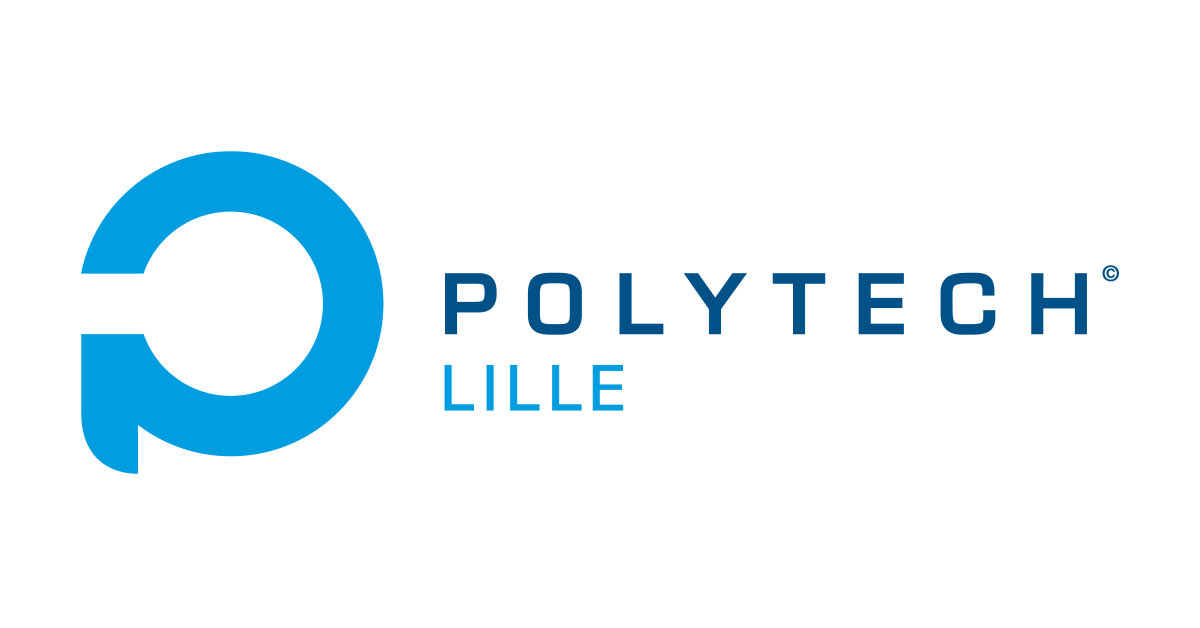
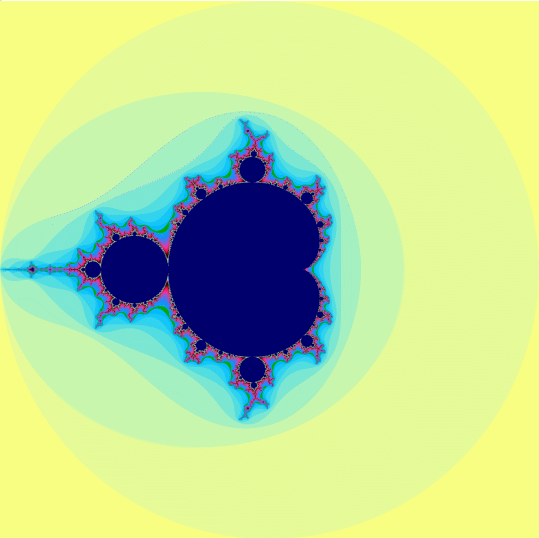
**  
  
RAPPORT DE TRAVAUX SUR L’UTILISATION DE MPI SUR L’ENSEMBLE DE MANDELBROT**



Thomas Firmin - Yann Cauchepin

IS5 - Polytech Lille - 23/10/2020

**Parallélisation de l’ensemble de Mandelbrot :**

On peut distinguer deux manières de paralléliser l’algorithme. Une première statique, en répartissant équitablement les données sur les processeurs. Une seconde en répartissant les données selon la charge de travail.

