AUGER Romain | PETITBOULANGER--MELEUX Erwan | RIO Charles
INFO3



# Rapport de projet APD
# ~ Dessin de figures fractales ~



Franck BUTELLE Décembre 2018

# Rappel de l’objectif

Sujet n°6 : Écrire un programme MPI pour calculer deux figures fractales IFS (Iterated Function System), le flocon de Koch et le dragon de Lévy. Le programme utilisera la bibliothèque Cairo pour dessiner des lignes et permettra de spécifier la profondeur d'itération de la fonction de transformation. La taille du "papier" sera calculée en fonction de la profondeur d'itération.

On demandera d’abord à l’utilisateur un nombre d’itérations. La création des figures fractales utilisera la bibliothèque graphique Cairo et sera répartie entre les différents processeurs avec MPI. On s’efforcera de conserver la lisibilité de la figure quelle que soit l’itération en agrandissant le format de l’image en conséquence.

# Généralités

## Bibliothèque Cairo

La bibliothèque Cairo nous permet de dessiner sur des “surfaces”. Celles-ci sont de différents formats pour gérer couleurs et transparence de différentes façons.
Dans le cas du flocon, pour optimiser les performances, nous utilisons un format en niveaux de gris pour gérer chaque pixel sur un seul octet. Toutefois, pour afficher des résultats permettant de distinguer le travail des processus, le root (uniquement le root) travaille sur une surface couleur (4 octets par pixel avec la transparence).
Dans le cas du dragon, nous verrons que seul le root dessine (les autres processus ne font que du calcul), on se permet donc de travailler en couleurs.

Dans le cadre de ce projet, nous devons uniquement tracer des lignes et nous ne nous appuierons sur aucune forme standard. Cairo dispose pour nous de deux méthodes de dessin. La première est de travailler avec les coordonnées absolues, la seconde avec des coordonnées relatives. Or, le calcul de coordonnées pour nos deux figures est très compliqué, même en relatif. L’astuce que nous utilisons est de travailler avec les angles des figures. On peut en effet orienter notre curseur et utiliser une seule dimension des coordonnées relatives comme une distance.

```
cairo_rotate(curseur, angle_rad);
cairo_rel_line_to(curseur, longueur, 0); // x += longueur; y += 0;
```

En effet, le repère de coordonnées est relatif à l’orientation du curseur. De plus, les segments tout autour du flocon sont de longueur égale à une itération donnée et cette longueur est facile à calculer.

# Communications avec MPI

## Concernant le flocon

Le dessin de la figure est partagé entre les différents processus. Chacun a donc sa surface Cairo et la renvoie au root après le tracé de sa partie. Les surfaces de Cairo ne peuvent être envoyées par MPI telles quelles. On a donc mit en place deux fonctions pour convertir une surface Cairo A8 (niveaux de gris) en tableau, et pour utiliser ce tableau pour écrire sur la surface couleur du root (sous sa forme matricielle et non avec les opérations de dessin de Cairo). La fonction de réception du tableau écrit directement dans la surface root, on n'attend pas la réception des données de tous les processus pour écrire.

Afin d'optimiser les communications, la conversion des surfaces en tableaux n'est pas une simple transcription de la matrice de l'image. En effet, la partie dessinée de la figure représente une faible proportion de la surface. Plutôt que d'envoyer les valeurs de chaque pixel, on envoie donc uniquement les coordonnées des pixels blancs (la surface étant initialement noire). Les coordonnées sont codées sur des *int* et on en envoie deux par pixel dessiné (x et y). Mais il y en a si peu que ce compromis reste avantageux.

## Concernant le dragon

Dans le cas du dragon de Levy, ce n'est pas le dessin qui est réparti entre processus mais le calcul de la figure en elle-même. En effet, à l'inverse du flocon de Koch, nous n'avons pu établir un algorithme nous donnant de façon immédiate la suite d'instructions permettant le dessin de la figure. Les processus vont donc calculer cette suite et la renvoyer au root qui se chargera du dessin.

