Grenoble INP - Ensimag

Ecole Nationale Supérieure d'Informatique
et de Mathématiques Appliquées

Rapport
-
Make distribué
-
MPI & OpenMP

Systèmes distribués

3e année - Filière ISI

Table des matières

1 Introduction **2**

2 Algorithme utilisé **2**

3 Architecture & Exécution du programme **3**
3.1 Transfert des données/fichiers entre machines 4

4 Expériences & tests **4**
4.1 Préparation du matériel des expériences 4
4.2 Tests 4

5 Analyse de résultats **5**
5.1 Synthèse 5

Table des figures

1 Structure d'arbre 2
2 Etapes du Make 2
3 Master-workers 3
4 Légende des courbes 6
5 Test : premier 6
6 Test : premier_small 7
7 Test : simu_gif 7

1 Introduction

Ce document est un rapport du projet Make distribué utilisant les bibliothèques MPI et OpenMP. Il fournit toutes les informations nécessaires concernant l'algorithme utilisé pour réaliser la version distribuée du Make, l'architecture du système exécutant cette version ainsi que des expériences réalisées sur Grid5000.

2 Algorithme utilisé

La version distribuée du Make telle que nous avons conçue passe par trois étapes principales. En effet, notre programme prend en entrée un fichier Makefile, le parcourt pour extraire pour chaque règle, ses dépendances et sa commande. Ce triplet est stocké dans un tableau. KLa deuxième étape consiste à parcourir la liste des mots extraits et de créer pour chaque triplet un noeud Node, qui est une structure de données contenant les informations d'une règle (nom, dépendances, commande, etc). A partir de ces noeuds, on crée une structure d'arbre renversé, c'est à dire que c'est un arbre à plusieurs racines (qui représentent les règles sans aucune dépendance) et à une seule feuille (la première règle du Makefile) La troisième étape consiste à parcourir l'arbre pour exécuter les commandes en

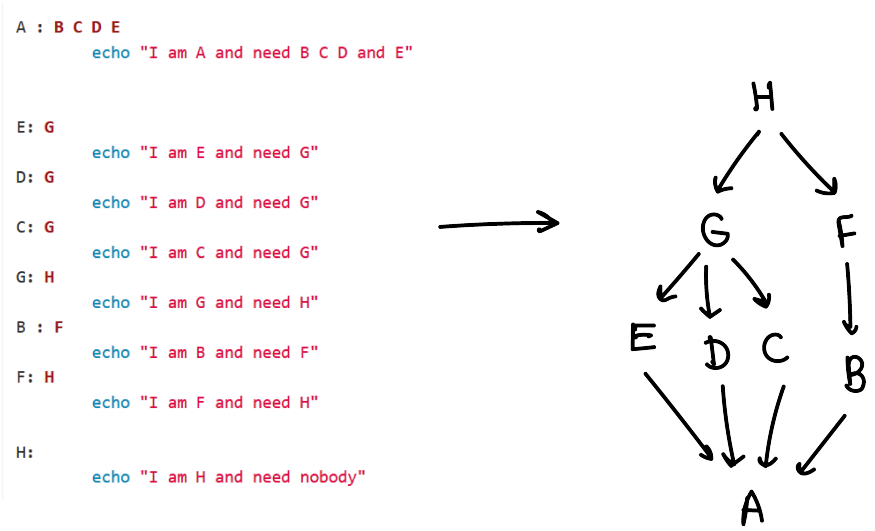

FIGURE 1 – Structure d'arbre

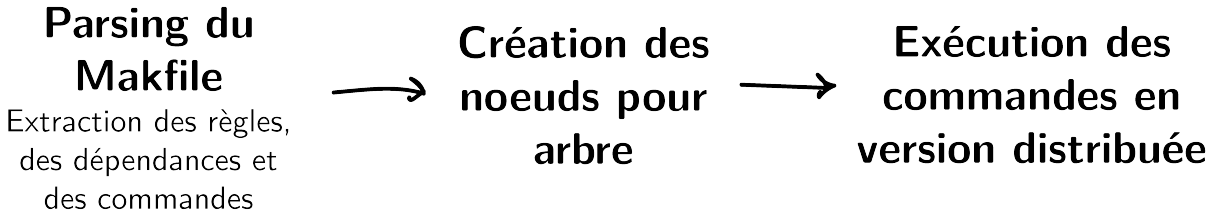

FIGURE 2 – Etapes du Make

respectant les dépendances. En effet, la structure Node contient un attribut nommé parents qui est un tableau de tous les noeuds dont le noeud en question est une dépendance. Par exemple, dans la figure 1, le noeud H a pour parent G et F et le noeud F a un seul parent qui est B, par contre le noeud G a pour parents C, D et E, etc. Lors du parcours de l'arbre, un noeud est prêt à être exécuté si et seulement si toutes ses dépendances sont exécutées. Pour éviter d'ajouter un autre attribut

