UFR sciences & techniques

UNIVERSITÉ DE BOURGOGNE

FractalSharp

LACHAUD Samuel / PAZOLA Loïs – M1 Informatique

SOMMAIRE

I) Introduction

En quoi consiste FractalSharp

Le projet de système distribué FractalSharp consiste en l'implémentation de la **suite de Mandelbrot** en utilisant un système distribué. Il peut être exécuté de plusieurs manières

- Sur 1 seul processus
- Sur une multitude de processus

Une fois l'image de la fractale de Mandelbrot calculée, elle est alors affichée à l'écran et l'utilisateur peut dessiner un rectangle (gardant le même ratio que l'image de base) pour recalculer de la même manière la suite de Mandelbrot, générer l'image zoomée et l'afficher.

Choix des technologies

La partie système distribué de FractalSharp est réalisée à l'aide **d'MPI (Message Passing Interface)**. De plus, FractalSharp est réalisé avec deux langages de programmation différents :

- **FractalSharp** et **FractSharpMPI** sont deux programmes réalisés en **C#** pour Windows, pour faire fonctionner MPI, nous avons utilisé MPI.NET de Microsoft. FractalSharp est conçue et adaptée pour Windows, cependant nous avons besoin d'exécuter notre programme sur des ordinateurs Linux afin d'utiliser le cluster qui nous est mis à disposition. Hors MPI.NET est très complexe à compiler sur linux et dotnet est absent sur les ordinateurs mis à disposition. Il existe donc :
- **FractalPlusPlus** et **FractalPlusPlusMPI** sont deux programmes réalisés en C++ et sont principalement prévus pour Linux, pour fonctionner sur le cluster mis à disposition.

Découpage de la problématique

La problématique peut facilement être découpée à partir de deux points cruciaux :

- L'image générée par le calcul de la suite de Mandelbrot doit être réalisé par **un programme à part** (Les version …MPI des programmes) car on ne peut pas relancer de calcul MPI à partir d'un programme qui viens de finir son calcul MPI (Les processus MPI finissent le programme main et ne peuvent donc pas être recréés).
- L'affichage de l'image et la partie qui demande le calcul de la suite de Mandelbrot au programme MPI doivent tout deux être dans **un Thread différent** car l'attente des clics souris pour dessiner le rectangle du zoom sur l'image serait bloquant pour le calcul MPI.

Structure de données + Algorithme

Dans cette partie nous allons expliquer comment nous avons utilisé MPI, et pourquoi nous avons choisis MPI :

Nous avons choisi d'utiliser MPI car c'est la première technologie qui nous est venue en tête, elle existe sur une multitude de langages de programmations (parmi eux **C#** langage que nous voulions utiliser pour le projet pour sa **facilité d'utilisation sur Windows**). De plus la **syntaxe d'MPI étant très basique**, cela semblait le choix le plus judicieux pour réaliser le projet dans le temps imparti.

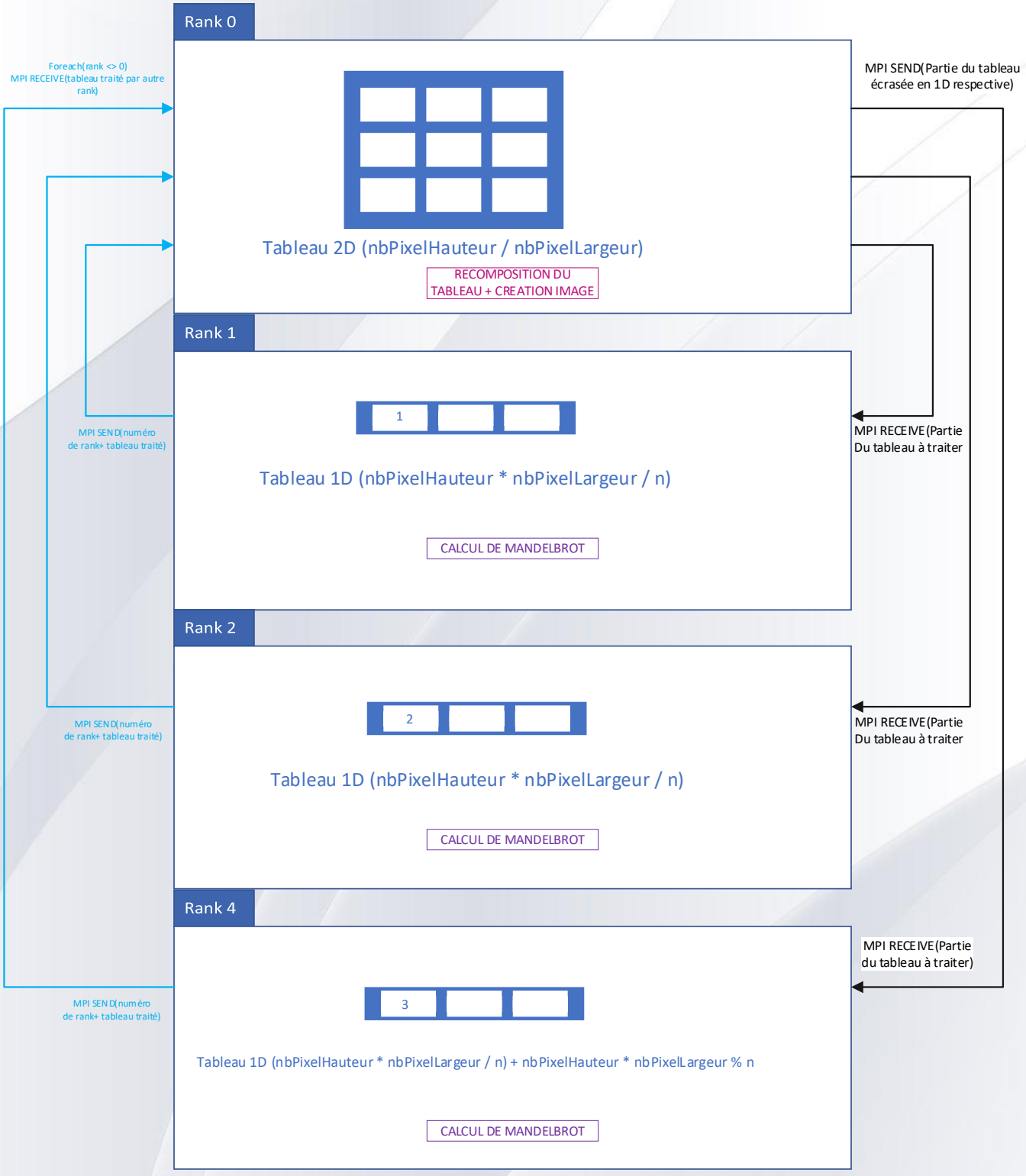
Les structures de données utilisées sont les suivantes :