Le processus root va initialiser une série d'instructions sous forme de liste chaînée (nous détaillerons ci-après pourquoi) puis la transformer en tableau transmis aux différents processus. On utilisera la fonction *MPI_Scatterv()* pour pouvoir répartir ce tableau même si sa taille n'est pas divisible par le nombre de processus, et *MPI_Gatherv()* pour réaliser l'opération inverse.

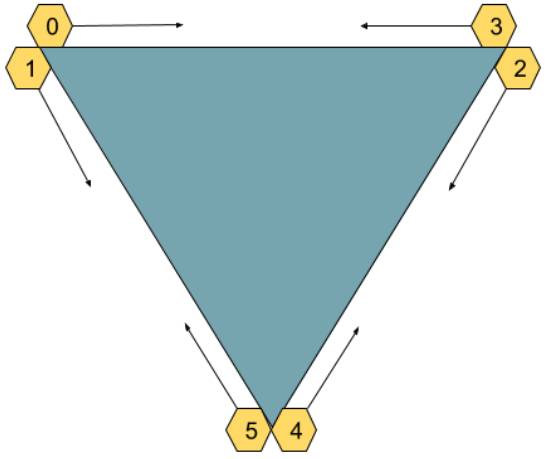
Les processus transforment le tableau reçu en liste chaînée à nouveau. Ensuite, à partir des instructions initiales reçues du root, ils vont construire la suite d'instructions des itérations suivantes. Enfin, quand les processus ont terminé, la totalité des instructions calculées est transmise au root qui se charge de dessiner l'image.

Les calculs de la suite d'instructions sont expliqués et détaillés plus loin dans la partie *Dragon de Levy*.

# Flocon de Koch

## Description du Flocon

Le flocon de Koch est une figure fractale qui se dessine à partir d'un triangle équilatéral. À chaque itération, de nouveaux triangles équilatéraux se dessinent sur le tiers central de chaque segment. Ces triangles ont donc pour longueur de côté ⅓ de la longueur des segments de l'itération précédente. La partie centrale des segments, sur laquelle les triangles apparaissent, est effacée. Seuls les contours de la figure subsistent et on obtient un flocon.




Les curseurs des processus sont disposés aux sommets du triangle initial et un processus sur deux dessine à l'envers.

L'ordre de placement des processus autour du triangle a été pensé en accord avec les communications de transmission vers le root pour éviter/limiter les blocages.
En clair, les processus ayant le moins de travail sont les premiers à communiquer pour que le root reçoive au plus tôt leurs données dans sa surface.

## Algorithmique et complexité

Nous avons pu mettre en place un algorithme qui permet à un processus de lire la suite d'angles de la figure à tracer. On utilise pour ça uniquement un tableau de taille du nombre d'itérations disponible en O(1) sans aucun calcul. On fera ensuite uniquement des incrémentations/décrémentations dans le tableau.

Le nombre de segments élémentaires et leur longueur sont calculés au préalable. Chaque processus dispose du tableau pour la lecture des angles et connaît le nombre de segments qu'il doit tracer. À chaque "pas", le processus oriente son curseur en lisant l'angle dans le tableau et trace un segment. Notre algorithme évite tout calcul, cependant, on ne peut s'affranchit de tracer $4^n$ segments, avec n la profondeur d'itération.

La complexité est donc en $O(4^n)$ en séquentiel et idéalement en $O(4^n/p)$ en parallèle.

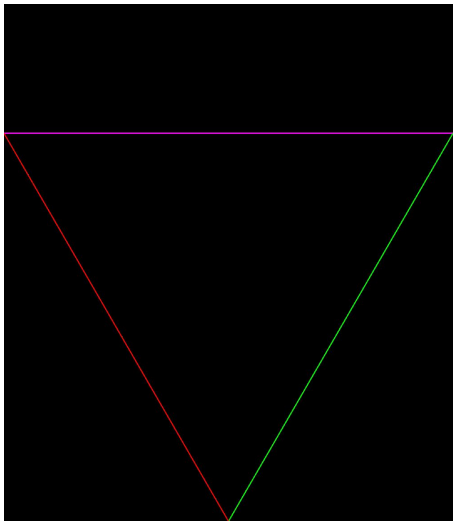
Cependant, nous n'utilisons que jusqu'à 6 processus et donc nous sommes au mieux en $O(4^n/6)$, donc toujours en $O(4^n)$.
De plus, le travail est équitablement réparti si $p \in \{1, 2, 3, 6\}$, mais lorsque $p \in \{4, 5\}$, malgré un gain de performance car les processus ayant le moins de travail communiquent leurs données à la surface root en avance, le proc 6 fait le même travail qu'avec p=3.