dépendances dans la structure Node, on ajoute un attribut entier **isReady** qui stocke le nombre de dépendances déjà exécutées et dès que ce nombre atteint le nombre total de dépendances cela veut dire que le noeud est prêt à être exécuté. Par exemple, H n'a pas de dépendances : c'est donc un noeud prêt. Dès qu'il sera exécuté, on incrémente l'attribut `isReady` de 1 pour tous ses parents, c'est-à-dire G et F. G a une seule dépendance qui est H et `G.isReady = 1`, donc G est prêt et on incrémente l'attribut `isReady` de E, D et C de 1, etc. Supposons que E, D et C sont exécutés et que B non. Alors `A.isReady = 3` alors que le nombre de dépendances de A est 4, donc A doit attendre que B soit exécuté pour qu'il soit prêt.

3 Architecture & Exécution du programme

Nous avons adopté une architecture de **Master-Workers** pour exécuter le programme en version distribué. En effet, parmi tous les processus disponibles, un seul processus est élu pour être le maître et tous les autres sont des esclaves. C'est le maître qui est chargé de la première et deuxième étape. Il est également chargé de vérifier si un noeud est prêt ou non et s'il est prêt, il envoie la commande à l'esclave qui l'exécutera. Après exécution, l'esclave doit envoyer les résultats au maître. Ce sont les messages MPI (généralement deux messages `MPI_Send` & `MPI_Recv`) qui permettent la communication entre les processus esclaves et le processus maître d'une part, et entre les processus d'une machine et d'une autre d'autre part. La figure 4 montre un scénario d'échange de messages

