Rapport

21 janvier 2013
Projet d'Algorithmique Parallèle et Distribuée:
- Solveur de problème SAT-

Adlen AFANE
Edouard LE GAC de LANSALUT
Nicolas SCHMIDT

1. Fonctionnement de l'algorithme

a. Schéma d'ensemble

Nous avons décidé d'implémenter une architecture maitre-esclave. Le maître gère une pile de travaux à envoyer aux esclaves. Les messages du maître aux esclaves va dépiler cette liste, alors que les message des esclaves vers le maître vont en général agrandir la pile.

i. Le maitre

Le maître ne joue aucun rôle "algorithmique" dans le calcul de la solution a proprement parler: il n'est "que" le chef d'orchestre qui permet aux esclaves de travailler.

Le maître s'assure donc que tant que le problème n'est pas fini, les esclaves libres reçoivent du travail.

Conscients que les nombres de messages gérés entre les différents processus est un facteur critique d'efficacité de l'algorithme, nous avons dès le début stocké dans une variable le nombre de travaux par message à envoyer aux esclaves pour pouvoir ensuite étudier l'influence de ce paramètre.

ii. Les esclaves

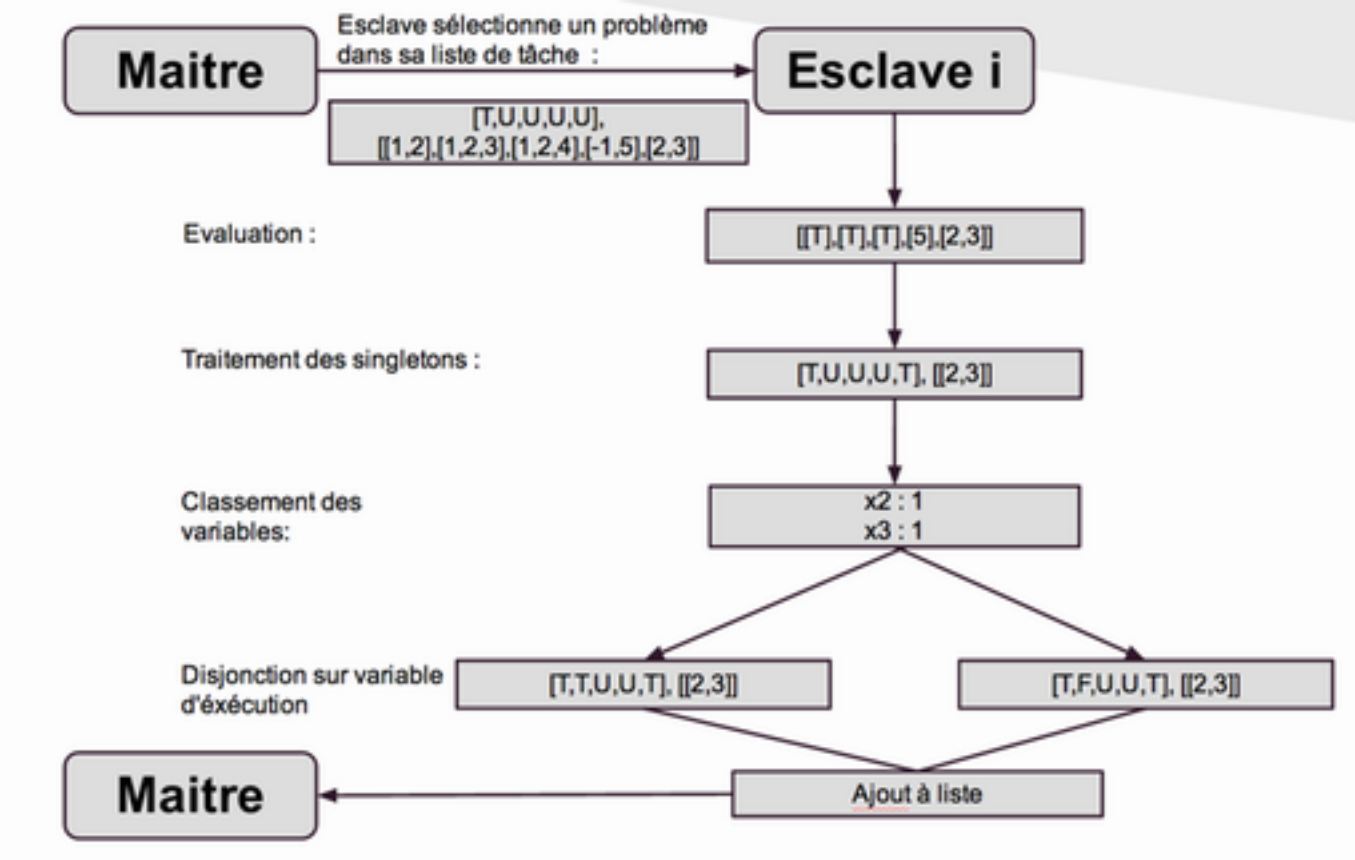
Ce sont eux qui effectuent le travail.