La conversion des surfaces en tableaux tel que décrit dans la partie *Communications avec MPI* est indépendante du nombre d'itérations puisque quel que soit le dessin obtenu, on parcourt la matrice de l'image en entier.

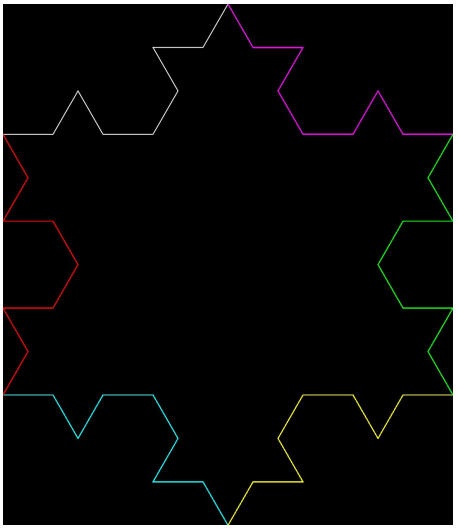
La taille du message envoyé grandit avec le nombre d'itérations puisque le nombre de pixels dessinés augmente (on rappelle qu'on envoie les coordonnées des pixels dessinés). En revanche, en nombre de message il n'y a aucune évolution. Chaque processus transmet toujours un message contenant la taille du tableau à transmettre puis le tableau en une fois. A cela s'ajoute un broadcast depuis le root pour transmettre le nombre d'itérations saisi par l'utilisateur. Le reste des données utiles à la réalisation du flocon est initialisé par tous les processus sans échanges de messages. On a donc 3*(nombre de processus - 1) messages échangés. (- le root)

## Résultats

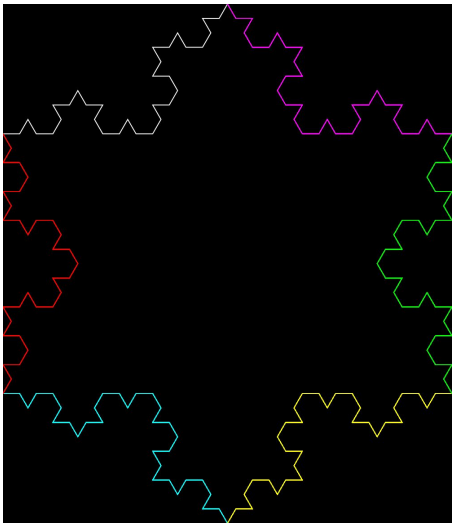
A l'origine, le root aussi travaillait en niveaux de gris, mais, pour le confort des présentations, nous sommes passé à une version en couleurs. La couleur est attribuée à chaque processus lors de la réception des données au niveau du root. De leur côté, ils continuent de fonctionner en niveaux de gris.



*0 itération*
*6 procs*



*2 itérations*
*6 procs*



*3 itérations*
*6 procs*

| Itérations | Temps séquentiel (s) | Temps distribué (p = 3) (s) |
|---|---|---|
| 5 | 0,017 | 0,2 |
| 6 | 0,052 | 0,205 |
| 7 | 0,147 | 0,22 |
| 8 | 0,45 | 0,29 |
| 9 | 1,69 | 0,58 |
| 10 | 7,22 | 2,22 |
| 11 | 25,6 | 13,5 |

# Dragon de Levy

## Description du Dragon

Le dragon de Levy est une figure fractale dont chaque itération peut être vue comme une rotation de 45° puis l'ajout du symétrique de la figure courante selon une symétrie axiale dont l'axe est le bord droit de la figure courante.