CALCUL DE L'IMAGE DE MANDELBROT :

- Dans le **rang 0**, un **Tableau 2D** de la taille de taille verticale et horizontale équivalents à la taille de l'image calculée
- Chaque processus MPI crée un **tableau 1D de la taille 1D du tableau 2D divisée par le nombre de processus MPI** (le dernier processus MPI a une taille de tableau un peu plus grande si cette division comporte un reste). Ce tableau 1D comporte une valeur supplémentaire tout au départ, celle-ci est le rang du processus MPI.
- Le rang 0 récupère chaque tableau 1D, récupère la première valeur et sais donc quelles cases du tableau 2D remplir avec les valeurs du tableau 1D reçu

Voici un petit schéma expliquant le fonctionnement :

1. En noir : Envoi de chaque partie du tableau a chaque processus (ici n = 4 processus)
2. En violet : Les processus récupèrent le tablent font le calcul de la suite sur chaque case du tableau 1D, et en fonction de la vitesse de divergence de la suite, on inscrit dans la case une teinte de gris différente. On met dans la première case le numéro du processus qui a reçu le tableau (utile uniquement pour MPI.NET)
3. En bleu : Chaque processus renvoie le tableau remplis au processus 0.
4. En rose : A partir du numéro de processus de chaque trame, le processus 0 remplis sont tableau 2D avec les valeurs de chaque trame. Puis il fabrique l'image au format BITMAP et on l'enregistre dans le dossier temporaire du système d'exploitation.


Rank 0
Foreach(rank <> 0)
MPI RECEIVE(tableau traité par autre rank)
MPI SEND(Partie du tableau écrasée en 1D respective)
Tableau 2D (nbPixelHauteur / nbPixelLargeur)
RECOMPOSITION DU TABLEAU + CREATION IMAGE
Rank 1
1
MPI SEND(numéro de rank+ tableau traité)
MPI RECEIVE(Partie Du tableau à traiter
Tableau 1D (nbPixelHauteur * nbPixelLargeur / n)
CALCUL DE MANDELBROT
Rank 2
2
MPI SEND(numéro de rank+ tableau traité)
MPI RECEIVE(Partie Du tableau à traiter
Tableau 1D (nbPixelHauteur * nbPixelLargeur / n)
CALCUL DE MANDELBROT
Rank 4
MPI RECEIVE(Partie du tableau à traiter)
3
MPI SEND(numéro de rank+ tableau traité)
Tableau 1D (nbPixelHauteur * nbPixelLargeur / n) + nbPixelHauteur * nbPixelLargeur % n
CALCUL DE MANDELBROT

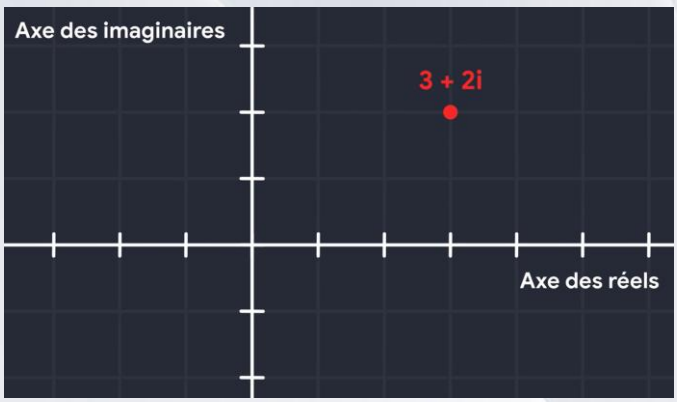
II) Structure du code

Les différentes classes

Comme dit précédemment, FractalSharp et FractalPlusPlus sont tous deux composés de deux programmes différents. Ces deux programmes sont écrits principalement avec une **approche fonctionnelle**. Cependant, tous deux, utilisent des classes C# ou C++, et une classe écrite pour l'occasion.

- FractalSharp (GUI) – C#
  - Utilisation principalement des classes **Form** et **PictureBox** qui respectivement servent à afficher un GUI et afficher une image de celui-ci.
- FractalSharpMPI – C#
  - Utilisation de la classe **SkiaSharp** pour gérer la création de **fichier BMP** sur Windows et Linux.
  - Classe **Complex** pour gérer les calculs nécessaires sur des nombres complexes pour la suite de Mandelbrot (**Modulo, addition et multiplication**)
  - Classe **PixelColor** pour regrouper les 3 composantes de la couleur dans une seule classe (R, G et B)
- FractalPlusPlus (GUI) – C++
  - Utilisation de la librairie **SDL** pour la création d'un GUI. Utilisation dans cette bibliothèque des classes **SDL_Surface** pour créer l'affichage graphique et l'affichage de l'image.
- FractalPlusPlusMPI – C++
  - Utilisation de la bibliothèque **SDL** pour la création du **fichier BMP** de l'image sur Windows et Linux.
  - Utilisation d'une **structure color** qui comporte les 3 composants d'une couleur (R, G et B)
  - Classe **Complex** pour gérer les calculs nécessaires sur des nombres complexes pour la suite de Mandelbrot (**Modulo, addition et multiplication**)

La classe qui a été créé pour l'occasion est la classe **Complex**, nous l'utilisons car pour les calculs de la suite de Mandelbrot, **les axes X et Y sont les composants réels et imaginaires** d'un nombre complexe. Voici un exemple :



Les méthodes importantes (MPI)