Un esclave reçoit un couple (jeu de variables, forme normale conjonctive). L'esclave va d'abord simplifier l'expression de la forme normale conjonctive en fonction du jeu de variables donné.

Une fois la forme normale conjonctive réduite à sa plus simple expression permise par le jeu de variable, il existe trois cas de figure:

- L'esclave teste si toutes les clauses du problème sont vérifiées (on a abouti à une solution),
- ou si cette expression contient une clause fausse, (dans ce cas là on rend compte au maître que cette branche de recherche de solution ne peut pas conduire à une solution).
- Sinon l'esclave choisit la variable la plus propice à une disjonction (classement basé sur la fréquence d'apparition des variables et leur répartition entre son expression "xi" et sa négation "xi barré"), et génère deux nouveaux problèmes issus de cette disjonction. Il les stocke alors dans sa liste de traitement qu'il enverra au maître, qui les ajoutera dans la pile de travaux à traiter à envoyer aux esclaves.

Un exemple de fonctionnement de l'esclave



b. Problèmes techniques rencontrés

i. Installation d'une bibliothèque mpi pour Python

Nous avons choisi d'utiliser le langage Python parce que c'est celui que nous connaissons le mieux, et dans lequel nous souhaitons progresser. Lorsque nous avons vu qu'il y avait plusieurs bibliothèques mpi implémentées en python, nous avons franchi le pas sans hésiter.

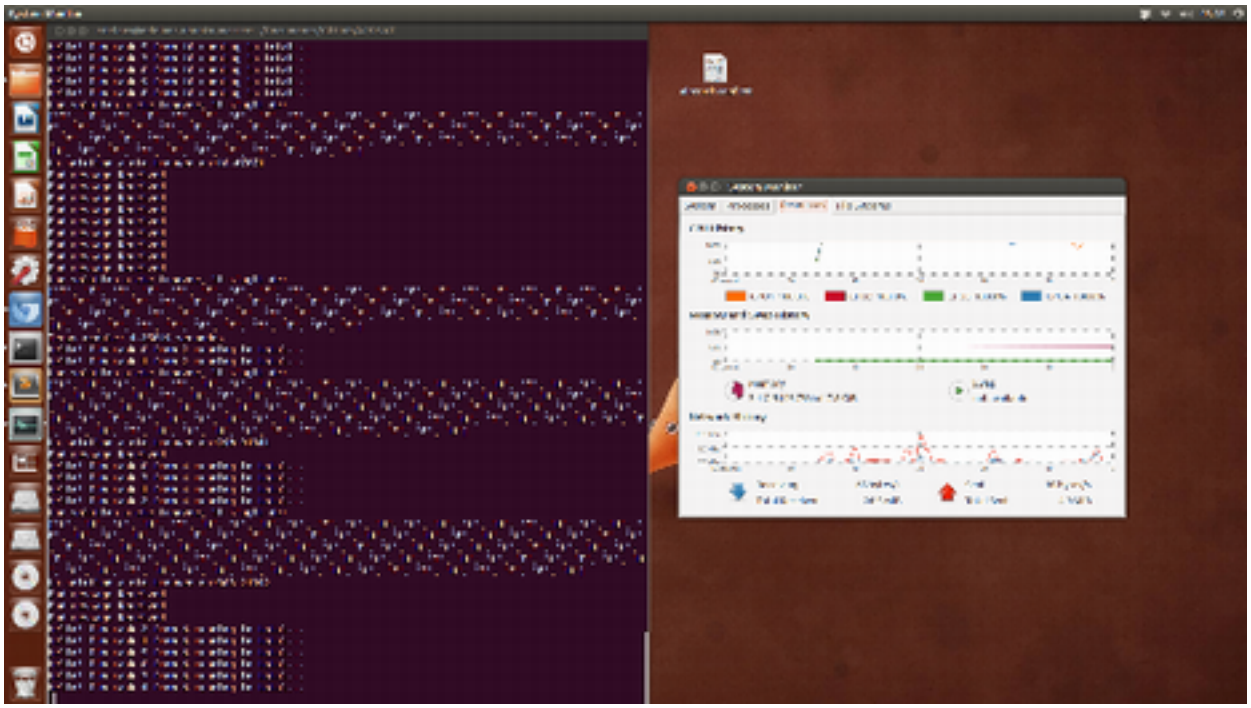
Cependant, l'installation (sur Windows) et le manque de documentation n'ont pas rendu la chose facile. Heureusement, la personne responsable de la bibliothèque (Lisandro Dalcin) s'est montrée très disponible (sur les forums et même en répondant à un mail que nous lui avons adressé !), ce qui nous a permis d'arriver à nos fins.