Le dragon de Levy est généralement réalisé selon une des deux méthodes suivantes :

- Par un système de fonctions itérées. Cette méthode calcule la position à l'itération n de chaque point en multipliant leurs coordonnées d'itération n-1 par deux matrices prédéfinies.

- Par un système de Lindenmayer (L-système), méthode que nous avons choisie.

Pour le dragon, on utilise le L-système suivant :

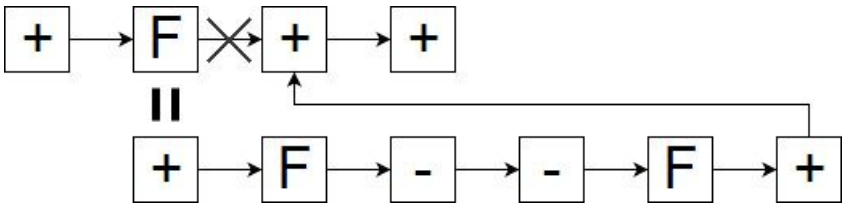
Alphabet : { F, +, - } | Axiome de départ : F | Règle : F → + F - - F +

- $n = 0 \rightarrow$ F
- $n = 1 \rightarrow$ +F--F+
- $n = 2 \rightarrow$ ++F--F+--+F--F++
- $n = 3 \rightarrow$ +++F--F+--+F--F++--++F--F+--+F--F+++

La suite finale de symboles est la liste d'instructions à exécuter pour le tracé de la figure. ('+' et '-' = droite et gauche, 'F' = avancer).

L'utilisation d'un tableau pour enregistrer les instructions n'aurait pas été une solution adéquate. En effet, on veut rajouter des éléments entre les cases du tableau à chaque itération. Celui-ci doit donc s'agrandir et ses éléments doivent être déplacés. Nous avons donc opté pour une liste chaînée, les opérations de conversion d'un format à l'autre étant plus efficaces que le travail constant sur l'allocation mémoire et le déplacement des éléments d'un tableau.

Avec la liste chaînée, on peut procéder comme suit : à la lecture du caractère 'F', on transforme ce maillon en un maillon '+', on ajoute cinq nouveaux maillons ('F', '-', '-', 'F', '+') que l'on raccorde à la liste selon le schéma suivant :




Nous avons créé une bibliothèque pour la manipulation des listes.

## Algorithmique et complexité

Nombre total de symboles : S = nombre de 'F' + nombre de '+' et '-'

A l'itération 0, il y a pour unique symbole un 'F' ($S_0 = 1$). De plus, chaque itération double le nombre de 'F'. On a donc $2^L$ 'F' à l'itération L.

A chaque itération, le nombre de '+' et de '-' (combiné) correspond au nombre qu'il y en avait à l'itération précédente, plus quatre fois le nombre de 'F' de l'itération précédente.

Au total, pour une itération L quelconque, on a le nombre de symboles S suivant :

$$S_L = 2^L + 4 \sum_{i=0}^{L-1} 2^i$$

$$S_L = 2^L + 4\,(2^L - 1)$$

$$S_L = 2^L + 2^{L+2} - 4$$

Pour dessiner une itération L, on parcourt L fois la liste pour avoir la liste finale puis une fois de plus pour lire les instructions et dessiner. La taille totale T de liste parcourue est donc :