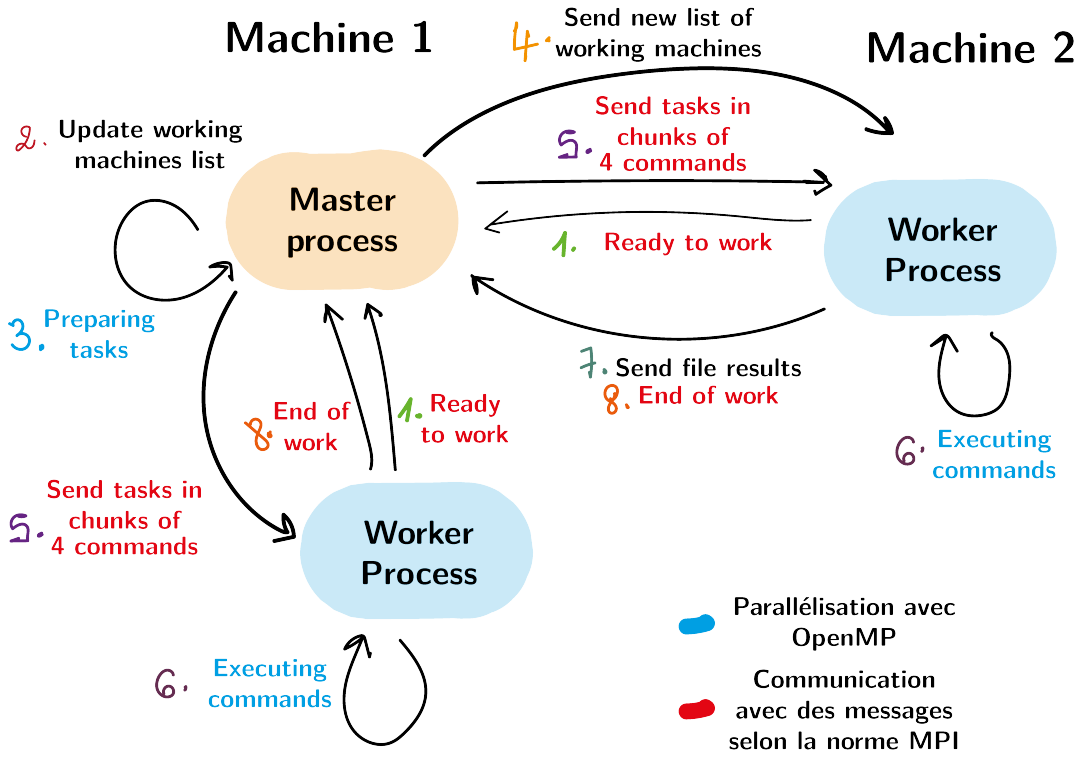

FIGURE 3 – Master-workers

et d'actions lors de communication entre processus et machines. En effet, après le déploiement de l'application, les esclaves envoient des messages MPI au maître qui est, au début à l'écoute de tout type de messages provenant de tout type d'émetteurs, pour l'informer qu'ils sont prêts à travailler. Le maître met à jour une liste contenant les machines où au moins un seul esclave travaille. Le maître prépare également des blocs de données qui sont les commandes des noeuds qui sont prêtes à être exécutées. Il envoie donc ces blocs de commandes aux esclaves prêts à travailler avec la liste de machines mise à jour aux esclaves contenus dans d'autres machines. Les esclaves reçoivent les blocs

de commandes et les exécutent parallèlement grâce à OpenMP. Dès qu'ils finissent, si l'esclave est dans une autre machine que le maître, il consulte la liste de machines et envoie les fichiers qu'il a produits à toutes les machines pour que d'autres esclaves s'en servent dans le futur. Les esclaves informent le maître avec un message MPI comme quoi ils ont achevé leur travail et qu'ils sont prêts à retravailler. Et le cycle reprend jusqu'à ce que la dernière commande soit exécutée. Le schéma suivant résume en gros ce que je viens de dire et là en rouge sont marqués tous ce qui se fait grâce à MPI, MPI_Send et MPI_Recv. Et en bleu, tout ce qui s'exécutent en parallèle avec OpenMP.

3.1 Transfert des données/fichiers entre machines

Dans la version initiale de notre application distribuée,le transfert des fichiers entre machines se faisait directement à partir du programme principal. Ce qui signifie que la préparation des commandes scp se fait dans le programme et en langage C. En revanche, lors de la création de la structure `data_send` (une structure à ajouter à MPI pour envoyer au maître la liste des nodes déjà exécutés et le nom de la machine où le travail a été fait. Avant d'envoyer cette donnée, il faut allouer l'espace mémoire en se servant d'une variable prédéfinie de MPI nommée : `MPI_MAX_PROCESSOR_NAME`. Ce qui cause parfois l'apparition de caractères spéciaux à la fin du nom de la machine ce qui cause une erreur lors du transfert vers la machine au nom erroné. Pour remédier à ce problème, nous avons ajouté un script python `working_machine.py` qui s'occupera du travail de transfert, et dans le programme pricipal, on lance juste une commande pour exécuter le script python.

4 Expériences & tests

4.1 Préparation du matériel des expériences

Les expériences sont effectuées sur Grid5000. Avant de commencer toute expérience, il est primordial de préparer le matériel sur Grid5000, c'est-à-dire réserver les machines qui contiendront les processus. Dans notre cas, nous nous sommes servis de 8 machines de deux clusters différents pour 3 expériences. La première expérience est réalisé sur le cluster `grisou` de Nancy avec 4 machines à 2 CPU et 8 cores par CPU chacune, ce qui fait au total 16 processus par machine, donc 64 processus. La deuxième et la troisième expériences sont réalisées sur le cluster `chetemi` avec 4 machines à 2 CPU et 10 cores par CPU chacune, ce qui fait au total 20 processus par machine, donc 80 processus.

4.2 Tests

Les expériences consistent à faire varier le nombre de processus à utiliser lors de l'exécution de l'application distribuée et de voir l'évolution du temps d'exécution, de l'accélération et de l'efficacité du programme. La courbe orange représente le temps d'exécution, la courbe verte, l'accélération, c'est-à-dire le gain de performance entre le système à p processus et le système à 2 processus uniquement. Et enfin, la courbe rouge représente l'efficacité qui est une mesure résumant la façon par laquelle un système amélioré utilise-t-il les ressources disponibles.
Les trois expériences concernent trois Makefiles : `premier`, `premier_small` et `simu_pi`. Les deux premiers Makefile permettent d'obtenir la liste de nombres premiers inférieurs à une borne supérieure choisie (pour `premier` la borne est grande et pour `premier_small` petite). Le troisième Makefile permet de créer des fichiers gifs visualisant une simulation de PI selon la méthode de Monte-Carlo. La lecture des courbes se fait en référence à la légende attachée aux courbes.

