques analysées incluent : le temps d'exécution total, l'accélération (rapport entre le temps séquentiel et le temps parallèle) et l'efficacité (accélération divisée par le nombre de machines). Ces mesures permettent d'identifier les gains liés à l'ajout de machines et les limites du parallélisme dans ce système distribué.

3.3.2 Temps d'exécution

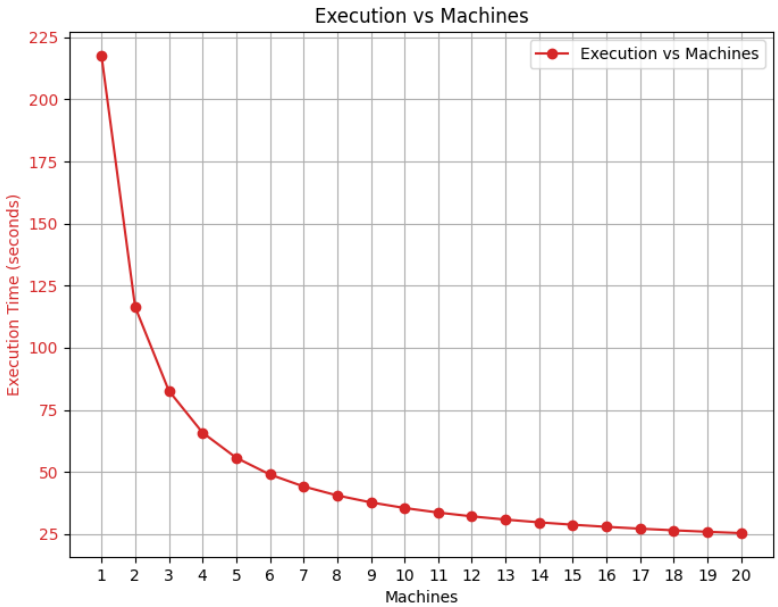

Fig. 3.6 : Temps d'exécution en fonction des machines

Quand on augmente le nombre de machines, le temps pour exécuter les tâches diminue fortement au début. Par exemple, avec 1 machine, cela prend 262,39 secondes, mais avec 10 machines, cela ne prend plus que 31,60 secondes, ce qui est une grosse amélioration.

Cependant, après 10 machines, ajouter plus de machines n'apporte presque plus de bénéfices. Parfois, cela peut même ralentir un peu, comme entre 10 machines (31,60 secondes) et 12 machines (33,60 secondes), à cause du temps nécessaire pour que les machines se coordonnent entre elles.

Au-delà de 18 machines, le temps reste presque le même, car le coût de la communication entre les machines devient plus important que le travail qu'elles peuvent effectuer. Cela montre qu'il y a une limite au nombre de machines utiles pour ce système.