$$T_L = \sum_{i=0}^{L} S_i$$

$$T_L = \sum_{i=0}^{L}(2^i + 2^{i+2} - 4)$$

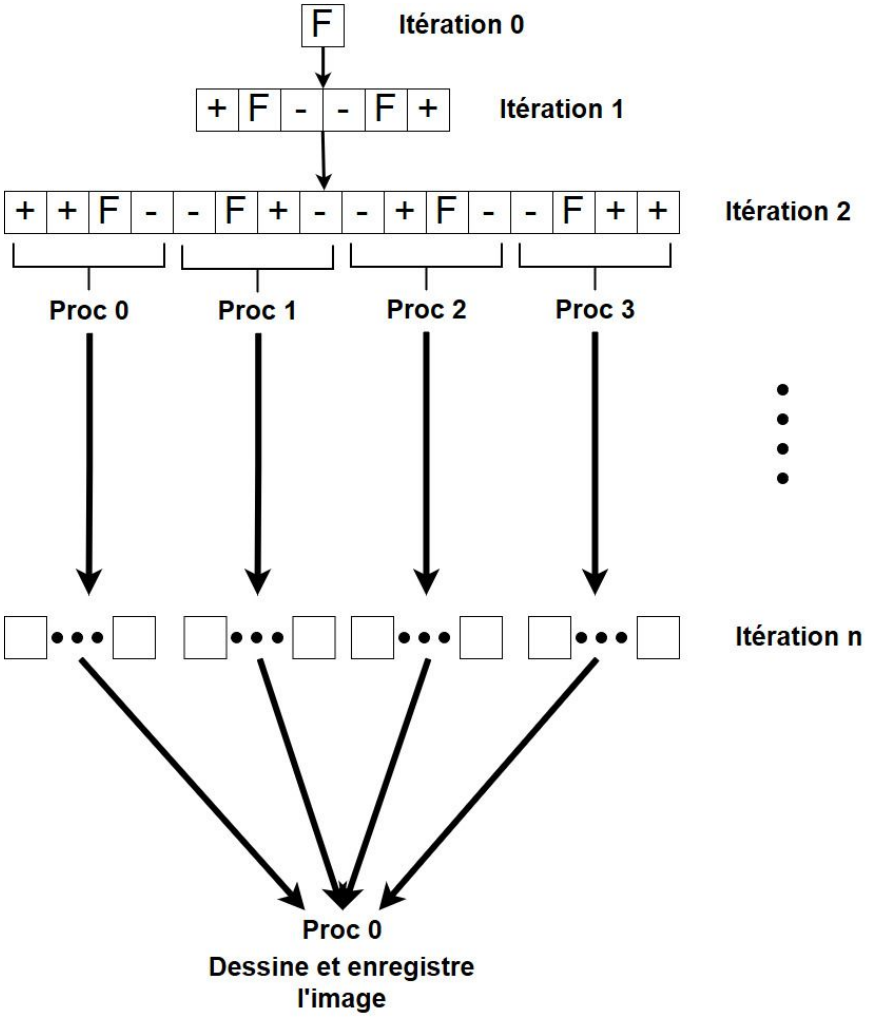
$$T_L = \sum_{i=0}^{L} 2^i + \sum_{i=0}^{L} 2^{i+2} - \sum_{i=0}^{L} 4$$

$$T_L = (2^{L+1} - 1) + (2^{L+3} - 4) - (4 * (L+1))$$

La complexité en temps du Dragon en séquentiel est donc en $O(2^{L+3})$ pour L itérations.

Concernant la distribution entre les différents processus. Nous réalisons plusieurs itérations sur le processus root jusqu'à ce qu'il soit possible de répartir notre liste de manière à ce que chaque processus ait au moins un caractère 'F'. Comme il y a $2^L$ caractères 'F' à l'itération L, le root procède à $\lceil log_2(p) \rceil$ itérations (p le nombre de procs) pour remplir cette condition.

Exemple ci-après avec 4 processus : le root va procéder à 2 itérations avant de distribuer la liste pour poursuivre les opérations.



Si le nombre d'itérations demandé est trop faible (c'est-à-dire inférieur à $\lceil log_2(p) \rceil$), le processus root réalisera alors toutes les itérations. Par exemple, demander le calcul d'une unique itération avec huit processus n'aurait aucun intérêt, le root réalisera donc ce calcul seul.

Voyons ce que cela nous donne en terme de complexité (toujours en étudiant le parcours des listes) :

La formule précédente donnait le nombre de maillons parcourus en comptant la lecture finale pour dessiner. Ici, le root va soit réaliser des itérations jusqu'à pouvoir envoyer à tous les processus, soit tout faire et dessiner. On a donc :

Soit L le nombre d'itérations et N = min($\lceil log_2(p) \rceil$, L).