ii. Communication asynchrone

Au départ, notre maître avait pour objectif de recevoir des données de manière asynchrone. Il devait vérifier si un esclave lui avait envoyé un message et le cas échéant le traiter, et sinon passer à la suite. Nous avons eu des problèmes pour gérer la mémoire et les buffer avec Python et avons adopté une solution moins performante mais fonctionnelle. Le maître envoie des messages à tous les esclaves possibles et récupère ensuite ces messages de manière séquentielle (il récupère le message du processus n, puis celui du processus n+1, etc.).

Les travaux étant de tailles équivalentes, et en supposant que les processus disposent de la même puissance de calcul, on peut en déduire que le temps de traitement de chaque message émanant du maître est sensiblement le même pour tous les esclaves, et donc au final, le parallélisme peut toujours garder un certain avantage.

Cependant il est évident que cette méthode nous fait perdre en efficacité du parallélisme et donc en performance, il s'agit d'un des points à améliorer pour une future version.



iii. Choix de l'architecture Maître / Esclave

Nous avons fait le choix d'une architecture Maître / Esclave car cela permettait plus de facilité dans la gestion des messages. En effet, une structure sans maître serait potentiellement plus performante mais la communication devient plus complexe: chaque processeur doit pouvoir traiter un problème et quand le travail devient trop important en déléguer une partie à un autre. Pour cela chacun doit tenir à jour une liste des processeurs libres, ce qui implique beaucoup de messages échangés à chaque fois qu'un processeur se libère ou reçoit du travail.

iv. Gestion de la file d'attente des problèmes

En première approche, nous avons choisi, de gérer la file d'attente en FIFO, ce qui s'apparente à un parcours en largeur de l'arbre. Nous avons également implémenté une gestion aléatoire de la file d'attente; le maître sélectionne dans ce cas au hasard un des problèmes dans la file et l'ajoute au batch en cours. Nous avons également implémenté une file d'attente fonctionnant en attribuant à chaque problème une priorité basée sur le nombre de variables restantes.

v. Alimentation des statistiques avec un script de test

Nous avons voulu implémenter un script pour faire tourner notre algorithme automatiquement sur plusieurs fichiers d'entrée et en faisant varier les paramètres de calculs

(nombre de processus, taille des messages MPI). La gestion du script de script a donné lieu à quelques pertes de cheveux.

2. Test de performances

Pour tester notre algorithme, nous avons commencé par le tester sur des formes normales conjonctives que nous avions créées nous même.

Une fois notre algorithme assez robuste, il a fallu le mesurer à des problèmes plus complexes. Ceux-ci n'étaient pas forcément plus difficiles à créer, mais en revanche, il était bien compliqués de prévoir leurs résultats.

Nous avons alors trouvé un site internet avec un solveur Sat en ligne: http://boolsat.com/ , ce qui nous a permis de vérifier quelques problèmes, mais ne nous permettait toujours pas de passer à une phase de test plus "industrielle".