5 Analyse de résultats

Pour le test `premier`, on remarque que le temps d'exécution décroît de façon exponentielle et rapide pour une variation de 2 à 4 processus. En effet, sachant que pour ce cas, 1 seul règle contient des dépendances, cette décroissance est due au faite que pour 2 processus, 1 seul esclave travaille donc il reçoit des blocs de 4 commandes, les exécute parallèlement puis en reçoit d'autres, etc, jusqu'à ce qu'il exécute toutes commandes qui sont au nombre de 21. Donc il a besoin de travailler au moins $21/4 + 1 = 6$ fois. Alors que pour 3 processus, 2 esclaves travaillent donc chacun travaille au plus 3 fois. Et pour 4 processus, chacun travaille au plus 2 fois... Donc, à partir d'un certain seuil, le temps d'exécution devient quasiment stable vu que chaque processus travaille une seule fois et il y a un seul qui se charge à la fin de la dernière règle. En dépassant le nombre total de slots disponibles pour une machine, on est censé utiliser des processus d'une autre machine donc s'envoyer les fichiers produits entre machines. Ceci n'affecte pas trop le temps d'exécution. Il suffit juste de voir que la courbe de l'accélération ne décroit pas même pour de grands nombres de processus. En effet, ceci est dû au fait que le temps mis pour envoyer les fichiers aux machines est quasiment négligeables devant la charge du travail.
On peut constater la différence avec le test premier_small. Le principe du Makefile dans ce cas est le même que premier sauf que les bornes sont plus petites. La charge du calcul sera donc plus petite. Ce qui explique une décroissance de la courbe pour 2 à 4 processus puis une croissance pour de grands nombres de processus car le transfert de fichiers entre machines n'est plus négligeable devant la charge du travail. On peut remarquer qu'il existe deux instants où la courbe croît brusquement : le premier pour la valeur 17 car c'est à ce moment-là qu'on passe d'une machine à deux et le deuxième pour la valeur 51 en passant de trois à quatre machines.
Pour le dernier test, il s'agit d'une simulation de la valeur de PI à partir de points dans un cercle suivant la méthode de Monte-Carlo. En effet, le Makefile permet de générer des gifs à partir de succession d'images et pour chaque image le nombre de points de simulation de la valeur de PI augmente pour avoir une valeur bien précise. Dans ce test, la charge du travail devient négligeable par rapport au transfert de fichiers produits, car à chaque exécution du code, plus de 20 images sont générées et donc transférées entre machines si on utilise un grand nombre de processus. En effet, contrairement au test précédent, premier_small où la croissance de la courbe du temps d'exécution est visible, ici, cette croissance n'est pas bien claire. Mais on peut s'en servir de la courbe de l'accélération pour déduire cela. En effet, pour un nombre considérablement grand de processus, l'accélération décroît et ceci en trois phases. Elle décroît brusquement pour la valeur 17 (passage de 1 à 2 machines) et la valeur 51 en passant à quatre machines.

5.1 Synthèse

Les trois tests montrent trois cas différents pour deux aspects entrant en jeu dans l'exécution d'un programme distribuée. Ces deux aspects sont la charge du travail et le transfert de fichiers entre machines. Le premier cas, le temps de calcul est un grand O du temps de transfert de fichiers. Le deuxième cas, les deux temps sont très proches et le troisième cas, le temps de calcul est un petit O du temps de transfert de fichiers.