Le root parcourt $(2^{N+1} - 1) + (2^{N+3} - 4) - (4 * (N + 1))$ maillons, soit au total si N=L, soit avant d'envoyer aux autres processus leur part de travail. Dans ce dernier cas, chaque processus réalise encore $\sum_{i=N}^{L} (Si / p)$ opérations.

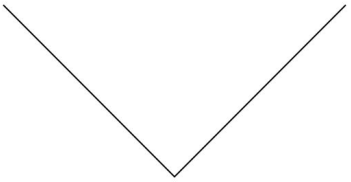
Enfin, côté messages :

Chaque processus, sauf le root, reçoit deux messages contenant d'une part la liste sur laquelle il doit travailler et d'autre part le nombre d'itérations qu'il doit calculer. Lorsque les calculs sont terminés, chaque processus transmet sa liste d'instructions ainsi que la taille de cette dernière au root.
Il y a donc exactement : 4*(p - 1) messages échangés.
(- le root)

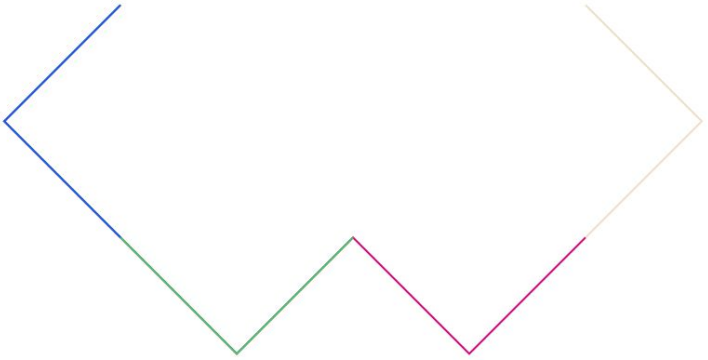
Si le root ne se charge pas seul des calculs et qu'il y a effectivement des messages, il y en aura toujours 4*(p - 1), indépendamment du nombre d'itérations. En revanche, la profondeur d'itération va déterminer la taille des messages contenants les listes envoyées au root. En effet, ces derniers contiennent les listes d'instructions dont la taille augmente de manière exponentielle.

# Résultats

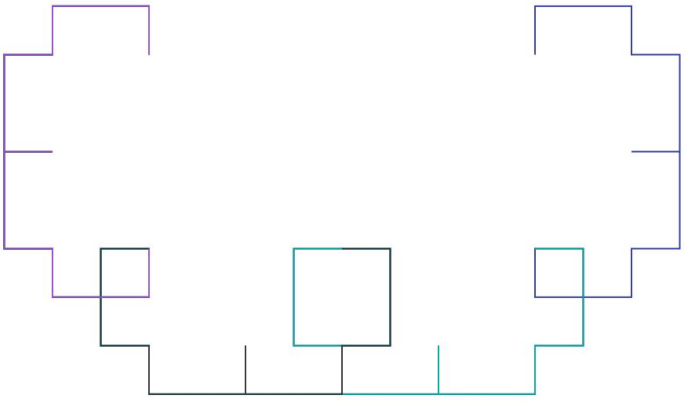
Les figures suivantes sont réalisées avec 4 processus.
Les parties de la fractale calculées par différents processus apparaissent de couleurs différentes.



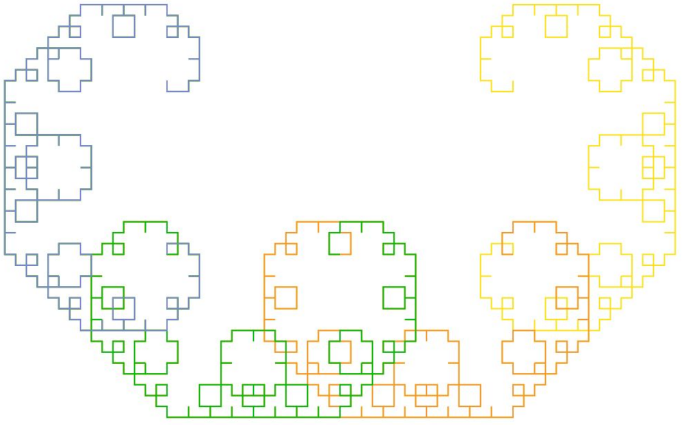
itération 1



itération 3



itération 6



itération 10.