Nous nous sommes donc finalement appuyés sur des sets de problèmes[1] mis en ligne par UBC (University Of British Columbia). L'intérêt était de pouvoir disposer d'un certain nombre de problèmes importants avec leurs "satisfaillabilité". Nous avons effectué des tests sur:

	Variables	**Clauses**
uf20-91	20	91
uf50-218	50	218
uuf50-218	50	218
uf75-325	75	325
uuf75-325	75	325
uf100-430	100	430
uuf100-430	100	430

Cette banque de problèmes nous a permis de valider la correction de notre algorithme. Nous l'avons ensuite utilisée pour en étudier les performances: nous avons modifié le code de

[1] http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

notre algorithme pour qu'il enregistre à chaque lancement des statistiques de résolution dans un fichier. Cela nous a permis de disposer d'environ 1800 tests pour étudier les performances de l'algorithme.

3. Performances de l'algorithme

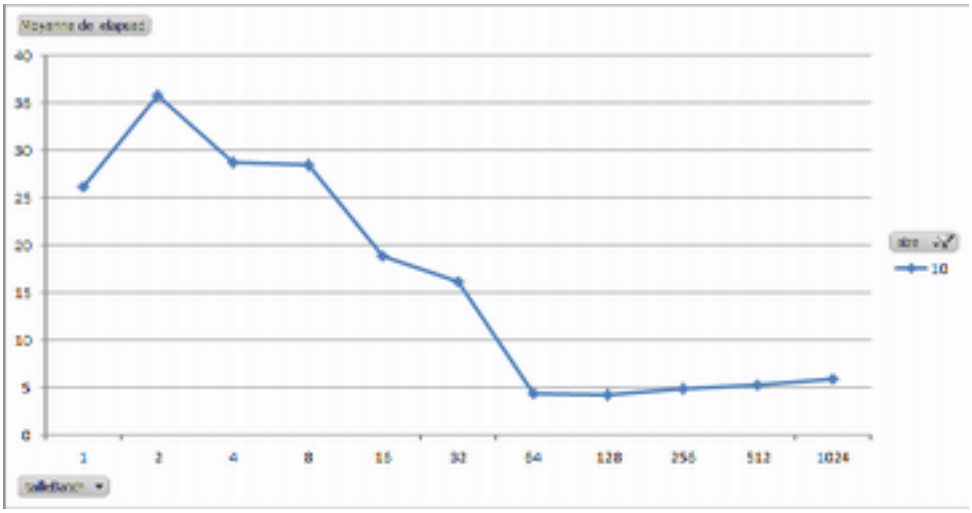
a. Distribué VS sequentiel

Une de nos principales préoccupations était de savoir si notre algorithme distribué était plus performant qu'un algorithme séquentiel... Nous avons donc adapté notre algorithme distribué pour construire un algorithme séquentiel (utilisation des mêmes fonctions et de la même logique).

Les résultats obtenus avec l'algorithme séquentiel sont utilisés comme cas de base dans les graphes qui en ont l'utilité.

b. Variation de la taille des messages maître -> esclaves

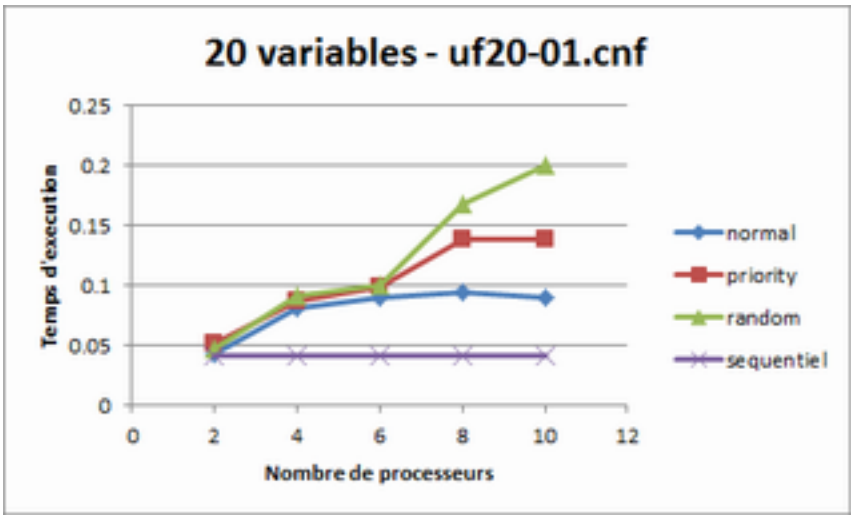
On observe une taille de batch optimale, celle-ci correspond à l'optimum entre temps de communication et temps de travail des esclaves. Voici les résultats obtenus:

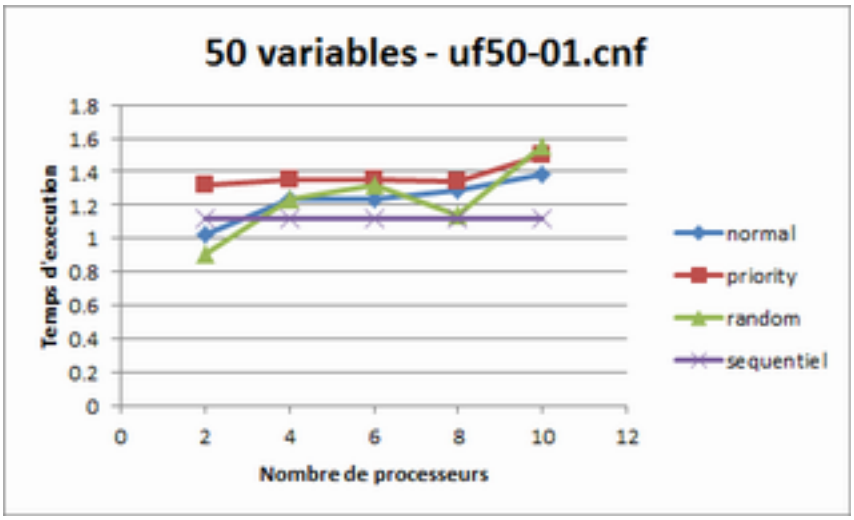


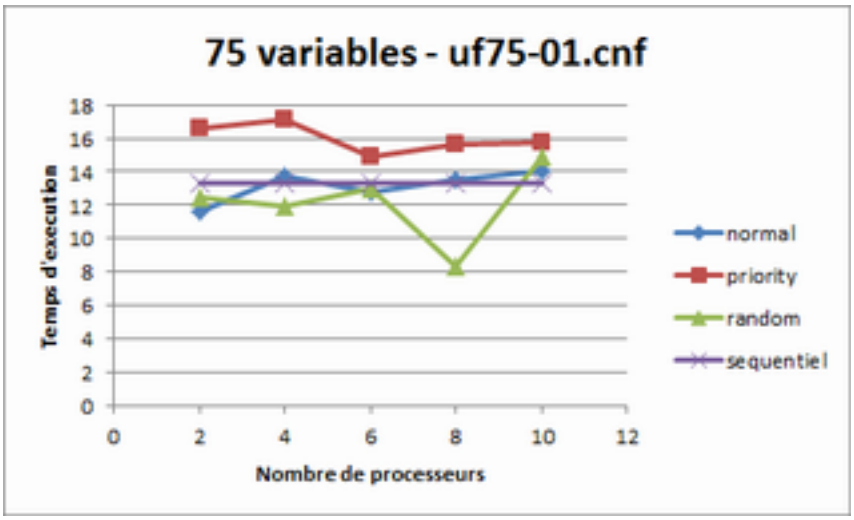
c. Variation du nombre de processus

Les résultats de variation de ce paramètre nous ont surpris. Nous nous attendions à avoir un optimal pour le nombre de coeurs de la machine utilisée, mais il n'en n'est rien et les graphiques ne montrent pas de tendance vraiment significatives.

Certains tests ayant été lancés alors que d'autres taches étaient menées sur la machine (musique, edition du rapport, navigateur web, …). Cela a probablement biaisé certains résultats.