3.3.3 Accélération

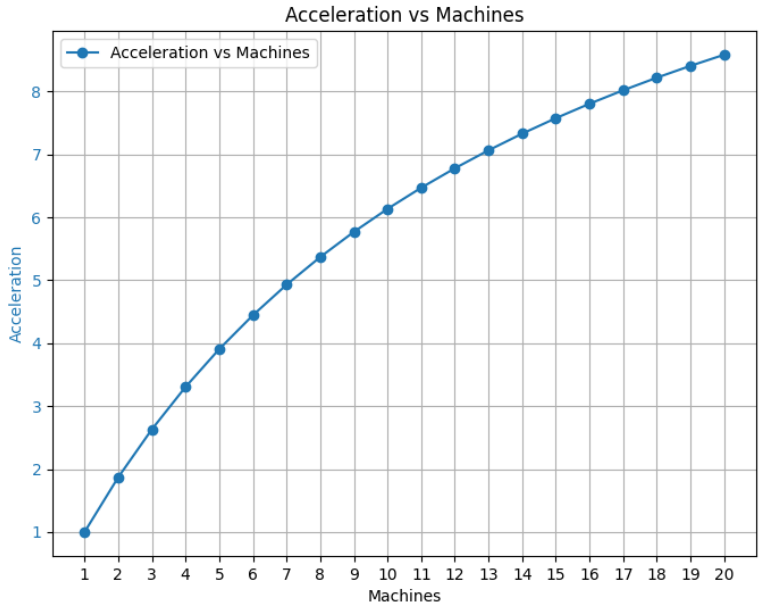

Fig. 3.7 : Accélération en fonction des machines

Au début, quand on augmente le nombre de machines jusqu'à 10, le système fonctionne très bien et accélère presque proportionnellement au nombre de machines, avec une accélération qui atteint 8,30. Cela montre que le parallélisme est très efficace à ce stade.

Après 10 machines, les choses deviennent moins régulières. Par exemple, avec 12 machines, l'accélération baisse un peu à 7,80, ce qui veut dire que les machines supplémentaires n'apportent pas autant de bénéfices. Mais avec 19 machines, il y a un pic inattendu à 13,08, probablement parce que les tâches étaient mieux réparties ou les conditions étaient favorables.

En résumé, le système fonctionne de manière optimale jusqu'à environ 10 machines, mais après, les gains deviennent imprévisibles et moins constants.

3.3.4 Efficacité

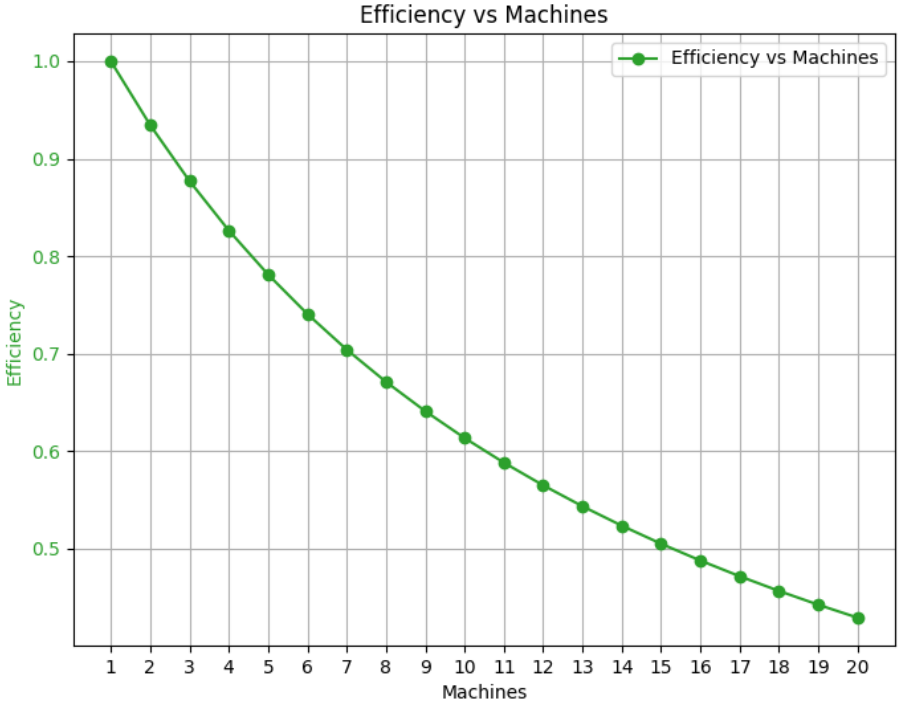

Fig. 3.8 : Efficacité en fonction des machines

Quand on augmente le nombre de machines, l'efficacité commence très bien. Jusqu'à 6 machines, presque toutes les machines sont utilisées de façon optimale, avec une efficacité entre 94% et 88%.

Mais à partir de 10 machines, l'efficacité baisse plus vite, tombant à 65% avec 12 machines, car il y a trop de coordination entre les machines. Après 14 machines, l'efficacité se stabilise autour de 62% à 68%, car il n'y a pas assez de tâches (22 au total) pour occuper toutes les machines.