Sur le résultat de l'itération 1, on peut remarquer qu'il n'y a qu'une seule couleur. En effet, comme indiqué précédemment, si le nombre d'itérations demandé est trop faible par rapport aux nombre de processus, alors le root réalise la totalité de l'image.

Sur tous les autres résultats présentés, le calcul est bien distribué et on voit le travail effectué par chaque processus.

| Itérations | Temps séquentiel (s) | Temps distribué (p = 4) (s) |
|---|---|---|
| 18 | 0,37 | 0,45 |
| 19 | 1,31 | 2,26 |
| 20 | 1,43 | 1,97 |
| 21 | 4,95 | 9,04 |
| 22 | 6,34 | 9,34 |
| 23 | 27 | 51 |
| 24 | 34,2 | 55 |
| 25 | 95 | 192 |

# Pour aller plus loin

## Pistes d'amélioration

### Pour le Flocon

Premièrement, dans l'optique d'optimiser les communications via MPI, on pourrait penser à transmettre seulement les coordonnées d'un pixel sur deux au root. Cela serait suffisant pour tracer la figure et diviserait par deux la taille des tableaux transmis vers le root.

On pourrait aussi penser à transmettre uniquement les coordonnées des sommets de la figure. Cairo dispose en effet d'une fonction pour récupérer la position d'un curseur. On peut donc, lors du tracé, récupérer les coordonnées au moment où l'on s'oriente.
Les tableaux transmis vers le root sont alors d'une taille bien plus faible : on transmet les coordonnées d'un pixel par segment, au lieu de l'ensemble des pixels des segments. De plus, les processus non-root n'ont plus à vraiment tracer la figure, mais juste à se déplacer entre chaque point. Cela ajoute un léger gain de performance côté processus non-root, car il n'y a pas de modification de l'état de la surface et d'enregistrement du parcours du curseur.
Cependant, ce gain de performance sur les autres processus et en termes de message MPI se perd au niveau du root, qui doit alors tracer toute la figure. En effet, notre version séquentielle actuelle a accès de façon immédiate à la liste des angles et trace la figure grâce à eux. Ici, le travail distribué nous donne la liste des coordonnées à relier, et tout le travail reste à faire par le root. On est donc perdant par rapport à notre programme séquentiel (qui utilise un excellent algorithme).

Enfin, le L-système est applicable au flocon de Koch (règle F → F + F - - F + F).
On a souhaité utiliser 2 systèmes différents pour nos 2 figures, mais l'algorithme du dragon est facilement adaptable au cas du flocon. Cela nous permettrait d'utiliser autant de processus que l'on souhaite. En revanche, la création de listes induit plus de calculs, d'autant que la complexité du flocon est en $4^n$ contre $2^n$ pour le dragon.
Voyons alors les pistes d'améliorations pour le dragon, qui utilise un L-système.

### Pour le Dragon

On peut noter 3 pistes d'améliorations du côté du dragon de Levy. Premièrement, on pourrait simplifier la liste de symboles à chacun de ses parcours. Par exemple, lorsqu'on rencontre un '+' immédiatement suivi d'un '-' et inversement, on supprime les 2 maillons. Cela s'étend évidemment lorsqu'on a une suite de '+' et '-' (sans 'F'), on peut supprimer autant de '+' que de '-' dans cette liste.

Deuxièmement, plutôt que de faire travailler le root jusqu'à ce que la liste de symboles contiennent des 'F' pour tous les processus (au risque de surcharger le root), il faudrait pouvoir distribuer le travail au fur et à mesure. (À voir l'impact sur le nombre de messages, et en temps avec les conversions listes - tableaux)

Enfin, lors de la division du travail entre les processus, le reste de la division est actuellement attribué au root, qui peut donc être surchargé. Il faudrait donc redistribuer ce reste entre les processus.