Dans cette partie, nous allons voir les principales méthodes du programme MPI de FractalSharp. Les codes C++ seront donnés ici, mais les versions C# sont tout de même disponibles.

```
/// <summary>
/// This method calculates the color of a pixel and returns it.
/// The pixel is black if the sequence converge.
/// The pixel is in other colors (gray scale) if the sequence diverge.
/// </summary>
/// <param name="iXpos">X position of the pixel</param>
/// <param name="iYpos">Y position of the pixel</param>
/// <param name="pixelWidth">number of pixels in width</param>
/// <param name="pixelHeight">number of pixels in height</param>
/// <param name="minRangeX">minimum range of the X axis</param>
/// <param name="maxRangeX">maximum range of the X axis</param>
/// <param name="minRangeY">minimum range of the Y axis</param>
/// <param name="maxRangeY">maximum range of the Y axis</param>
/// <returns>color of the pixel</returns>
color GetPixelColor(int iXpos, int iYpos, int pixelWidth, int pixelHeight, double minRangeX, double maxRangeX, double minRangeY, double maxRangeY)
{
	// Calculate if Mandelbrot sequence diverge

	double rangeXPos = (double)iXpos / (double)pixelWidth * (maxRangeX - minRangeX) + minRangeX;
	double rangeYPos = (double)iYpos / (double)pixelHeight * (maxRangeY - minRangeY) + minRangeY;

	Complex c = Complex(rangeXPos, rangeYPos);
	Complex z = Complex(0, 0);

	int iteration = 0;
	const int maxIteration = 1000;
	while (iteration < maxIteration && z.Modulus() <= 2) // AND Z mod 2 < 2
	{
		// Max iteration --> If not diverge
		// z mod 2 < 2 --> If diverge
		z = z.NextIteration(c);
		iteration++;
	}
	if (iteration == maxIteration)
	{
		return color{ 0, 0, 0 };
	}
	else
	{
		// Color smoothing Mandelbrot (a little bit)
		double log_zn = log(z.Modulus());
		double nu = log(log_zn / log(2)) / log(2);
		iteration = iteration + 1 - (int)nu;

		// Gray gradient with color smoothing
		int colorValue = (int)(255.0 * sqrt((double)iteration / (double)maxIteration));
		return color{ colorValue, colorValue, colorValue };
	}
}
```

La méthode **GetPixelColor** permet d'obtenir les composantes rouge, vert et bleu à afficher pour un pixel donné. En appliquant la suite de Mandelbrot, on obtient donc à chaque itération une valeur complexe (ici z est le complexe changeant à chaque itération).

Le nombre **maximal d'itération est fixé à 100**. Ainsi, si on ne détecte pas que la suite de Mandelbrot diverge sur ce pixel au bout de 100 itérations, on donne une couleur R, G et B au pixel actuellement traité, sinon on lui donne la couleur noire (s'il diverge)

Pour savoir **si le point diverge**, nous vérifions si le **modulo de z** (notre nombre complexe est **inférieur ou égal à 2**. Si ce n'est pas le cas, c'est que la suite diverge.

Pour l'application des couleurs, nous avons choisi d'appliquer des échelles de gris en fonction de la vitesse de divergence. Cependant, des **frontières très marquées** apparaissaient entre les **différentes vitesses de divergences**. Nous avons alors rendu plus **fluide le changement de couleur** pour avoir plus un aspect **dégradé** à l'aide de logarithmes.

```
/// <summary>
/// This method creates a Bitmap image with the pixels passed in parameter
/// </summary>
/// <param name="pixels">2D array of color (r,g,b) which contains the color of each pixel</param>
void CreateMandelbrotImage(color** pixels)
{
	// Create the surface
	SDL_Surface* surface;
	Uint32 rmask, gmask, bmask, amask;

#if SDL_BYTEORDER == SDL_BIG_ENDIAN
	rmask = 0xff000000;
	gmask = 0x00ff0000;
	bmask = 0x0000ff00;
	amask = 0x00000000;
#else
	rmask = 0x000000ff;
	gmask = 0x0000ff00;
	bmask = 0x00ff0000;
	amask = 0x00000000;
#endif
	surface = SDL_CreateRGBSurface(SDL_SWSURFACE, pixelWidth, pixelHeight, 32, rmask, gmask, bmask, amask);
	if (surface == NULL) {
		fprintf(stderr, "CreateRGBSurface failed: %s\n", SDL_GetError());
		exit(1);
	}

	// Fill the Bitmap with black, so we only need to set the pixels where Mandelbrot is diverging
	SDL_FillRect(surface, NULL, SDL_MapRGB(surface->format, 0, 0, 0));

	// Set diverging pixels
	for (int i = 0; i < pixelWidth; i++)
	{
		for (int j = 0; j < pixelHeight; j++)
		{
			if (IsDiverging(pixels[i][j]))
			{
				unsigned char* surfacePixels = (unsigned char*)surface->pixels;
				surfacePixels[4 * (j * pixelWidth + i) + 0] = pixels[i][j].b; // Blue
				surfacePixels[4 * (j * pixelWidth + i) + 1] = pixels[i][j].g; // Green
				surfacePixels[4 * (j * pixelWidth + i) + 2] = pixels[i][j].r; // Red
			}
		}
	}

	// Save Mandelbrot image as a file
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) && !defined(__CYGWIN__)
	std::string path = std::filesystem::temp_directory_path().string() + "Mandelbrot.bmp";
#else
	std::string path = "/tmp/Mandelbrot.bmp";
#endif
	SDL_SaveBMP(surface, path.c_str());
	std::cout << "Mandelbrot image saved in " << path << std::endl;
	std::cout << "----------------------------------------" << std::endl;
}
```

La méthode **CreateMandelbrotImage** permet de créer une image au format Bitmap (BMP) à partir d'un tableau 2D de pixels (couleurs R, G et B).

Pour faire cela, nous utilisons **deux boucles for imbriquées** pour parcourir le tableau 2D puis on applique la couleur sur le pixel du **SDL_Surface** associé.

Enfin le **fichier est sauvegardé dans le répertoire temporaire du système d'exploitation** :

- **/tmp** sur Linux.
- **%temp%** sur Windows.

Les méthodes importantes (GUI)

