TD/TP 2 et 3 : Ensemble de Mandelbrot

Version du 27 août 2015

Enseignants :

Pour le "futur" : passer a l'ensemble de Julia ? (histoire de changer un peu et d'eviter qu'ils recopient sur les annees passees...) Voir fichier "julia.c"

Dépôt des codes à la PPTI : essayer dans `/Infos/lmd/201X/master/ue`

Déroulement des 2 séances :

TD 2 :

— lecture du sujet

— expliquer : l'ensemble de Mandelbrot (dessin au tableau de l'ensemble et du cercle de rayon 2), double approximation en informatique par rapport à l'objet mathématique initial : pixel et profondeur, "mapping" entre l'image (pixels) et le plan complexe, l'algorithme séquentiel (dont calcul de la couleur du pixel (255 ou f(Nb_Iter) et boucles séquentielles), et les infos extraites de la ligne de commande (`./a.out w h xmin ymin xmax ymax prof`)

— ils cherchent seuls et écrivent l'algorithme parallèle avec équilibrage de charge statique et en version "naïve" (i.e. avec ordre de réception imposé par le processeur racine), et avec `h%NP=0`.
Attention : en M1 SFPN, décompositions 1D pas encore vues en cours.

— correction pour tout le monde au tableau

TP 2 :

— récupérer le code source séquentiel, puis essayer de générer différentes images :
  — les paramètres par défaut (vue de l'ensemble complet centré) suffisent ; il faut que le temps séquentiel dure plusieurs secondes afin qu'on puisse utiliser efficacement une dizaine de noeuds pour paralléliser le calcul (au besoin : augmenter la profondeur) ;
  — au besoin, d'autres exemples sont proposés lors de l'exécution du programme : ils permettent de mieux voir la fractale et éventuellement d'avoir plus de déséquilibre de charge, par ex. :
  ```
  ./a.out 800 800 -1.5 -0.1 -1.3 0.1 10000
  ```

— lecture du code `mandel.c` (juste les fonctions `xy2color` et `main`)

— coder l'algorithme vu en TD

— vérifier le programme parallèle sur un processeur (leur machine locale)

— éventuellement : présenter la solution avec `MPI_Gather` et coder la version améliorée (cf. code `mandel-par.c`) où on récupère en premier le travail des processus qui ont terminé avant les autres (sans tenir compte de leur rang) (et grâce à MPI_Probe())
Attention au `MPI_Gather` : en fonction de l'implémentation MPI (et de la taille des messages ?) le `MPI_Gather` peut imposer une barrière de synchronisation

— s'il reste du temps : montrer l'efficacité avec 8 processeurs avec les paramètres par défaut (l'équilibrage de charge statique est alors mauvais).
Attention au cas à 2 processeurs : l'image étant symétrique, l'équilibrage de charge est parfait dans ce cas.

— ceci doit être fini pour la prochaine séance

TD 3 :

— revenir sur les causes des insuffisances de l'équilibrage de charge statique

— étudier des solutions possibles :
  — équilibrage de charge statique avec modulo,
  — équilibrage de charge dynamique avec des bandes fines de type "maître-escalves"
  — équilibrage de charge dynamique avec des bandes fines de type "auto-régulé" (vol de tâches entre processus)

  et montrer que les meilleures solutions sont avec l'équilibrage de charge dynamique avec des bandes fines
  Attention : en M1 SFPN, équilibrages de charge statiques/dynamiques pas encore vus en cours.

