Rapport: Génération de labyrinthe via MPI

Jean-Romain Luttringer

May 10, 2018

1 Implémentation

1.1 Présentation globale

Le code a été légèrement modifié afin de rendre la génération parallèle du labyrinthe correcte. De manière générale, la parallélisation se fait selon une répartition du travail "maître-esclave", à l'aide des fonctions d'échange de messages proposées par MPI.
Dans la suite de ce rapport, le nombre de processeurs utilisés sera appelé *size*.
Lors de la génération d'un labyrinthe de taille N*M, chaque processeur récupère une section du labyrinthe de taille N/size * M. Chaque processeur génère ensuite la partie de labyrinthe qui lui a été envoyée, et renvoie la section générée au processus maître une fois terminé. Le labyrinthe est divisé horizontalement.
Le processus "maître" (ici, le processus de rang 0) effectue également une partie de la génération du labyrinthe.
Afin de traiter des cas particuliers dans lesquels la hauteur du labyrinthe ne serait pas un multiple du nombre de processeurs, la fonction **scatterv** a été utilisée. Cette fonction est similaire à la fonction **scatter** vue en cours, mais permet de spécifier des tailles différentes pour chaque chunk. Ainsi, chaque processeur récupère un chunk de taille N/size * M, à l'exception du dernier, qui récupère un chunk de taille (N/size + N%size) * M (ce qu'il reste). Une fois les différentes sections du labyrinthe générées, elles sont récupérées par le processeur 0 via la fonction **gatherv**. Seul le processus maître affiche le labyrinthe terminé, et enregistre le labyrinthe dans un fichier.
Un des problèmes résultant de cette implémentation est la gestion des "frontières" entre chaque chunk. En effet, lors de la génération, le programme a besoin d'accéder aux valeurs contenues dans les voisins de la cellule traitée. Ces voisins peuvent cependant se trouver dans une section du labyrinthe générée par un autre processus. Ce problème se traduit par des lignes démarquant nettement la frontière entre deux sections du labyrinthe:
Une des solutions permettant de résoudre ce problème serait de faire en sorte que les processeurs

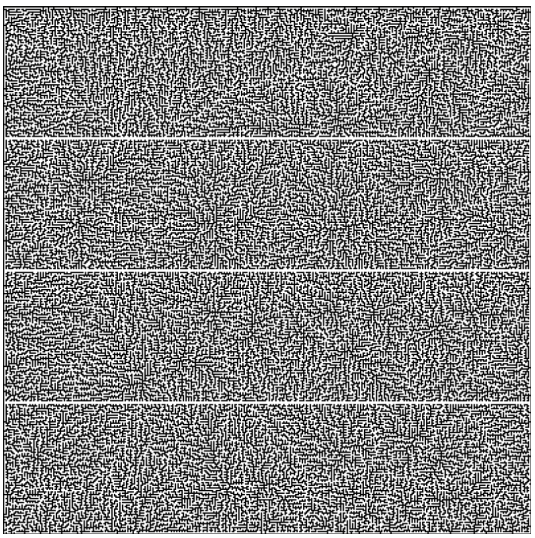


Figure 1: labyrinthe généré via une parallélisation naïve

communiquent entre eux afin de "partager" la ligne marquant la frontière entre leur section. Cependant, afin d'assurer que chaque processus possède une vision suffisamment récente de cette frontière, certaines sécurités devraient être rajoutées, handicapant le potentiel speed up du programme, et alourdissant le code.
La solution implémentée ici consiste à générer ces lignes frontières avant la génération parallèle. Ainsi, les lignes situées aux position [i*N/size][0] du tableau sont initialisées via une pose aléatoire de murs le long de cette ligne.
La ligne générée est envoyée au deux processus concerné par cette frontière. Ainsi, l'état de cette frontière n'est pas modifié au cours de l'exécution, et chaque processus possède une copie locale de cette ligne. L'implémentation est commentée dans le fichier source et sera détaillée plus tard dans ce rapport.
Voici le labyrinthe généré via le programme parallèle:
La première moitié de la figure montre les sections générées par chaque processus. On voit effec-