Dans cette partie, nous allons voir la principale méthode du programme GUI de FractalSharp. Le code C++ sera donné ici, mais la versions C# est tout de même disponibles.

```
/// <summary>
/// Call the MPI program to calculate Mandelbrot with all the parameters
/// </summary>
/// <param name="P1x">Optional parameter which is the x coordinate of the top left point after selecting an area to zoom
in</param>
/// <param name="P1y">Optional parameter which is the y coordinate of the top left point after selecting an area to zoom
in</param>
/// <param name="P2x">Optional parameter which is the x coordinate of the bottom right point after selecting an area to zoom
in</param>
/// <param name="P2y">Optional parameter which is the y coordinate of the bottom right point after selecting an area to zoom
in</param>
void CalculateMandelbrot(double P1x = 0, double P1y = 0, double P2x = 0, double P2y = 0) {
        // Calculate the previous absolute range of the image
        double rangeX = abs(P2XinAxe - P1XinAxe);
        double rangeY = abs(P2YinAxe - P1YinAxe);

        // Calculate the new range
        double localP1XinAxe = P1x / pixelWidth * rangeX - rangeX / 2;
        double localP1YinAxe = P1y / pixelHeight * rangeY - rangeY / 2;
        double localP2XinAxe = P2x / pixelWidth * rangeX - rangeX / 2;
        double localP2YinAxe = P2y / pixelHeight * rangeY - rangeY / 2;

        // Reorder the points in case the user selected the area from bottom to top and/or from right to left
        if (localP1XinAxe < localP2XinAxe)
        {
                P1XinAxe = localP1XinAxe;
                P2XinAxe = localP2XinAxe;
        }
        else
        {
                P1XinAxe = localP2XinAxe;
                P2XinAxe = localP1XinAxe;
        }
        if (localP1YinAxe < localP2YinAxe)
        {
                P1YinAxe = localP1YinAxe;
                P2YinAxe = localP2YinAxe;
        }
        else
        {
                P1YinAxe = localP2YinAxe;
                P2YinAxe = localP1YinAxe;
        }

        // Display the new range
        std::cout << "-----------------------------------------------" << std::endl;
        std::cout << "Range of the Mandelbrot set :" << std::endl;
        std::cout << "rangeX = " << P2XinAxe - P1XinAxe << ", rangeY = " << P2YinAxe - P1YinAxe << std::endl;
        std::cout << "-----------------------------------------------" << std::endl;

        // Execute the MPI program to generate the Mandelbrot image

#if defined(WIN32) || defined(_WIN32) || defined(__WIN32) && !defined(__CYGWIN__)
        constexpr char FPPExeName[] = "FractalPlusPlusMPI.exe";
#else
        constexpr char FPPExeName[] = "./FractalPlusPlusMPI"; // In linux we need to prepend "./" when it's in the current
directory
#endif
        constexpr char MPIExeName[] = "mpiexec";
        std::string commandeString;

        if (nbProcessMpi == 1)
        {
                // Store the command in the commandeString variable
                commandeString = std::string(FPPExeName) + ' ' + std::to_string(pixelWidth) + ' ' +
std::to_string(pixelHeight) + ' ' + std::to_string(P1XinAxe) + ' ' + std::to_string(P2XinAxe) + ' ' + std::to_string(P1YinAxe) +
' ' + std::to_string(P2YinAxe);
        }
        else
        {
                // Store the command in the commandeString variable
                commandeString = std::string(MPIExeName) + " -n " + std::to_string(nbProcessMpi) + ' ' +
std::string(FPPExeName) + ' ' + std::to_string(pixelWidth) + ' ' + std::to_string(pixelHeight) + ' ' + std::to_string(P1XinAxe)
+ ' ' + std::to_string(P2XinAxe) + ' ' + std::to_string(P1YinAxe) + ' ' + std::to_string(P2YinAxe);
        }

        std::cout.flush(); // Flush the terminal buffer before calling the system method to avoid mixing the output of the two
programs
        system(commandeString.c_str());

        SetMandelbrotImage();
}
```

La méthode **CalculateMandelbrot** permet de lancer le programme MPI avec les paramètres requis. A partir des points dessinés par l'utilisateur, on interpole les deux autres points du rectangle délimitant la zone de zoom.

L'enjeu de cette partie est de lancer le programme MPI sur Linux ou sur Windows **en fonction de l'architecture.**

III) Performance

Comparaison des performances

L'enjeu de notre programme, c'est de proposer une analyse des performances sur l'ajout de MPI dans le calcul d'une fractale basée sur la suite de Mandelbrot. Nous avons 3 conditions de tests différents :

- **C# (Windows)**
- **C++ (Windows)**
- **C++ (Linux)**

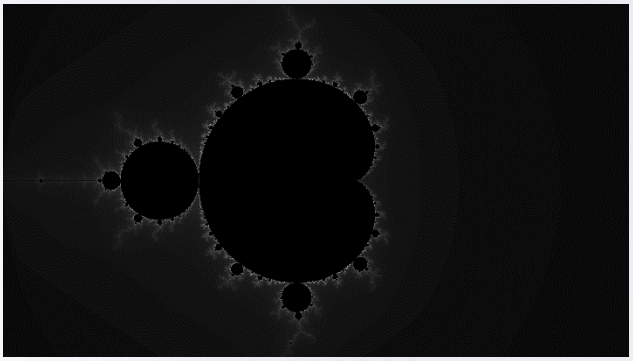
Sur ces 3 conditions, nous allons lancer 6 tests différents :

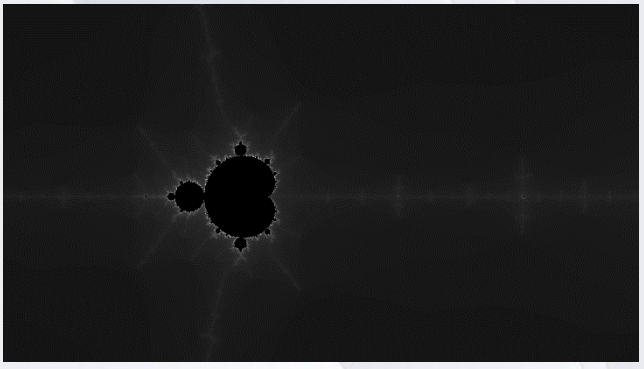
- Image en **640x630** non zoomée
- Image en **640x360** zoomée
- Image en **1280x720** non zoomée
- Image en **1280x720** zoomée
- Image en **4K (3840x2160)** non zoomée
- Image en **4K (3840x2160)** zoomée
- Image en très haute qualité **(16000x9000)** non zoomée

Sur ces 5 premiers tests, nous avons pris les nombres de processus suivants :

- C# - de 1 à 8
- C++ Windows – de 1 à 8
- C++ Linux – de 1 à 8 puis 12, 16, 32 et 64