**Algorithme Statique** :

Le but de cet algorithme est de répartir équitablement les données sur tous les processeurs. Ainsi un processeur sera chargé de récupérer toutes les données calculées par les autres processeurs. Il devra aussi calculer la part de des données qu’il lui a été attribuée. Ainsi tous les processeurs travaillent.

L’image est donc divisée en blocs de ligne de largeur : *hauteur de l’image/nombre de processeur*. A l’initialisation, chaque processeur s’attribue donc le numéro de la première ligne à traiter, selon son numéro (processeur n°0,1…), et la largeur d’un bloc.  
La taille des données à traiter est donc de: *largeur image \* largeur bloc*

Ainsi on pourra distinguer deux cas:

* Le cas où le processus n’a que les données à traiter et à envoyer au processus central.
* Le cas où le processus doit traiter les données et récupérer les données des autres processus afin de reconstruire l’image. Dans l’algorithme ce sera le processus n°*0*.

**Cas du n° de processus différent de 0 :**

Le cas est simple à traiter ; le processus doit :

* Déterminer la largeur d’un bloc
* Calculer la position y de départ en fonction du n° de son processus (n° de bloc à traiter) et de la largeur d’un bloc
* Calculer les pixel correspondant aux données
* Envoyer les pixels

*Taille des message :*

On utilisera donc ***MPI\_Send***pour envoyer un message de taille *largeur\_bloc\*largeur\_image\*taille\_d’un\_pixel*. Il n’y a pas besoin d’utiliser un *Ssend* ou *Isend* puisqu’une fois que le processus a fini de calculer les pixels, il n’y a plus d’autre charge de travail à effectuer, indépendante du processus 0. Hormis l'enregistrement de l’image qui nécessite la réception de toutes les données

**Cas n° du processus égal à 0 :**

Le processus doit :

* Déterminer la largeur d’un bloc
* Calculer la position y de départ en fonction du n° de processus (n° de bloc à traiter) et de la largeur d’un bloc
* Calculer les pixel correspondant aux données et les stocker directement dans l’image finale
* Réception des pixel des autres processus et stockage des pixels au bon endroit de l’image

Réception des messages :

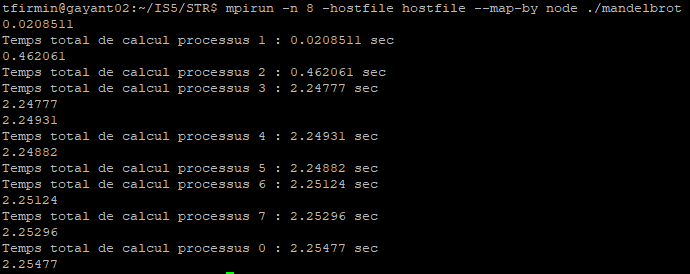
La réception est elle aussi très basique. Puisque le n° du processus indique la position des pixels reçus dans l’image. On reçoit de façon séquentiel les pixels sur chaque processus avec la boucle suivante :

*Pour i allant de 1 à nombre\_de\_processus*

Ainsi on utilise la fonction ***MPI\_Recv***, qui est une réception bloquante. En effet, on attend d’avoir reçu l’entièreté du message contenu dans le tampon avant de passer à la lecture des données du processus suivant.  
Il est important de noter que l’on stocke directement le message reçu à l’emplacement correspondant dans l’image. C’est à dire que l’on fournit à MPI\_Recv, l’espace mémoire situé à : *&image [largeur\_bloc \* w \* n°\_processus]*

**Analyse des résultats :**

On remarque que les processus ne s’exécutent pas toujours dans le même ordre ce qui est normal. Mais, le problème de la méthode statique est qu’il suffit qu’un seul processus ait une charge de travail plus élevée que les autres pour ralentir l’ensemble de l’algorithme.  
En effet la convergence du calcul de chaque pixel n’est pas fixe. Ainsi certains processus vont converger rapidement tandis que d’autres vont être très lent. On peut l’observer facilement:



Ici avec 8 processus, on remarque bien que les processus 1 et 2 se terminent bien avant les autres. Ils vont donc passer quasiment 2 secondes à ne rien faire. Tandis que les processus 4 et 5 vont déterminer une grosse partie d’environ 2.24 sec du temps global d’exécution qui est de ~2.25 sec lorsque le processus 0 se termine. Les 0.01 secondes restantes sont certainement dues aux dernières communication et aux différents traitement effectués dans le processus 0.