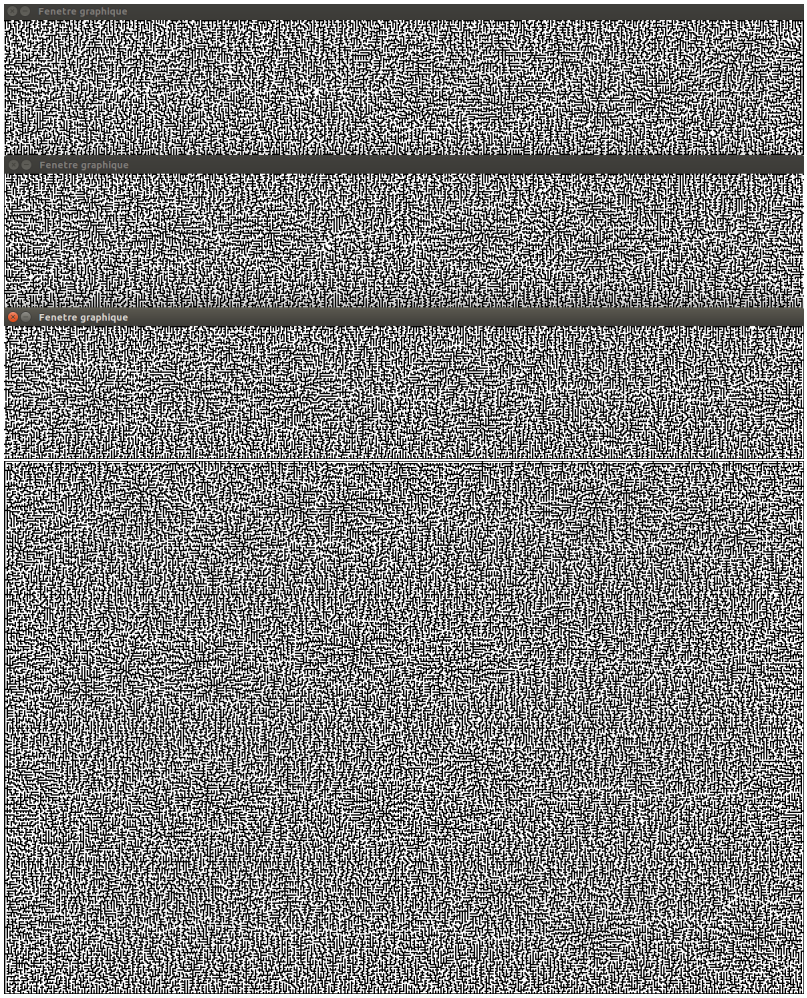


Figure 2: labyrinthe généré via parallélisation

tivement que chaque processus n'a généré qu'une section de taille N/size * M.
La deuxième moitié de la figure montre les différentes sections fusionnées, affichées par le processus 0. Enfin, la validité du labyrinthe peut être vérifiée en lançant le programme *chemin_lab*:

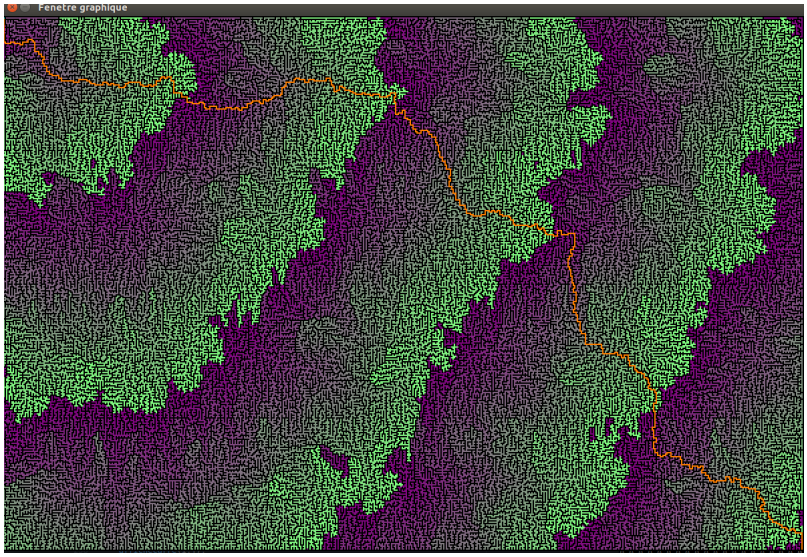


Figure 3: résolution du labyrinthe

On peut également vérifier que les labyrinthes dont la hauteur n'est pas un multiple du nombre de processus lançés sont gérés correctement. Par exemple, en lançant un labyrinthe de taille 43*40 sur 4 processeurs.
La figure de gauche montre la section générée par le processeur numéro 4. Ce processeur gérant

Figure 4: section du labyrinthe 43*40

la dernière section du labyrinthe, le mur marquant la fin du tableau est visible en dernière ligne. La figure de droite montre une section du milieu du labyrinthe. On remarque que les sections ne sont pas de taille égale, et que le dernier processeur a bel et bien récupéré le surplus du tableau. Il est important de noter que la section montrée sur la figure de droite récupère une section plus grande que N/size * M car elle récupère, comme expliqué précédemment, la ligne frontière générée préalablement.

1.2 Présentation détaillée

Le programme commence par une initilisation de MPI. Ensuite, un seul processeur génère les murs autour du labyrinthe, et ce même processeur initialise les lignes frontières.
La répartition des chunk se fait de la manière décrite précédemment: chaque processus récupère une section de taille (N/size+1) * M, afin de récupérer la partie du labyrinthe à générer et la ligne frontière n'appartenant à l'origine pas à la section que le processus est chargé de générer. Le dernier processus récupère en plus l'éventuel surplus si la hauteur du labyrinthe n'est pas un

multiple du nombres de processus.
Les données sont ensuite réparties via **scatterv**. Chaque processus affiche la génération de sa section du labyrinthe. Afin de faciliter la lecture et la programmation, chaque processus met à jour les bornes qu'il doit utilisée dans les différentes boucles. Ainsi, le code est identique et les bornes utilisées restent N et M, modifiées afin de s'adapter à la taille de la section gérée par le processus. Il est important de noter que le nombre de graines renseigné indique à présent le nombre de graines par section, et non le nombre de graines total. Le premier processus retire certaines cases constructibles sur la bordure supérieure et le dernier processus retire certaines cases constructibles sur la bordure inférieure. Tous les processus retirent des cases constructibles sur les bordures gauche et droite.
Outre les noms des variables traitées, la boucle de génération principale n'a pas été modifiée. Les différentes sections sont ensuite réunies via la fonction **gatherv**. Les fonctions **scatterv** et **gatherv** permettent de renseigner le nombre d'éléments de chaque chunks, et l'offset par rapport à l'origine du tableau. Ainsi, en jouant sur ces paramètres, il est facilement possible d'envoyer au processus la ligne frontière supplémentaire, et de réunir les données sans que cette "duplication" ne soit visible.
Enfin, le processus maître affiche les données et enregistre le fichier. Les tableaux sont ensuite free et la fonction MPI_Finalize est appelée.
Voici une comparaison d'un labyrinthe généré séquentiellement et d'un labyrinthe généré de manière parallèle via la méthode explicitée ci-dessus:
Le speed up moyen obtenu via cette implémentation est d'environ 9 sur les paramètres par défaut