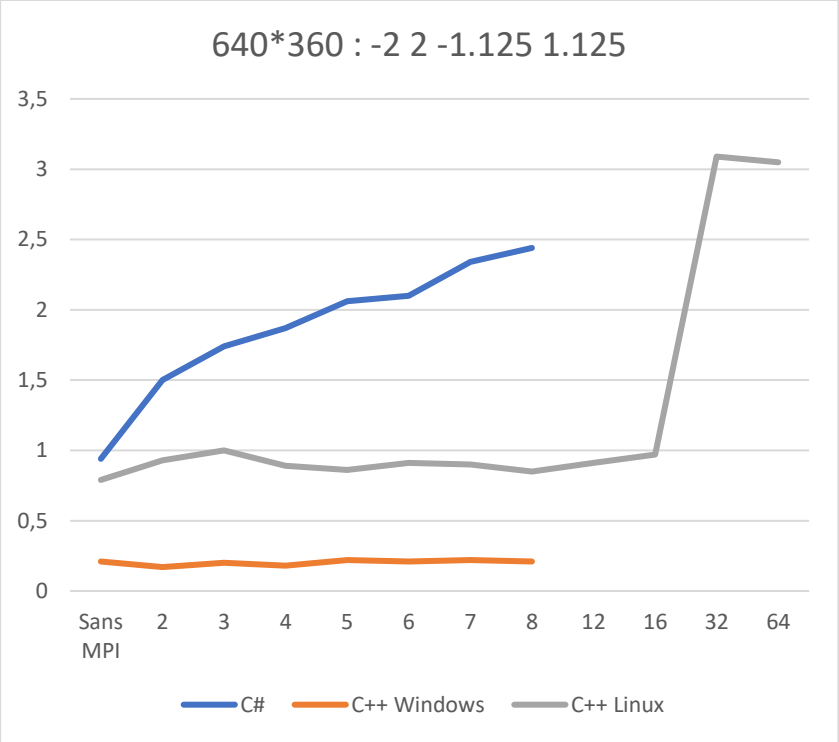
Sur la dernière image seuls les tests C++ Linux ont été effectués.





640*360 : -2 2 -1.125 1.125

Nombre de processus MPI	Windows C#	Windows C++	Linux C++
Sans MPI	0,94	0,21	0,79
2	1,5	0,17	0,93
3	1,74	0,2	1
4	1,87	0,18	0,89
5	2,06	0,22	0,86
6	2,1	0,21	0,91
7	2,34	0,22	0,9
8	2,44	0,21	0,85
12			0,91
16			0,97
32			3,09
64			3,05

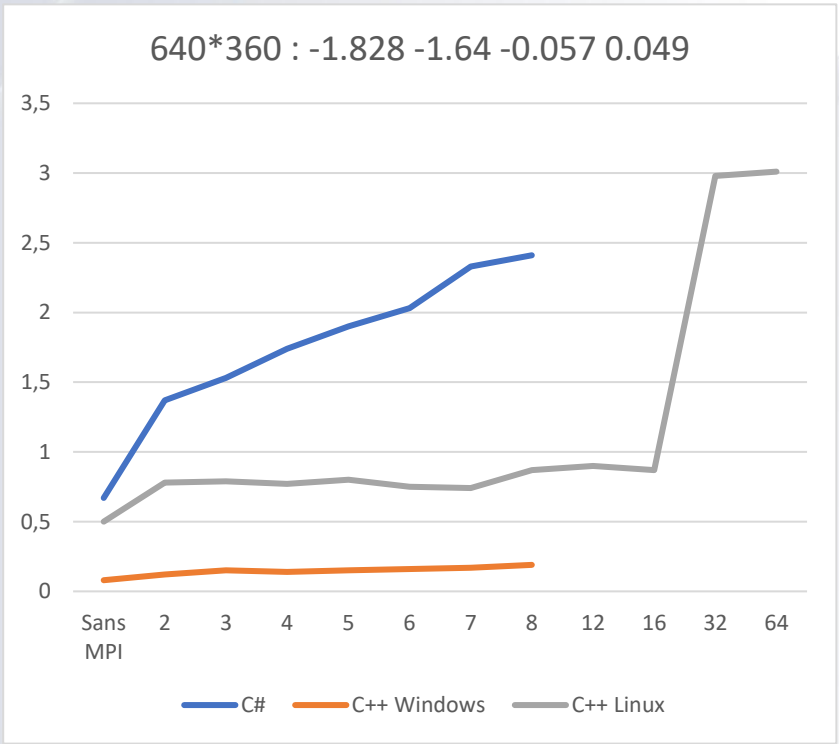


Sur ce premier test, nous avons une image en **640x630** non zoomée. Les performances sur Windows ne vont être testée que jusqu'à 8 processus car pas de possibilité dans notre configuration actuelle de tester sur plus.

Les performances de C# et C++ Linux sont de plus en plus mauvaise par rapport à la hausse du nombre de processus car l'image étant très petite, le temps de déplacement des données est supérieur au temps gagné en parallélisant les calculs. On obtient de très bonnes performances sur la version C++ Windows.

640*360 : -1.828 -1.64 -0.057 0.049

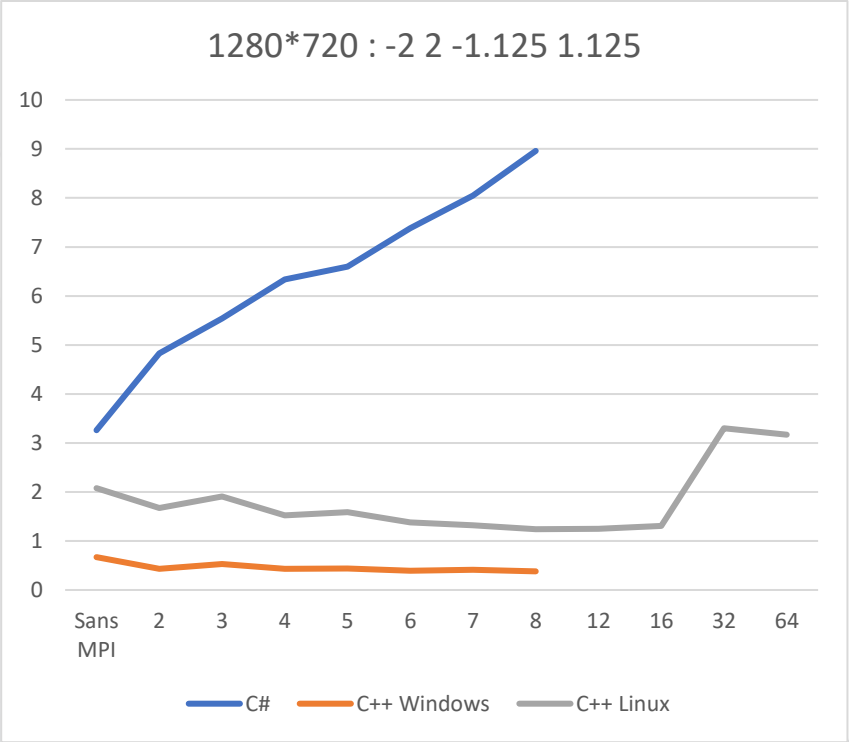
Nombre de processus MPI	Windows C#	Windows C++	Linux C++
Sans MPI	0,67	0,08	0,5
2	1,37	0,12	0,78
3	1,53	0,15	0,79
4	1,74	0,14	0,77
5	1,9	0,15	0,8
6	2,03	0,16	0,75
7	2,33	0,17	0,74
8	2,41	0,19	0,87
12			0,9
16			0,87
32			2,98
64			3,01



Sur ce second test, nous avons une image en **640x630** zoomée. Sur une petite image, les performances zoomées sont sensiblement les mêmes que sur une image dézoomée.