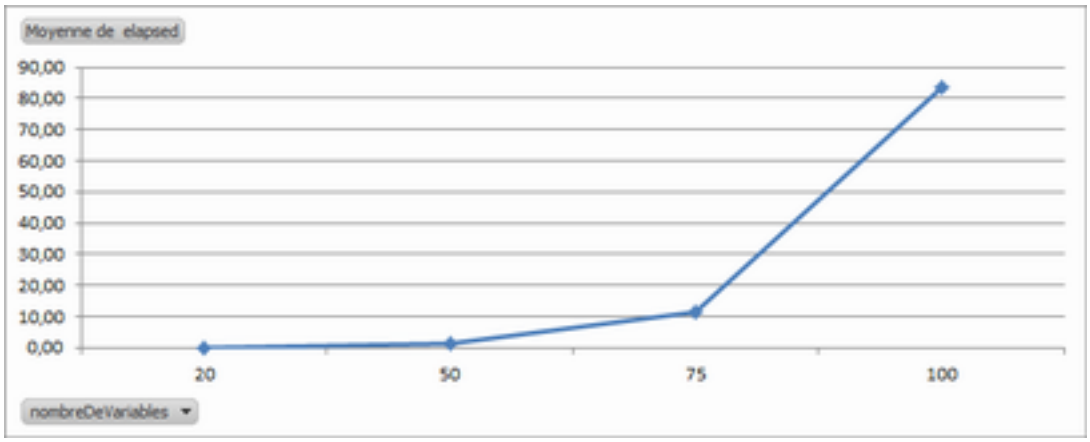






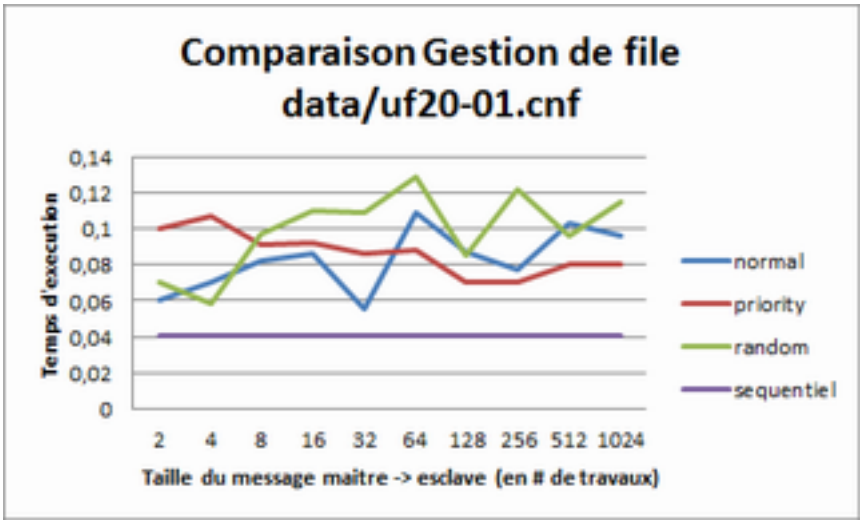
d. Variation de la taille du problème

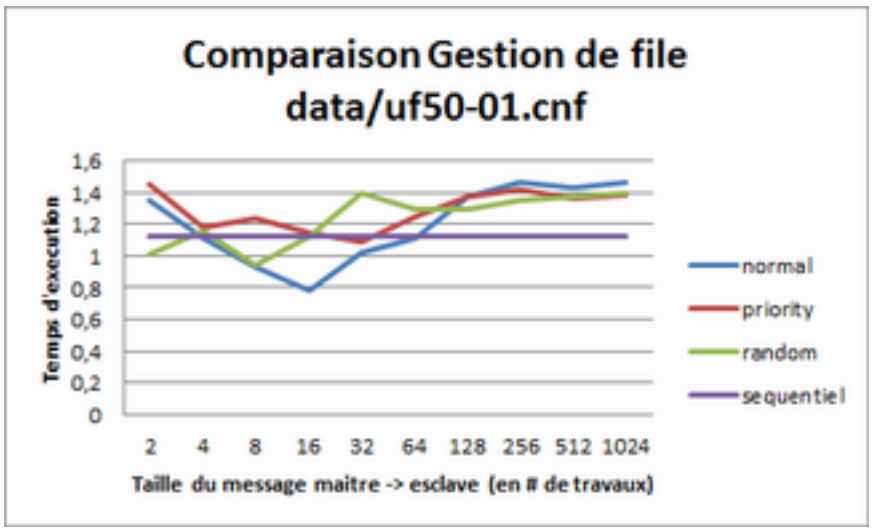
Comme prévu le temps de résolution augmente de manière exponentielle en fonction de la taille du problème :