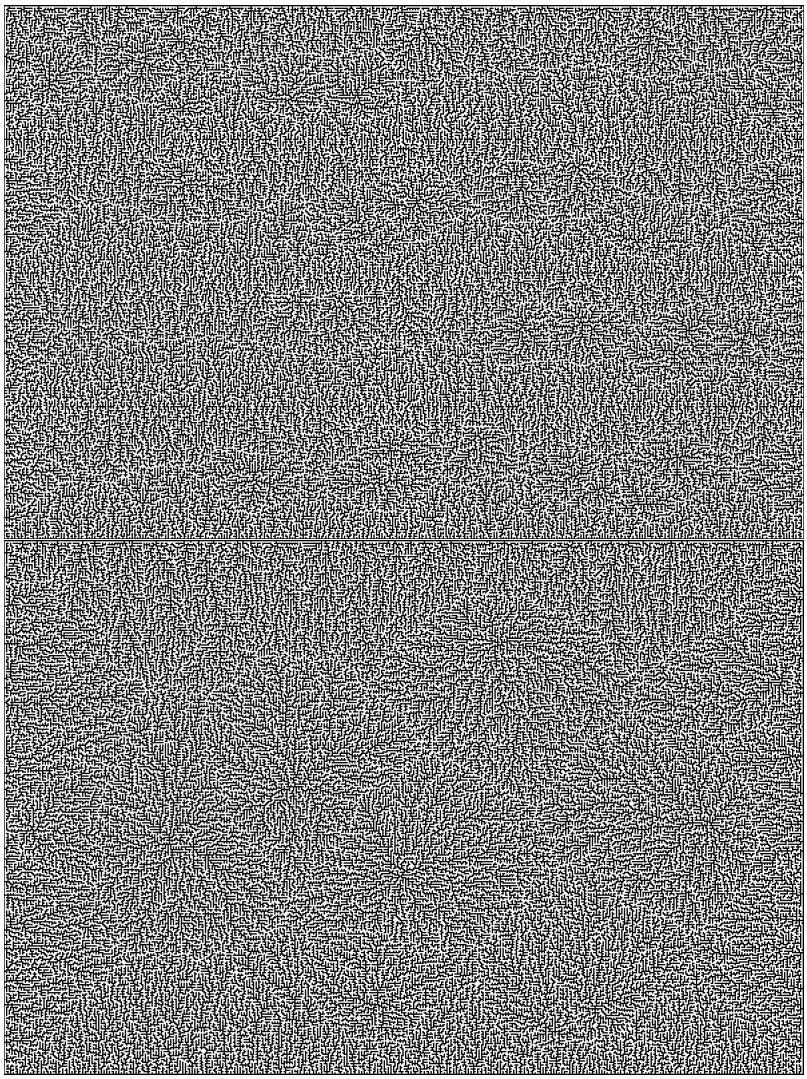


Figure 5: labyrinthes générés (version parallèle -au dessus- vs séquentielle -en dessous-

et sur 4 processeurs.