1280*720 : -2 2 -1.125 1.125

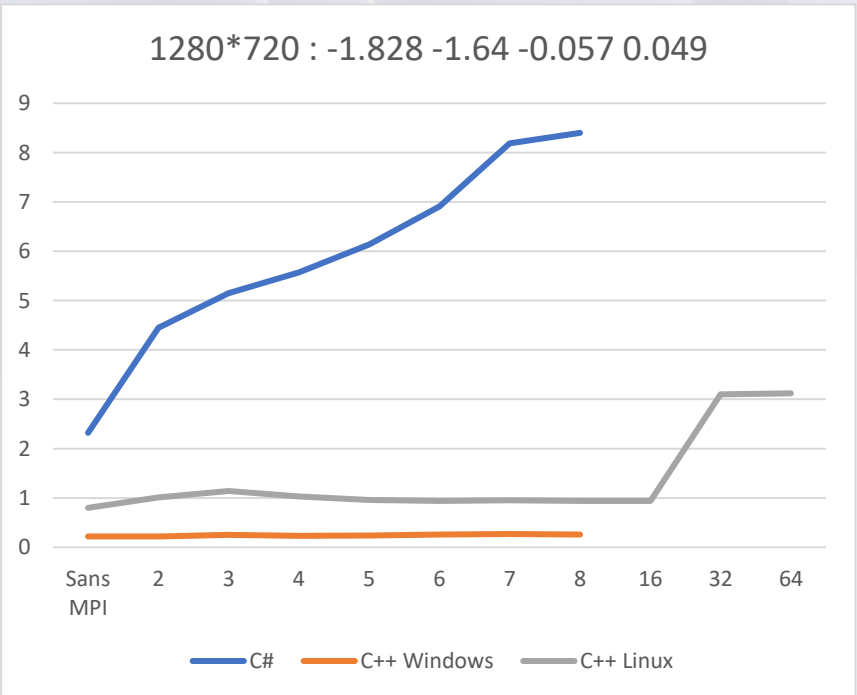
Nombre de processus MPI	Windows C#	Windows C++	Linux C++
Sans MPI	3,26	0,67	2,08
2	4,83	0,43	1,67
3	5,54	0,53	1,91
4	6,34	0,43	1,52
5	6,6	0,44	1,59
6	7,38	0,39	1,38
7	8,05	0,41	1,32
8	8,96	0,38	1,24
12			1,25
16			1,31
32			3,3
64			3,17



Sur ce troisième test, nous avons une image en **1280x720** non zoomée. Sur cette image de taille moyenne, nous commençons à avoir des différences significatives. La performance du programme C# sur Windows se dégrade très rapidement. Cependant, sur le même système d'exploitation, C++ obtient des performances légèrement meilleures en augmentant le nombre de processus MPI. Cependant le gain de performance ne justifie pas l'utilisation d'MPI et de 8 processus différents. L'autre grande différence se fait sur Linux avec C++. Jusqu'ici les performances étaient plutôt constantes jusqu'à 16 processus, mais se dégradaient énormément au-delà. Sur cette image les performances, comme sur Windows sont meilleures jusqu'à 16 processus, se dégradent au-delà, mais très nettement moins. Cela permet d'émettre l'hypothèse que les performances au-delà de 16 processus seraient très bonnes sur des grandes images.

1280*720 : -1.828 -1.64 -0.057 0.049

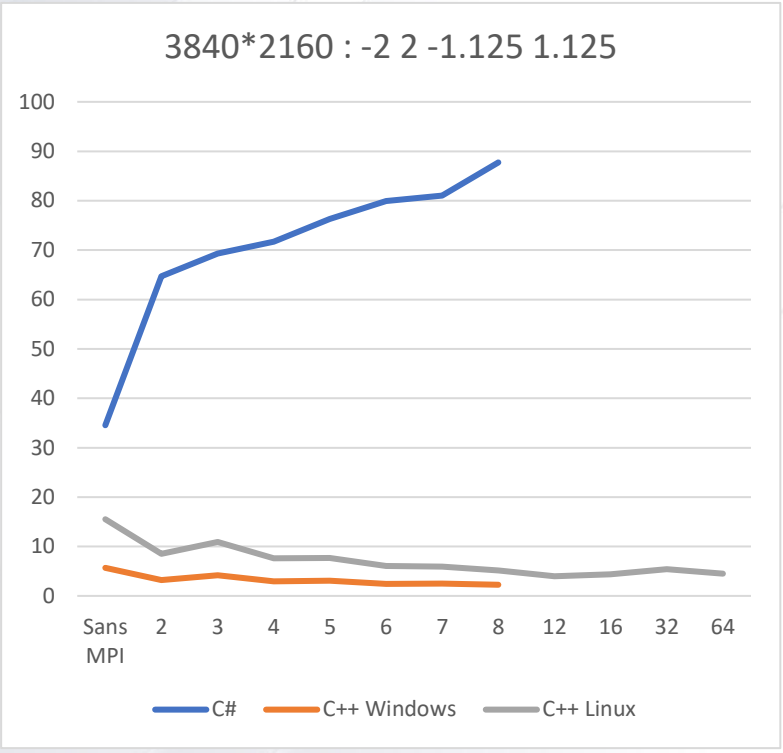
Nombre de processus MPI	Windows C#	Windows C++	Linux C++
Sans MPI	2,32	0,22	0,8
2	4,45	0,22	1,01
3	5,15	0,25	1,14
4	5,57	0,23	1,03
5	6,14	0,24	0,96
6	6,91	0,26	0,94
7	8,19	0,27	0,95
8	8,4	0,26	0,94
16			0,94
32			3,1
64			3,12



Sur ce quatrième test, nous avons une image en **1280x720** zoomée. Il n'y a pas de changement significatif entre l'image zoomée et l'image non zoomée

3840*2160 : -2 2 -1.125 1.125

Nombre de processus MPI	Windows C#	Windows C++	Linux C++
Sans MPI	34,54	5,68	15,52
2	64,7	3,2	8,52
3	69,3	4,17	10,92
4	71,72	2,93	7,61
5	76,31	3,11	7,7
6	79,92	2,46	6,03
7	81,04	2,47	5,94
8	87,74	2,26	5,12
12			3,98
16			4,35
32			5,39
64			4,53