**Conclusion sur l’algorithme statique :**

On remarque donc sur cet algorithme qu’une répartition équitable des données n’est pas forcément la manière la plus efficace de paralléliser le programme. En effet puisque les temps de convergence peuvent varier énormément entre deux pixels. Il se peut qu’un processus acquiert dès le départ des données qui vont converger très rapidement, tandis qu’un autre aura des données avec une convergence lente. Ainsi durant le temps d’exécution globale de l’algorithme un processus dont les données aura converger rapidement, ne travaillera pas durant le reste de l’exécution. Les charges de travails ne sont donc pas équitablement réparties entre les processus.

Pour palier à ce problème, nous pouvons réaliser deux solutions :

* Implémenter un système de token où un processus qui a fini son calcul envoie un token à tous les autres processus pour savoir si il reste du travail à l’un d’eux. Auquel cas l’information est transmise aux processus et le travail peut être réparti sur d’autres processus passifs. Le programme se termine lorsque le token est envoyé à tous les autres processeurs puis retourné au processus d’origine sans notification de travail.
* On peut également implémenter le programme en répartissant le charge de travail de façon dynamique, comme le système “maître esclave”.

**Algorithme Dynamique** :

Le principe de cette implémentation est de réduire le temps de calcul total effectué, en sachant qu’il n’est pas possible de prévoir à l’avance le temps de calcul pour chaque élément de l’image. Dans le cas d’une programmation statique, le temps de calcul pour chaque bloc de lignes fournit à un processeur ne peut pas être prédit à l’avance, et certains processeurs peuvent réaliser les calculs plus vite que d’autres. D’où l’utilité de la programmation dynamique : un maître gère dynamiquement plusieurs esclaves qui réalisent des calculs plus petits afin d’optimiser la capacité de calcul total.

Il y a donc nécessité de dissocier un processus des autres afin qu’il joue le rôle de maître alors que tous les autres joueront le rôle d’esclave. Dans notre cas, nous identifions le maître avec le n° de processus égale à 0.

Le traitement de l’image s’effectue ligne par ligne, de haut en bas, de façon dynamique et non prévisible. Le maître communique avec les esclaves et les esclaves ne communiquent pas entre eux.

Le critère d’arrêt est codé par un numéro de ligne envoyé égal à *-1*. Au début le maître envoie une première ligne à tous les processus afin que les esclave rentre dans la boucle: *while(n° ligne != -1)*.

A partir du moment où toutes les lignes ont été envoyé, à chaque fois que le maître reçoit un message d’un esclave, il lui envoie par la suite un n° de ligne égal à -1.

L’inconvénient majeur de cette méthode est le temps de communication entre les processus, et le nombre de message à envoyer.

* **Cas processus n° égale à 0 :** “Maître”

Le maître utilise un entier *nb\_reception* pour comptabiliser le nombre de commandes retournées par les processus esclaves, et l’entier *i* qui comptabilise les ordres de travail envoyés, i correspond au n° de la ligne à envoyer.

Une **première étape** d’initialisation consiste à attribuer un numéro de ligne unique à chaque processus esclave pour initier le travail de tous les esclaves. Nous avons utilisé une boucle *While* dans le programme pour vérifier deux conditions : le nombre de commandes effectués aux esclaves ne doit pas dépasser le nombre de processus esclave disponible (*nb\_processus-1* car le maître ne participe pas), ni le nombre de ligne de l’image.