En résumé, l'efficacité diminue quand on a plus de machines que nécessaire, ce qui est normal dans un système parallèle.

3.4 Modèle Théorique et Comparaison avec la Réalité :

3.4.1 Makefile premier :

Nous avons choisi le Makefile "premier" avec 22 tâches, dont 20 sont concurrentes, pour bien illustrer les défis de la parallélisation dans un environnement distribué.

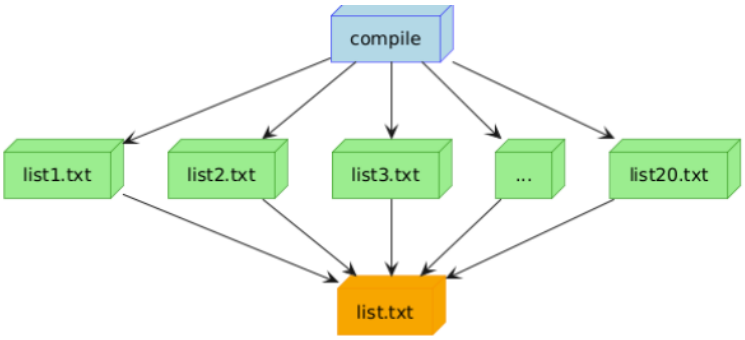

FIG. 3.9 : Makefile premier

3.4.2 Modèle Théorique :

Le modèle théorique repose sur la formule de majoration du temps d'exécution final d'un Makefile pour un nombre de machines m donné :

$$T\_\text{execution} \leq \frac{\sum T\_\text{tâches}}{m} + T\_\text{max}$$

Cette formule garantit que le temps total est borné et permet de comparer les performances théoriques et réelles.

3.4.3 Comparaison entre modèle et réalité :

3.4.3.1 Temps d'exécution

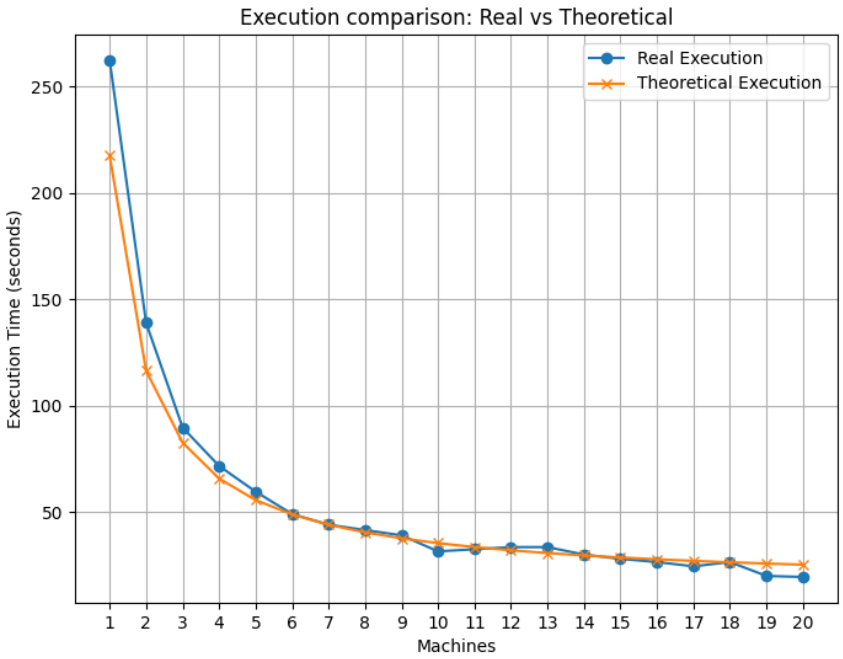

FIG. 3.10 : Comparaison du temps d'exécution

Au début, entre 1 et 10 machines, le temps d'exécution réel diminue plus vite que prévu, car les tâches sont bien réparties et la communication entre machines est efficace. Cela montre que le système utilise bien le parallélisme.