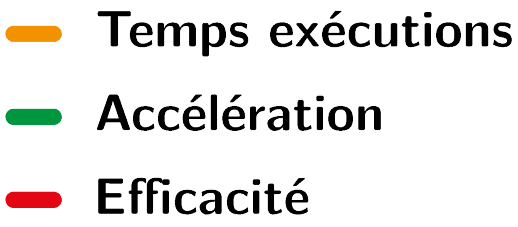

FIGURE 4 – Légende des courbes

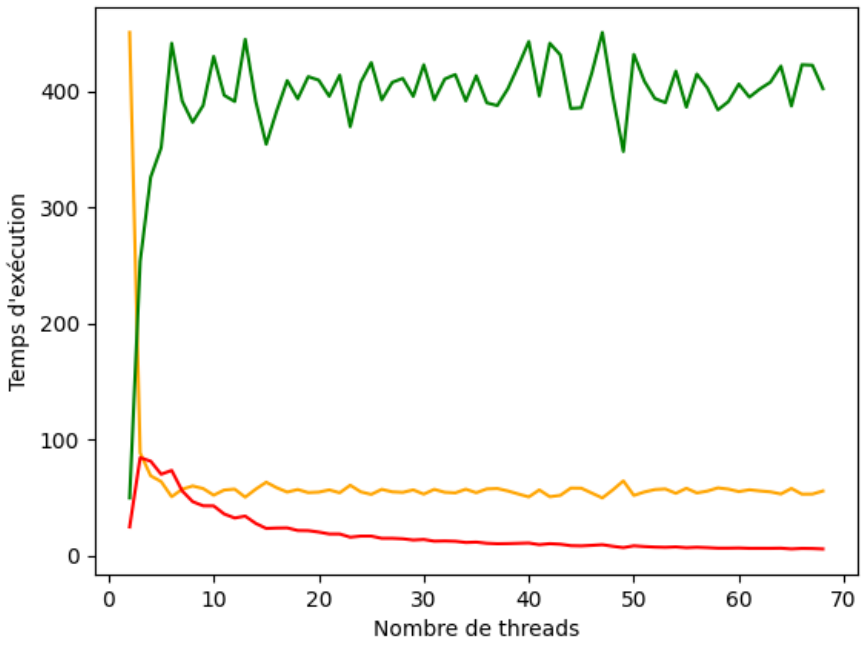

FIGURE 5 – Test : premier

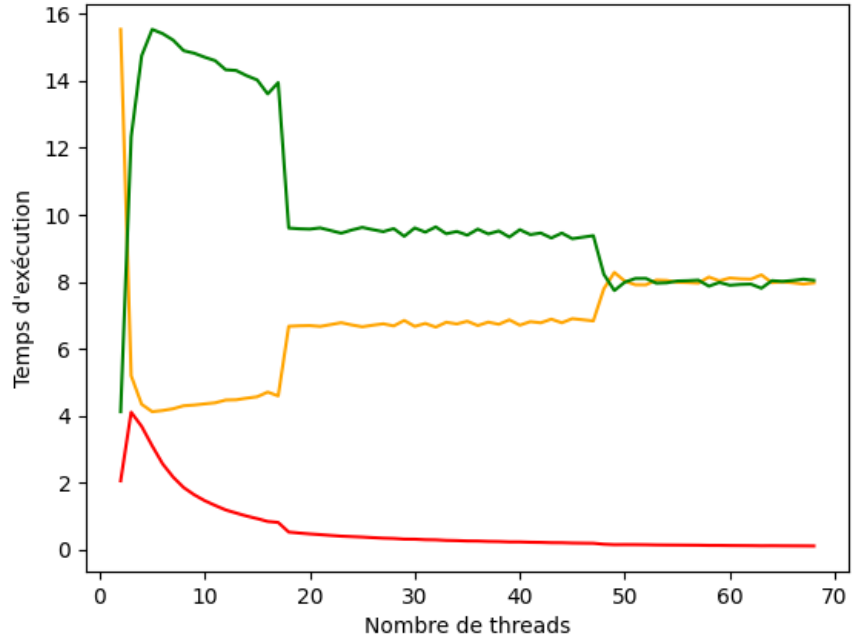

FIGURE 6 – Test : premier_small

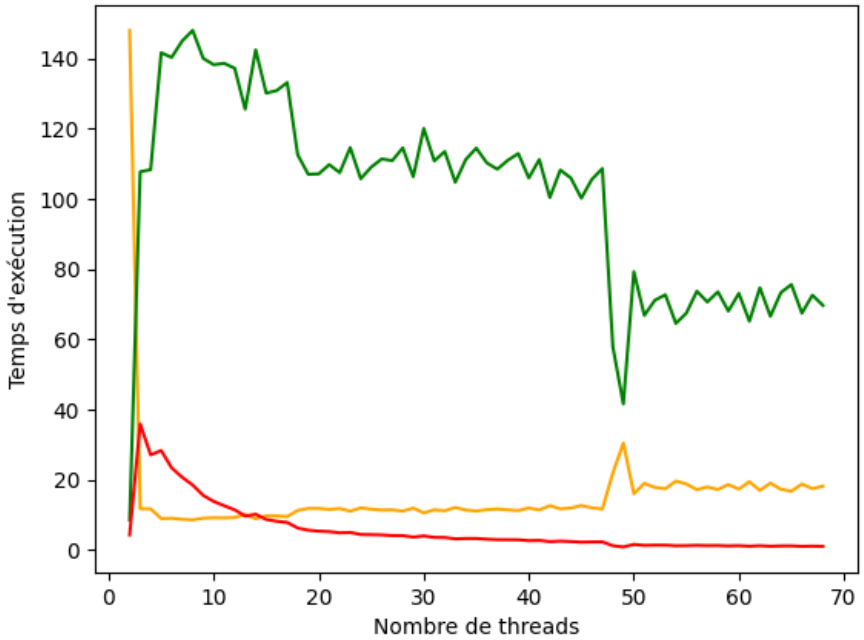

FIGURE 7 – Test : simu_gif