## Difficultés rencontrées

La première des difficultés a été de mettre au point une méthode pour dessiner les deux figures. En effet, la récursivité est la méthode préconisée dans les deux cas, mais ne correspond pas à nos besoins. Trouver un algorithme efficace pour le flocon de Koch d'une part et la solution du L-système pour le dragon de Levy d'autre part a donc constitué une part importante de ce projet, avant même de penser à distribuer nos calculs.

Deuxièmement, une fois la théorie en place, il a fallu passer en revue la librairie Cairo pour nous fournir les outils dont nous avions besoin. Si cela ne représente pas vraiment une difficulté, il s'agit au moins de temps consommé pour prendre en main la bibliothèque, d'autant que la documentation est parfois assez floue.

Du côté du flocon, nous nous heurtons à une grosse limite : il faudrait pouvoir utiliser un nombre quelconque de processus avec une répartition efficace du travail mais il nous manque une expression de coordonnées équiréparties tout autour pour y placer des processus. Ce problème est d'autant plus important que la complexité du flocon est importante et on profite mal du calcul distribué pour y faire face.

Le dernier problème, le problème majeur de ce projet d'ailleurs, concerne la taille de l'image. En effet, l'image doit s'agrandir pour pouvoir continuer à visualiser les itérations, sans compter que Cairo a besoin d'une image suffisamment grande pour dessiner correctement la figure. Au final, on a besoin d'une image énorme, dont la génération est coûteuse et qui, en plus, n'est plus manipulable sur nos machines.
Nous avions pensé au format SVG pour régler ce problème. Mais avec cette solution la représentation matricielle de l'image disparaît et ce n'est donc pas une solution viable.

# Conclusion

Le bilan de ce projet est que nous avons réussi à calculer et dessiner les figures fractales du flocon de Koch et du dragon de Levy, d'abord en séquentiel, puis en distribué sur différents processus avec MPI. Nos résultats permettent d'ailleurs d'identifier le travail de chacun des processus en couleurs.

Cela nous a permis de découvrir la bibliothèque graphique Cairo, qui recèle d'ailleurs beaucoup plus de possibilités que ce que nous exploitons ici. De plus, nous avons pu utiliser les communications MPI dans un cas plus concret que les exemples "bacs à sable" des TPs. Cela nous a beaucoup servi pour mieux maîtriser les fonctionnalités de MPI.

Nous avons mis en place deux méthodes distinctes, toutefois, le L-système est applicable au dessin du flocon et de manière générale à d'autres figures fractales et divers problèmes. L'algorithme implémenté pour le dragon est donc adaptable pour le calcul de toute suite issue d'un L-système. Nous avons d'ailleurs créé une bibliothèque de manipulation de listes chaînée. Autant d'éléments réutilisables.

# Bibliographie

https://fr.wikipedia.org/wiki/Flocon_de_Koch
https://fr.wikipedia.org/wiki/Courbe_de_L%C3%A9vy
(Et leurs équivalents en anglais) pour avoir un aperçu global des deux figures fractales.

https://www.geeksforgeeks.org/koch-curve-koch-snowflake/
Présentation détaillée du flocon de Koch et d'un algorithme récursif le réalisant.

http://ecademy.agnesscott.edu/~lriddle/ifs/levy/levy.htm
Présentation détaillé des deux méthodes les plus courantes pour réaliser le dragon de Lévy.

https://www.cairographics.org/documentation/
Documentation de la librairie Cairo et exemples de code.

https://stackoverflow.com/questions/43127823/cairo-draw-an-array-of-pixels
Accéder directement aux valeurs des pixels d'une surface de Cairo.

http://lists.cairographics.org/archives/cairo/attachments/20120914/6ac47ff8/attachment-0001.c
Exemple simple utilisant la librairie cairo_svg pour générer une image au format svg.

https://www.open-mpi.org
Documentation de openMPI.

https://cvw.cac.cornell.edu/MPIcc/exercise
Exemples d'utilisation de MPI_Scatterv (répartition inégalitaire entre les processus).

https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html
Exemples d'utilisation de MPI_Gatherv (pour réaliser l'opération inverse).