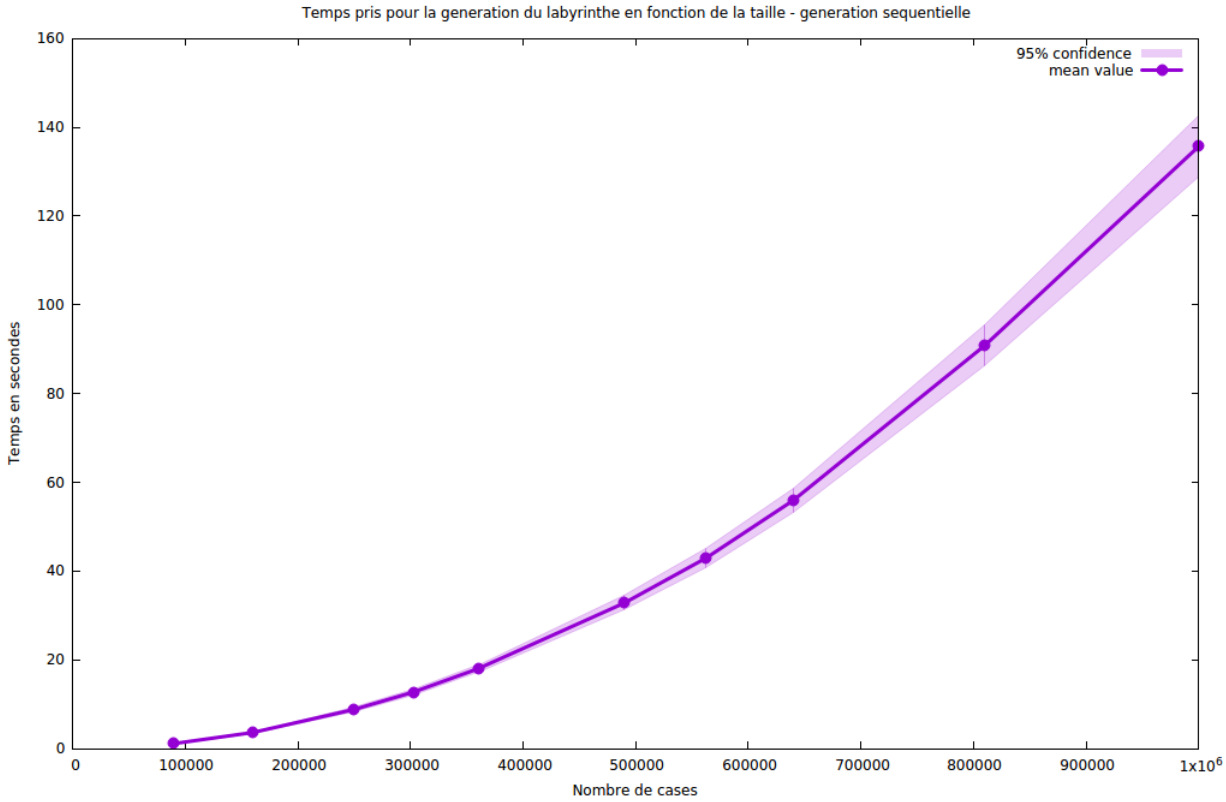
2 Evaluation de performances

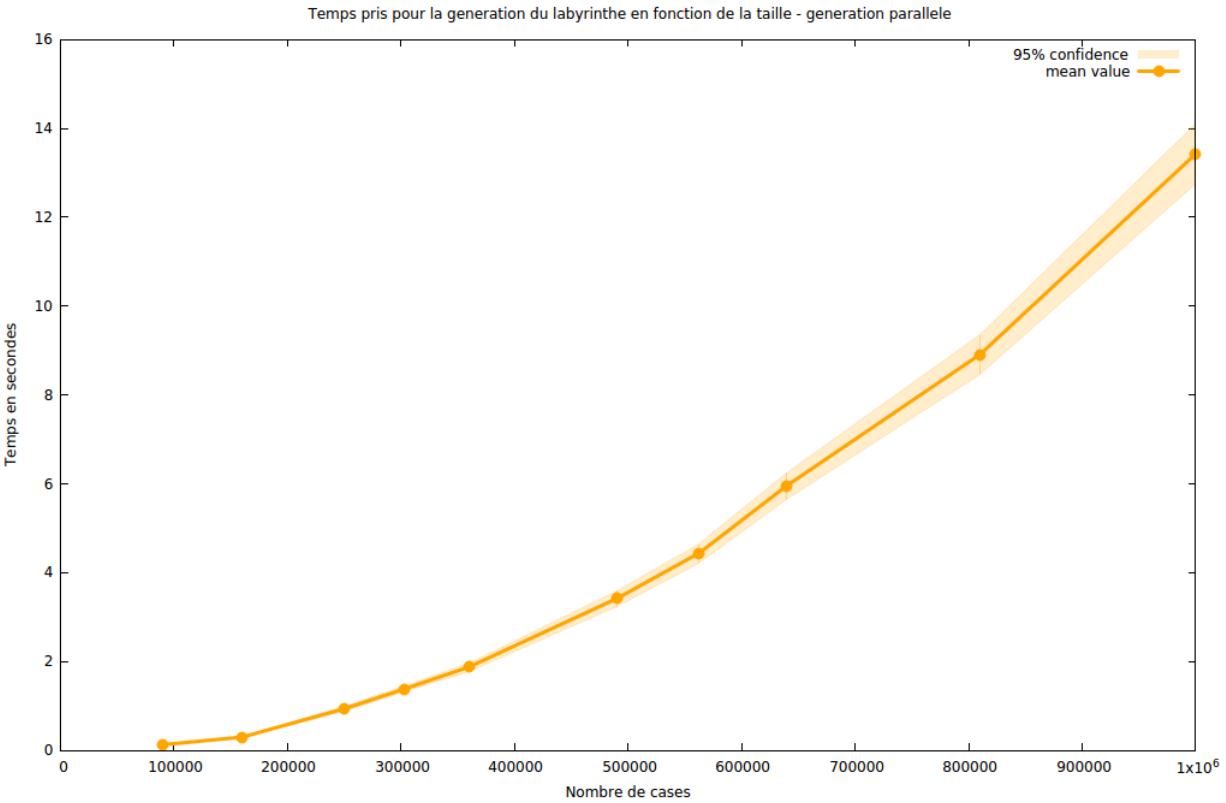
Les performances des programmes ont été évaluées sur dix tailles différentes, allant de 300^2 à 1000^2. Vingt exécutions ont été lancées pour chaque taille. La moyenne et l'écart type des expériences ont été calculée via un script awk. Le programme parallèle a été lancé en utilisant 4 processeurs. Afin que les paramètres soient identiques, le programme séquentiel a été lancé avec 8 ilots, et le programme parallèle avec 2 ilots par section, pour un total de 8 ilots également.
Les données brutes sont renseignées ci-dessous, et un graphique illustre l'évolution du temps pris pour la génération, accompagné d'un intervalle de confiance à 95%.