*Taille d’un message envoyé:  
sizeof(int)*

*Nombre de message envoyé:  
nombre de processus - 1 (on ne compte le processus maître)*

La **deuxième étape** consiste à gérer la communication avec les esclaves en observant la condition d’arrêt. Le maître reçoit les calculs de chaque ligne commandées aux esclaves et compte ces réceptions. On peut connaître l'identité du processus esclave en utilisant *status.MPI\_SOURCE* pour lui renvoyer des messages (ordre de travail ou d’arrêt).  
Pour optimiser le temps de communication, l’esclave envoie le numéro de la ligne correspondant à la ligne qu’il a calculé dans *status.MPI\_TAG*. L’appel *MPI\_Recv* contient les élements *MPI\_ANY\_SOURCE* et *MPI\_ANY\_TAG* pour permettre au maître de recevoir un message de n’importe quel esclave avec n’importe qu’elle indice de ligne. Il n’y a pas de contrainte sur la réception, le maître récupère au fur et à mesure les réceptions des esclaves sans discrimination.

Pour chaque réception de ligne calculée, deux cas sont à dissociés par le maître :

* S’il reste des lignes qui n’ont pas été envoyées aux esclaves (commandes envoyées *i <* nombre de lignes *h*) :

Le maître envoie la nouvelle ligne *i* à l’esclave qui lui a retourné la ligne calculée. *i* étant différent de -1, l’esclave continue à travailler

*Taille d’un message envoyé:*

*sizeof(int)*

*Nombre de messages envoyé:*

*nombre de ligne - (nombre processus -1)*

* Si toutes les lignes ont été commandées aux esclaves : (commandes reçues *nb\_reception* == nombre de lignes *h*) :

Le maître envoie un n° de ligne égal à -1 aux esclaves.

*Taille des messages :  
sizeof(int)*

*Nombre de messages:  
nombre de processeur -1*

Au fur et à mesure de cette seconde phase, le maître stocke le résultat de chaque ligne envoyée par les esclaves dans l’image, à l’indice ligne correspondant (*status.MPI\_TAG*).

Il est à noter que le remplissage de l’image ne s’effectue pas directement dans l’appel MPI\_*Recv* car l’indice de la ligne traité est stocké dans *status. MPI\_TAG*. On stocke donc d’abord les données dans la variable *Bloc*. Puis à l’aide du tag reçu, on place ce bloc dans l’image.

* **Cas processus n° différent de 0 :** “Esclave”

A l’initialisation du programme, l’esclave n’est pas mis au travail directement : il attend de recevoir une première ligne de la part du maître

La boucle de travail de chaque esclave permet de recevoir le numéro de ligne qu’il doit traiter, puis d’exécuter le calcul *xy2color(x,y,prof)*, d’envoyer la ligne calculé et enfin de recevoir un nouveau numéro de ligne à traiter ou bien l’ordre d’arrêt (n° ligne = -1).

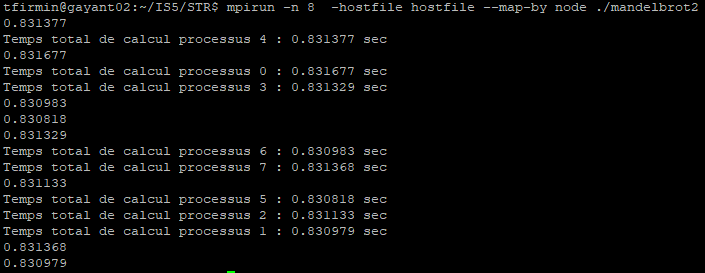
*Taille d’un message envoyé par l’esclave:*

*sizeof ((unsigned char) \* w*

*Nombre de message:  
 nombre de lignes dans l’image*

**Analyse des résultats:**

Avec un algorithme dynamique, on peut donc s’attendre à ce que l’ensemble des processus aient un temps d’exécution similaire. Parce qu’un processus esclave traitant des pixels convergeant rapidement, demandera en toute logique plus souvent du travail au processus maître.



On remarque donc bien que l’ensemble des processus ont une charge de travail à peu près équivalente. Ainsi, contrairement à la méthode statique, un processus dont la convergence est plus longue, va pénaliser beaucoup moins l’ensemble du temps d’exécution du programme.

**Conclusion de l’algorithme dynamique:**

L’algorithme dynamique permet donc de répartir de façon plus équitable les charges de travail entre les processus.