— ils cherchent seuls et écrivent l'algorithme parallèle avec équilibrage de charge dynamique (avec des bandes fines de type "maître-escalves")
— correction pour tout le monde au tableau

TP 3 :

— implémentation de l'algorithme parallèle avec équilibrage de charge dynamique (avec des bandes fines de type "maître-escalves", cf. `mandel-dyn.c`)
— tests de performance
— mettre en place un recouvrement des communications par le calcul (cf. `mandel-dyn-recouv.c`)
  — premiere version : on garde les communications bloquantes (messages d"étiquette TAG_REQ courts donc envoi immédiat) et le maitre a juste à envoyer une deuxième "vague" de numéros de blocs à tous les esclaves immédiatement après la première "vague"...
  — deuxième version (pour ceux qui veulent aller plus loin) : passer les réceptions des esclaves en non bloquant et gérer deux réceptions à la fois (on garde les émissions en mode bloquant) (technique dite du "double buffering")

Remarque sur la mesure des temps : on peut mesurer le temps CPU (temps effectivement consommé par le CPU, voir par ex. : `clock()`) ou le temps «wall-clock» (ici fait avec `gettimeofday()` mais aussi possible avec `MPI_Wtime()`). En parallélisme, il faut impérativement utiliser le temps «wall-clock» car un CPU en attente d'une communication peut être bloqué au niveau système et ne plus consommer de temps CPU (ce qui fausse l'évaluation des performances du code parallèle).

Attention aux mesures de temps :

— ne pas faire d'affichage dans les calculs ou dans les comms quand on mesure les temps ;
— ne pas prendre en compte la sauvegarde de l'image dans un fichier dans les mesures de temps.

1 Introduction

L'ensemble de Mandelbrot est constitué des points c du plan complexe C pour lesquels le schéma itératif suivant :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases} \tag{1}$$

ne diverge pas. En posant $z = x + iy$ et $c = a + ib$, l'équation (1) se réécrit :

$$\begin{cases} x_{n+1} = x_n^2 - y_n^2 + a \\ y_{n+1} = 2x_n y_n + b \end{cases}$$

avec les conditions initiales $x_0 = y_0 = 0$.

On peut montrer que s'il existe un entier n tel que $|z_n| \geq 2$ (soit $|z_n|^2 = (x_n^2 + y_n^2) \geq 4$), alors le schéma (1) diverge. Pour en savoir plus, on pourra consulter [1, 2].

2 Structure des images en mémoire

Une image est un tableau à deux dimensions. Chaque élément de ce tableau s'appelle un *pixel*, abbréviation de *picture element*. La valeur de ce point est, selon le type d'image, une valeur de niveau de gris, une couleur ou une valeur de radiance. Ce tableau est organisé en mémoire ligne par ligne : on trouve la première ligne, puis la seconde, *etc*. En particulier, nous manipulerons des images codées sur un octet (taille d'un pixel), et la valeur de chaque pixel (donc un entier compris entre 0 et 255) représente un index dans une table de couleur.

Enseignants :
Nouvelle palette de couleurs (écrite par F.X. Morel, cf. COS_COLOR) : 0 ↔ jaune pâle, 255 ↔ bleu foncé.
Ancienne palette : 0 ↔ marron, et 255 ↔ noir.