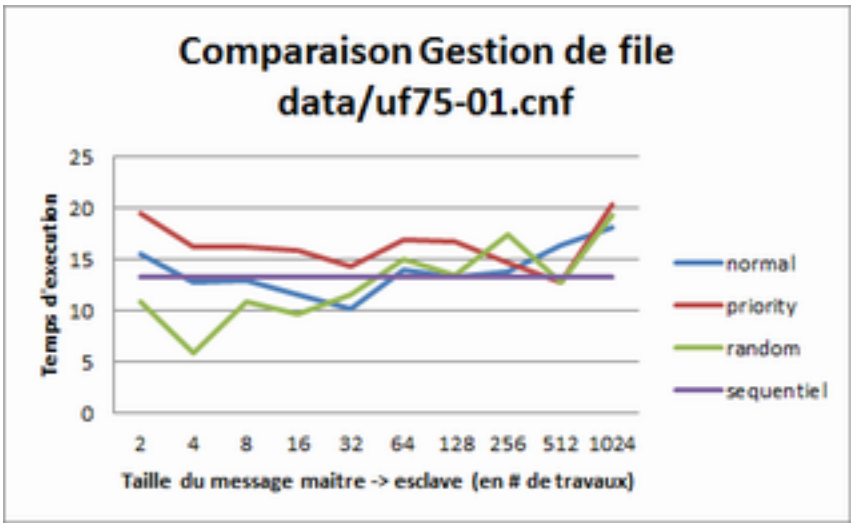


e. Variation du mode de gestion de la pile du maître

Une fois que l'on atteint un nombre de variables suffisamment grand, le mode de gestion de la pile joue également un rôle important, en général on observe que la file FIFO est moins performante que la file priorisée qui est elle-même semblable au séquentiel. Le choix aléatoire a quant à lui tendance à être plus performant.







4. Améliorations possibles

- Mise en place d'un cluster de machines pour exploiter réellement des processeurs différents
- Développement d'une nouvelle gestion de pile cx : aléatoire pondérée
- Mise en place d'une "abaque" pour trouver, à partir de la taille du problème et du nombre de processeur, la taille de batch optimale

Annexes

Sujet du projet et ressources du cours
http://www.prism.uvsq.fr/~joco/i.php/Main/Enseignement
cours de mpi: http://www.idris.fr/data/cours/parallel/mpi/IDRIS_MPI_cours_couleurs_presentation.pdf
Ressources pour Python:
http://stackoverflow.com/questions/1422260/which-python-mpi-library-to-use
tuto officiel mpi4py:
http://mpi4py.scipy.org/docs/usrman/tutorial.html
“utilisation de mpi avec python”:
http://calcul.math.cnrs.fr/Documents/Ecoles/2010/coursMPI.pdf
examples avec mpi4py:
https://github.com/jbornschein/mpi4py-examples
tutoriel echange de job slave/master:
http://www.lam-mpi.org/tutorials/one-step/ezstart.php

Ressources sur SAT:

Enoncé du projet:
http://www.prism.uvsq.fr/~joco/enseignement/td-programme.pdf
Description du problème:
http://en.wikipedia.org/wiki/Boolean_satisfiability_problem
Méthodes de résolution existantes
http://www.lita.univ-metz.fr/~singer/SatReview.pdf
Format CNF pour représenter les expressions boolénnes:
http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html
Algorithme de résolution de référence:
http://en.wikipedia.org/wiki/DPLL_algorithm
Question sur stackoverflow:
http://stackoverflow.com/questions/14254595/load-a-file-for-conjunctive-normal-form-into-a-list-of-lists
“a fast parallel sat solver” (article de recherche):
http://link.springer.com/content/pdf/10.1007%2FBF02127976
“An Exact Parallel Algorithm for the Maximum Satisability Problem”: http://euler.nmt.edu/~brian/students/jfurman.pdf
Microsoft // Programming
http://lion.disi.unitn.it/intelligent-optimization/LION3/slides/parallelSAT.pdf

Repository Git:
https://github.com/edelans/APDSAT
(pour installer git: https://help.github.com/articles/set-up-git)