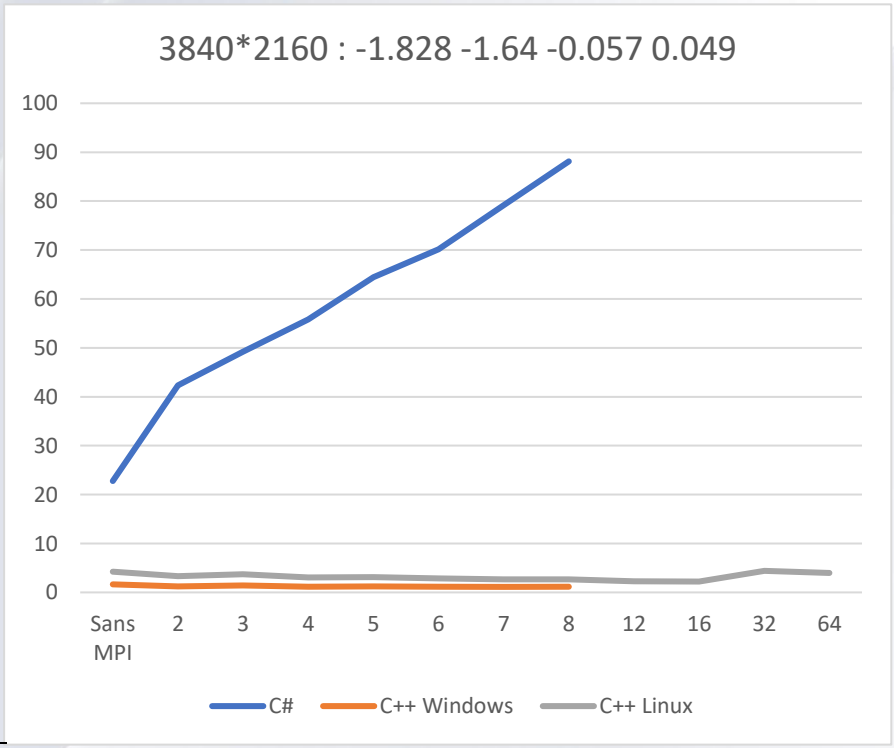


Sur ce cinquième test, nous avons une image en **4K** non zoomée. A partir d'ici, nous n'allons plus nous intéresser aux performances de C#, mais les performances C++ commencent à être de plus en plus intéressantes.

Les temps en secondes commencent à être de plus en plus grands, et les performances Windows et Linux de plus en plus proches. L'hypothèse précédente commence à se confirmer, Les performances sont maintenant mauvaises avec peu de processus MPI, et deviennent largement meilleur plus on rajoute de processus. Il n'y maintenant que très peu de différences au-delà de 16 processus.

3840*2160 : -1.828 -1.64 -0.057 0.049

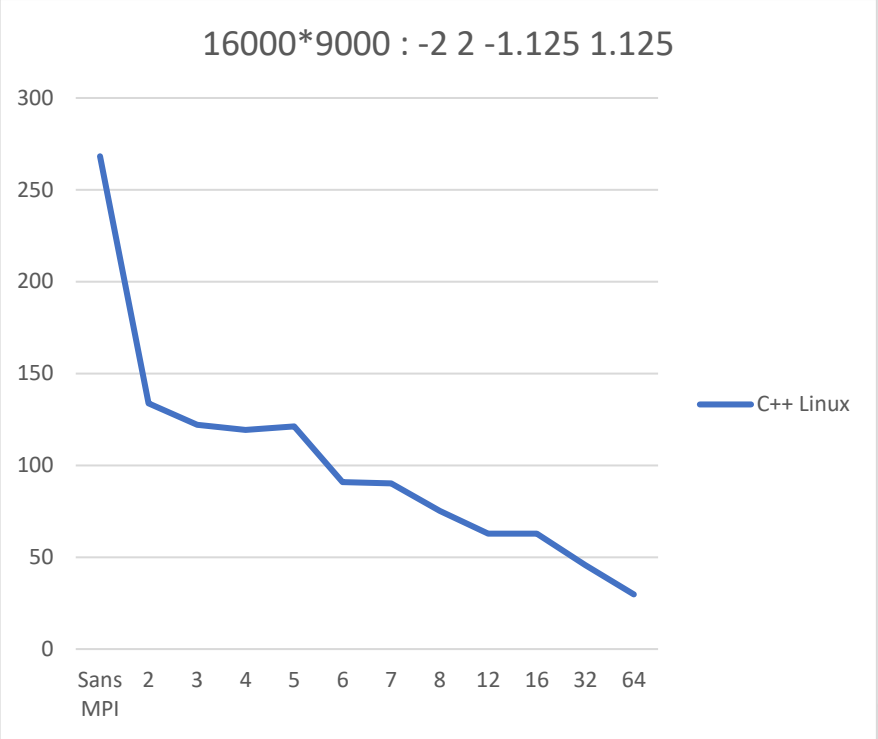
Nombre de processus MPI	Windows C#	Windows C++ Windows	Linux C++ Linux
Sans MPI	22,77	1,65	4,23
2	42,34	1,23	3,33
3	49,2	1,4	3,7
4	55,79	1,14	3,05
5	64,44	1,22	3,12
6	70,17	1,14	2,88
7	79,17	1,13	2,7
8	88,12	1,15	2,68
12			2,3
16			2,22
32			4,41
64			3,96



Sur ce sixième test, nous avons une image en **4K** zoomée. Il n'y a que peu de changement par rapport à l'image non zoomée, cependant la différence de performance entre C++ Windows et Linux s'efface encore un peu plus.

16000*9000 : -2 2 -1.125 1.125

	Linux
Nombre de processus MPI	C++ Linux
Sans MPI	268,27
2	133,82
3	122,10
4	119,22
5	121,15
6	90,92
7	90,25
8	75,29
12	62,84
16	62,8
32	45,81
64	29,75



Sur ce septième et dernier test, nous avons une image en **16000x9000** zoomée. Sur cette image, nous avons voulu démontrer notre hypothèse. Nous avons donc uniquement recueilli les données C++ Linux.

Sur cette très grande image, on observe une très nette différence entre l'usage de MPI et quand on ne l'utilise pas. On observe également de très bonnes performances quand on utilise 64 processus (plus on utilise de processus, plus les performances sont élevées).