X	Parallèle		Séquentiel	
Taille	Temps moyen	Ecart-type	Temps moyen	Ecart-type
90000	0.13	0.021	1.17	0.02
160000	0.30	0.013	3.65	0.06
250000	0.94	0.128	8.84	0.19
302500	1.38	0.074	12.76	0.60
360000	1.88	0.037	18.04	1.02
490000	3.42	0.036	32.84	0.89
562500	4.44	0.764	42.97	0.40
640000	5.96	0.652	55.96	1.21
810000	8.91	0.040	90.83	2.53
1000000	13.42	0.52	135.69	4.43

On remarque que les performances du programme parallèle sont nettement supérieures aux performances du programme séquentiel. On observe un speed-up d'environ 10 quelque soit la taille du labyrinthe généré.
En effet, on observe par exemple, pour un labyrinthe de taille $900x900$, un temps de 8.91s pour la version parallèle, et un temps de 90.83s pour la version séquentielle.
L'évolution de la moyenne des temps mesurés est visible sur les graphiques ci-dessous.

On remarque que le temps pris par le programme suivant la taille du labyrinthe généré suit une évolution quasi identique pour les deux versions, à un facteur 10 près.
En effet, le temps pris passe de 0.13 à 13.42 secondes pour la génération parallèle, contre 1.17s à 135s pour la génération séquentielle.
Dû à la grande différence d'échelle entre les deux graphes, ces derniers n'ont pas pu être superposés de manière lisible. On voit cependant que l'écart de temps est tel que les intervalles de confiance ne se superposent pas. Ainsi, le programme parallèle semble constamment plus efficace que le programme séquentiel.
En conclusion, la méthode de parallélisation implémentée décrite dans la section précédent a permis d'obtenir un speed-up de 10 (précisémment $9,8$) lors de la génération du labyrinthe.


Temps pris pour la generation du labyrinthe en fonction de la taille - generation sequentielle
95% confidence
mean value
Temps en secondes
Nombre de cases
0
20
40
60
80
100
120
140
160
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6


Temps pris pour la generation du labyrinthe en fonction de la taille - generation parallele
95% confidence
mean value
Temps en secondes
Nombre de cases
0
2
4
6
8
10
12
14
16
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1x10^6