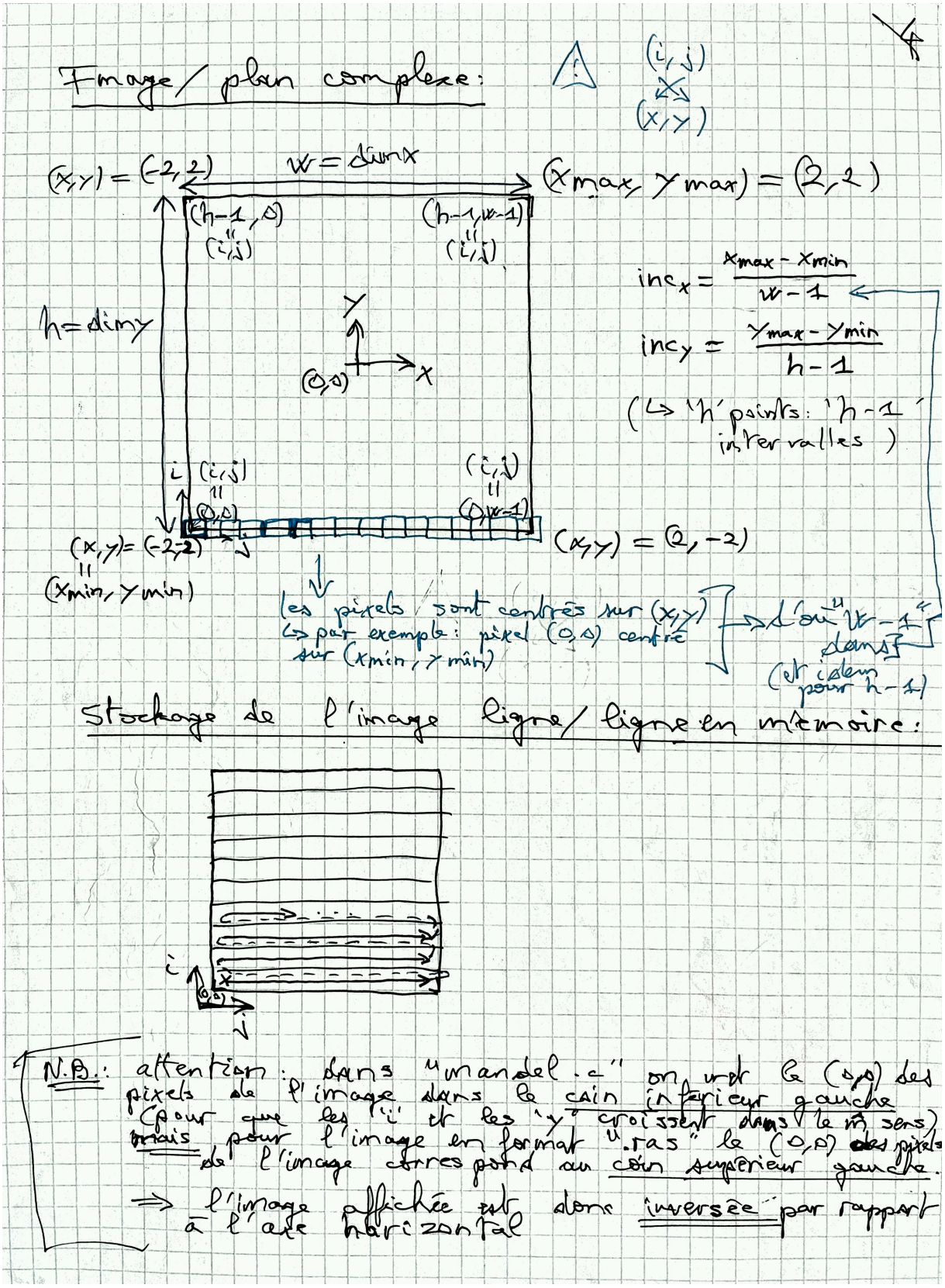
En C, cela se traduit par :

```
{
  unsigned char *Image;
  int dimx, dimy;         // largeur et hauteur de l'image
  int i,j;               // indices pour parcourir les pixels de l'image

  Image = (unsigned char *) malloc (sizeof(unsigned char)*dimx*dimy);

  for(i=0; i<dimy; i++)
    for(j=0; j<dimx; j++)
      Image[j+i*dimx] = 0;  // acces au pixel i,j
}
```

Par convention, le pixel de coordonnées $(0, 0)$ est le point supérieur gauche d'une image affiché à l'écran, et c'est donc aussi le premier élément du tableau `Image` en mémoire.

Enseignants :
Voir le scan ci-dessous pour la correspondance entre le plan complexe et l'image des pixels :

Image / plan complexe :



Stockage de l'image ligne / ligne en mémoire :

N.B. : attention : dans "mandel.c" on prend le (0,0) des pixels de l'image dans le coin inférieur gauche (pour que les 'i' et les 'y' croissent dans le même sens) mais pour l'image en format ".ras" le (0,0) des pixels de l'image correspond au coin supérieur gauche.