Ce qui pourrait être amélioré

À la suite de tous ces tests de performances nous pouvons en tirer quelques conclusions sur la pertinence et la performance de MPI sur ce cas d'utilisation. Nous allons avant apporter quelques points d'informations influant sur la pertinence de ces résultats.

- Les tests Windows ont été effectués sur un seul ordinateur portable avec un processeur 8 cœur.
- Les tests Linux ont été effectué sur 4 ordinateurs de la salle MI104 (4 * 16 processus)
- Chaque test a été réalisé 3 fois, le résultat en seconde est une moyenne des 3 résultats, arrondit à 2 décimales.

Nous pouvons donc déduire de ces tests que l'utilisation de MPI est justifiée sur les très gros calculs. Cependant sur des images de faibles résolutions, le temps de transfert et trop élevé par rapport au temps gagné en parallélisant les calculs.

- Si on calcule une image en 4K, le choix d'utilisation de MPI est justifié
- En dessous de la 4K, l'utilisation simple du programme est tout aussi performante.

Il y a tout de même quelques améliorations qui pourraient améliorer significativement les performances de ces tests :

- Le programme C# sérialise une classe pixelColor, ce qui ralentit énormément le calcul, surtout en MPI.
- Les tests C++ Linux ont été réalisés en pleine période d'affluence des PC de la salle MI104 (17/12/2022), réduisant les performances des tests.

IV) Lancement du programme

Procédure de compilation

Pour lancer le programme, nous allons tout d'abord procéder à sa compilation. Vu que FractalSharp et FractalPlusPlus sont deux programmes différents, nous allons détailler les étapes de compilation de chacun :

1. Cloner le projet avec la commande : `git clone` https://github.com/samlach2222/FractalSharp.git
2. Se rendre dans le dossier FractalSharp pour le projet C# et FractalPlusPlus pour le projet C++.

FractalSharp (Windows)

1. Installez Visual Studio, puis lancez le projet avec le fichier `FractalSharp.sln`
2. Effectuer un clic droit sur la solution puis `Générer la solution`
3. Rendez-vous dans le dossier `.\FractalSharp\bin\[Release|Debug]\net6.0-windows\`

FractalPlusPlus (Windows)

1. Installez Visual Studio, puis lancez le projet avec le fichier `FractalPlusPlus.sln`
2. Effectuer un clic droit sur la solution puis `Générer la solution`
3. Rendez-vous dans le dossier `.\x64\[Release|Debug]\`

FractalPlusPlus (Linux)

1. Exécuter le programme d'installation avec la commande `./build_linux.sh`
2. Rendez-vous dans le dossier `.\build_linux\`

Démarrage + Données de tests

Nous allons maintenant pouvoir lancer le programme. **Les versions précompilées sont disponibles dans l'archive fournie avec ce document**. Sur chaque version, nous avons deux manières de lancer le programme. La première est la plus classique, c'est à dire, lancer le programme GUI (avec l'affichage graphique). Sur celui-ci, vous pouvez zoomer en dessinant un rectangle. La deuxième manière de lancer le programme est d'utiliser uniquement la partie MPI auquel cas le lancement se fait avec des arguments en ligne de commande.

Sur Windows :

GUI → Lancer le programme **FractalSharp.exe** ou bien **FractalPlusPlusGUI.exe** (en fonction du langage de programmation souhaité.

MPI → Lancer le programme **FractalSharpMPI.exe** ou bien **FractalPlusPlusMPI.exe** (en fonction du langage de programmation souhaité) de la manière suivante :

```
mpiexec –n [NombreProcessusMPI] [FractalSharpMPI.exe | FractalPlusPlusMPI.exe] [TailleX] [TailleY] [minComplexX] [maxComplexX] [minComplexY] [maxComplexY]
```

Sur Linux :

GUI → Lancer le programme **./FractalPlusPlusGUI**

MPI → Lancer le programme **./FractalPlusPlusMPI** de la manière suivante :

```
mpiexec –hostfile [NomFichierHost] –n [NombreProcessusMPI] ./FractalPlusPlusMPI [TailleX] [TailleY] [minComplexX] [maxComplexX] [minComplexY] [maxComplexY]
```

Données de tests :

Pour la version GUI, il n'y a pas vraiment de données de tests, cette version sert surtout pour vérifier le bon fonctionnement. Libre à vous de zoomer à votre convenance. Cependant, à partir d'un moment (entre le 3ème et 4ème zoom), le programme affiche que du noir, c'est parce que nous atteignons le nombre de décimales maximal pour notre algorithme de calcul.

Pour la version MPI, Les données de tests recommandées sont les suivantes :

- Calcul d'une image en **FullHD non zoomée** : `TailleX = 1920 TailleY = 1080 minComplexX = −2 maxComplexX = 2 minComplexY = −1.125 maxComplexY = 1.125`
- Calcul d'une image en **FullHD zoomée** : `TailleX = 1920 TailleY = 1080 minComplexX = −1.828 maxComplexX = −1.64 minComplexY = −0.057 maxComplexY = 0.049`
- Calcul d'une image en 16000*9000 **non zoomée** : `TailleX = 16000 TailleY = 9000 minComplexX = −2 maxComplexX = 2 minComplexY = −1.125 maxComplexY = 1.125`

Conclusion

FractalSharp et FractalPlusPlus ont été un projet très enrichissant. Se rendant compte que FractalSharp ne pourrait pas suffire aux différents tests, nous avons essayé de compiler MPI.NET sur Linux, chose assez ardue à réaliser. Cependant faire fonctionner convenablement FractalSharp sur Linux aurait pris plus de temps que de recoder le programme en C++ comme vu en cours, nous permettant de réaliser cet exercice que nous sommes bien rarement amenés à faire durant nos années d'études. Nous ne regrettons pourtant pas d'avoir réalisé dans un premier temps le programme en C#. En effet le passage d'un langage à l'autre nous a permis de repenser à comment le code avait été réalisé, aux différentes améliorations possibles. L'utilisation de C# nous a également permis de découvrir d'autres syntaxes d'utilisation de MPI et d'apprendre les processus de compilation d'un fichier .DLL pour Linux. Les performances de notre application peuvent encore être améliorées, mais nous sommes plutôt fiers d'avoir tiré des hypothèses et pu les vérifier lors des tests de performances. Enfin, pour permettre à d'autre personnes d'utiliser, d'apprendre et d'améliorer notre programme, FractalSharp ainsi que FractalPlusPlus sont disponibles en open source sur GitHub.