Mais après 10 machines, le temps réel arrête de beaucoup baisser à cause de problèmes comme la coordination entre les machines, alors que le modèle théorique continue de prévoir de petites améliorations. Cela montre les limites pratiques du parallélisme.

3.4.3.2 Accélération

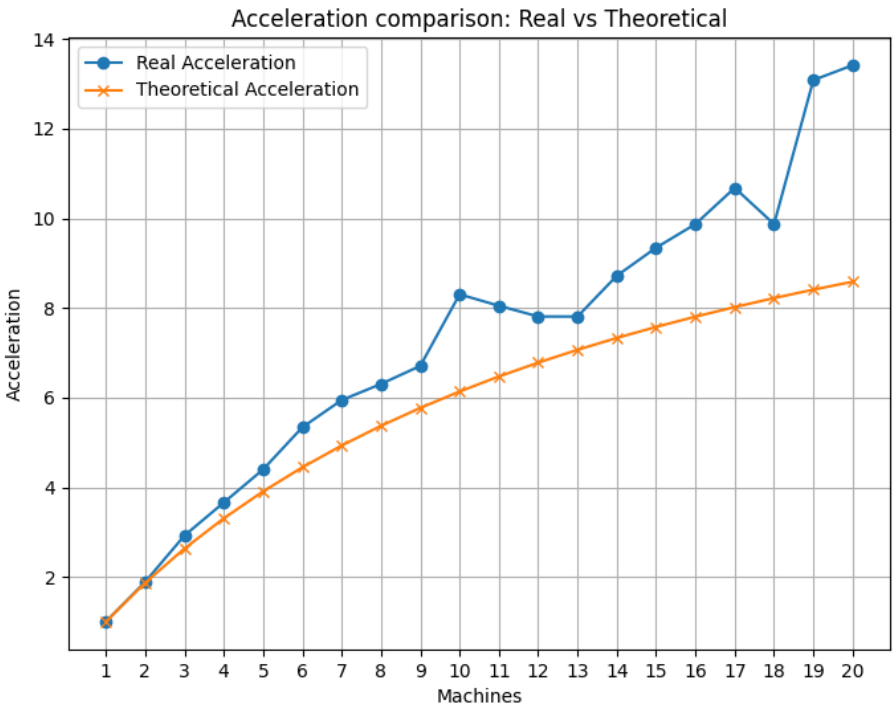

Fig. 3.11 : Comparaison de l'accélération

Dans la réalité, le système accélère beaucoup plus vite que prévu jusqu'à 6 machines. Par exemple, il va 5,3 fois plus vite, alors que le modèle théorique prévoyait seulement 2,24 fois.

Cela veut dire que le système utilise mieux les ressources comme les processeurs et la mémoire cache, ce qui améliore les performances de manière inattendue.