Cependant un des inconvénient de cette méthode est le nombre de communications plus élevé. En effet pour répartir au mieux la charge, on réduit le nombre de calculs effectués entre 2 échanges de données. On envoie des données de plus petite taille aux processus esclaves. Mais en contrepartie chaque processus effectue à peu près la même charge de travail.

Ainsi au bout d’un certains degré de parallélisme, c’est le temps d’envoi des messages qui déterminera la majorité du temps global d’exécution.

Enfin un dernier inconvénient, est le fait que le processur 0 ne travaille jamais sur les pixels. En effet il ne fait que s’occuper de la réception des pixels traités, de l’envoie des charges de travail, de la gestion de la condition d’arrêt des processus esclaves, et de la construction de l’image à partir des messages reçu.

Afin d’optimiser cette implémentation, nous pouvons implémenter plusieurs solutions :

* Chaque esclave doit attendre de recevoir la prochaine ligne à traiter après l’envoi de celle qu’il a calculé. Ce temps perdu pourrait être utilisé si chaque esclave avait en réserve une autre ligne pour calculer *xy2color(x,y,prof)*. Par conséquent, en phase d’initialisation, chaque esclave pourrait recevoir deux lignes ou plus (selon la comparaison du temps de calcul au temps d’attente) puis recevoir une ligne par itération. Les esclaves conserveraient donc en réserve une charge de travail traitable indépendamment du temps d’attente sur la communication.
* Le maître gère les communications et remplit l’image au fur et à mesure des réceptions des esclaves. Or, il a de nombreuse phase d’attente. C’est pourquoi, pour ne pas perdre du temps, nous pourrions utiliser du multithreading pour que le processus maître puisse faire du calcul sur une ligne en parallèle, comme un esclave.

**Analyse des vitesses et accélération :**

| Nb Proc | 2 | 4 | 6 | 8 | 10 | 12 |
| --- | --- | --- | --- | --- | --- | --- |
| Algo statique | | | | | | |
| Vitesse | 2.69 | 2.66 | 2.55 | 2.24 | 1.92 | 1.71 |
| Accélération | 2.02 | 2.04 | 2.13 | 2.43 | 2.83 | 3.18 |
| Algo dynamique | | | | | | |
| Vitesse | 5.44 | 1.84 | 1.11 | 0.83 | 0.67 | 0.59 |
| Accélération | 0 | 2.96 | 4.9 | 6.55 | 8.12 | 9.22 |

\**il n’y a pas d’accélération identifiable pour l’algorithme dynamique avec deux processeurs car au moins deux processeurs sont nécessaires (maître, esclave). On ne peut pas comparer avec un seul processeur.*

On remarque que l'accélération est beaucoup plus marquée pour l’algorithme dynamique. Par exemple une l’accélération avec 10 processeurs dans l’algorithme statique, est équivalente à l’accélération avec 4 processeurs dans l’algorithme dynamique.

L’accélération de l’algorithme 1 est dûe d’une part à l’augmentation du nombre de processeurs, mais cela implique aussi une meilleure répartition des données (moins bonne que l’algorithme dynamique). Ainsi, avec un très grand nombre de processeur, on peut espérer atteindre de bonnes accélérations, peut-être équivalente à un algorithme dynamique avec moins de processeurs.

On peut même noter que l’accélération de l’algorithme dynamique avec 2 processeurs est censée être négative, puisque ce qui change avec l'exécution sur 1 seul processeur, sont les communication entre 2 processeurs. Communication qui consomment elles même des ressources, et ralentissent dans une moindre mesure les temps d’exécution..

**Conclusion:**

Le choix de l’utilisation d’un algorithme statique ou dynamique est primordial. Il est nécessaire d’adapter ce choix au problème rencontré. En effet, dans notre problème ici, la charge de travail n’était pas prédictible et était variante. La meilleur solution était donc une répartition équitable de la charge à l’aide d’un algorithme dynamique.

Cependant, l’algorithme dynamique n’est pas toujours la meilleures solution, puisqu’il utilise énormément de communication. Ainsi, si la charge de travail des sous-problèmes est constante et prédictible, alors il faut privilégier une répartition équitable des données sur les processeurs ; dans ce cas l’algorithme statique est préférable puisqu’il minimise les communications.