3.4.3.3 Efficacité

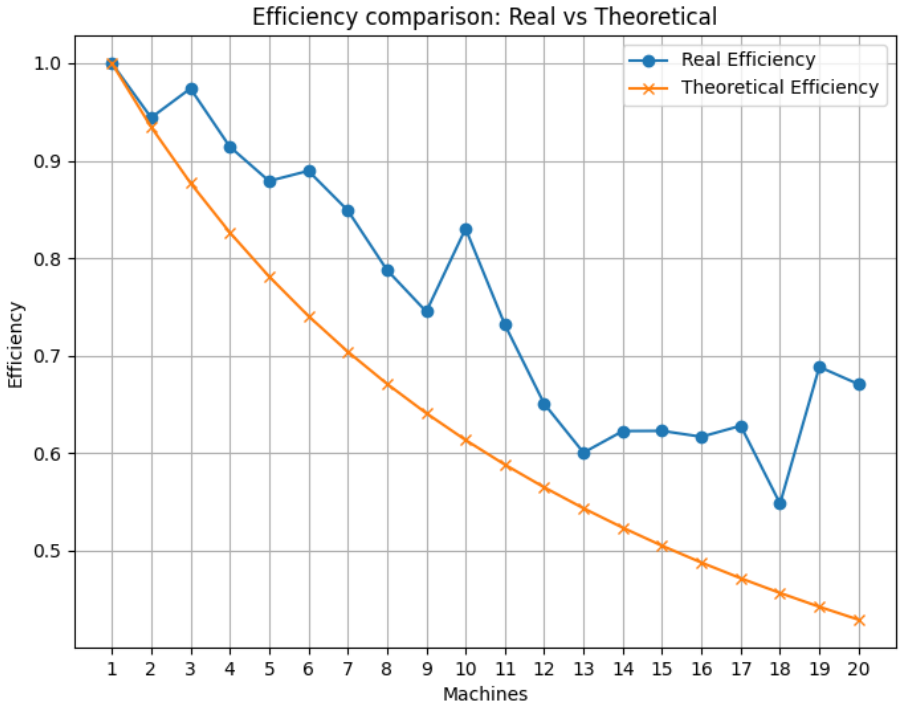

FIG. 3.12 : Comparaison de l'efficacité

Au début, avec 2 à 6 machines, le système utilise très bien les ressources, avec une efficacité réelle entre 85% et 94%, ce qui est meilleur que prévu.

Mais quand on ajoute plus de machines, l'efficacité baisse plus vite que ce que le modèle avait prévu, car le système commence à avoir des difficultés à gérer la coordination entre les machines.

Chapitre 4

Déploiement : réalisation de la commande gridmake

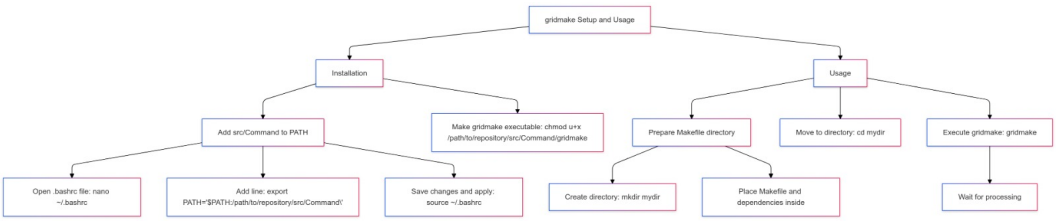

FIG. 4.1 : Réalisation de la commande gridmake

Pour installer, on ajoute le chemin src/Command à la variable PATH, on modifie le fichier .bashrc pour inclure cette configuration, et on applique les changements avec source, en rendant la commande gridmake exécutable. Pour utiliser, on place le fichier Makefile et ses dépendances dans un répertoire, on accéde à ce répertoire, puis on exécute la commande gridmake pour traiter le fichier Makefile.

Conclusion

Conclusion générale

Ce projet a permis de développer une version distribuée de GNU Make en utilisant Julia, afin d'exploiter les capacités des infrastructures de calcul comme Grid5000. Nous avons conçu un parsing des fichiers Makefile, construit un graphe de dépendances, et implémenté un modèle Master-Worker pour distribuer les tâches efficacement sur plusieurs nœuds.

Les tests de performance ont démontré que notre solution améliore le temps d'exécution en fonction du nombre de machines disponibles, tout en identifiant les limites liées aux ressources réseau et au parallélisme.

Ainsi, ce projet valide la faisabilité d'une solution distribuée pour GNU Make et ouvre la voie à des améliorations futures, telles que l'optimisation de l'ordonnancement des tâches et des communications réseau.

Bibliographie

[1] **Documentation Grid5000 :**
https://www.grid5000.fr/w/Getting_Started

[2] **Documentation Julia :**
https://docs.julialang.org/en/v1/