⇒ l'image affichée sera donc inversée par rapport à l'axe horizontal

3 Format d'image

Il existe une très grande multitude de format d'image, c'est-à-dire de façon de stocker dans un fichier une image. Nous utilisons le format Sun Rasterfile qui a le grand mérite d'être très simple à mettre en œuvre et qui est visualisable avec la plupart des programmes d'affichage d'image [1].

Un fichier rasterfile est constitué d'une entête qui décrit les caractéristiques de l'image (taille, codage, ...), suivie d'une suite de mots de 1 octet décrivant la table des couleurs (si besoin). Ensuite vient l'image proprement dite, stockée sous la forme d'un tableau de données brutes.

Enseignants :
Attention : contenu d'un fichier au format Sun Rasterfile en *big-endian byte order*. Juste besoin de conversion pour l'entête du fichier (sur processeur x86 par ex) : pas pour le corps de l'image, car constitué d'*unsigned char*.

4 Algorithme séquentiel

Synopsis de l'algorithme :

1. Pour le centre de chaque pixel de l'image :
   (a) calculer le nombre d'itérations pour lequel le schéma itératif diverge
   (nombre maximal d'itérations limité à une «profondeur» fixée par l'utilisateur) ;
   (b) mettre à jour la valeur du pixel correspondant suivant :
   Si profondeur atteinte : couleur_pixel ← 255
   Sinon : couleur_pixel ← NombreIterations % 255
2. Sauver l'image.

Enseignants :
Voici à quoi ressemblent les boucles dans le code séquentiel :

```
y = y_min
Pour i allant de 0 a h-1 par pas de 1
   x = x_min
   Pour j allant de 0 a w-1 par pas de 1
       *p++ = xy2color(x, y, prof)
       x = x + inc_x
   Fin pour
   y = y + inc_y
Fin pour
```

5 Questions

1. Discuter de la manière dont l'algorithme présenté en section 4 pourrait être parallélisé.
2. Proposer une version parallèle de cet algorithme qui répartisse de façon équilibrée les données à calculer sur chaque processeur. On utilisera les transmissions basiques de MPI (`MPI_Send` et `MPI_Recv`). En écrire le code C.

1. par exemple `display` (de la suite logicielle `ImageMagick`) sur Linux

Enseignants :

On utilise ici un découpage par bande de l'image, et pour simplifier on supposera que le nombre de processus divise exactement le nombre de lignes de l'image (cf. `mandel-par.c`).

Demander l'algorithme dans le compte-rendu (pour la Polytech).

Voici l'algorithme pour Mandelbrot avec équilibrage de charge statique et avec : $h\%NP = 0$

1: Chaque processus récupère NP (nombre de processus) et son rang P.
2: Chaque processus détermine la bande de pixels qu'il doit traiter, ainsi que les coordonnées correspondantes (à partir de la ligne de commande), et vérifie les hypothèses de travail.
Dimensions de la bande de pixels : h' lignes × w colonnes, avec $h' \leftarrow h/NP$.
Et : $x'_{min} \leftarrow x_{min}$ $y'_{min} \leftarrow y_{min} + p * h' * inc_y$
Avec : $inc_x \leftarrow \frac{x_{max}-x_{min}}{w-1}$ $inc_y \leftarrow \frac{y_{max}-y_{min}}{h-1}$
3: Allocation mémoire de la bande locale de taille $h' \times w$.
4: **Si** processus root **Alors**
5: Allocation de l'image totale
6: **Fin si**
7:
8: **Pour** i (resp. y) allant de 0 (resp. y'_{min}) à h'-1 par pas de 1 (resp. inc_y) **faire**
9: **Pour** j (resp. x) allant de 0 (resp. x'_{min}) à w-1 par pas de 1 (resp. inc_x) **faire**
10: pixel(i,j) ← calcul_couleur(x, y, profondeur)
11: **Fin pour**
12: **Fin pour**
13:
14: /*Récupération de l'image globale soit avec MPI_Gather(), soit via :*/
15: **Si** processus root **Alors**
16: Copie de la bande locale dans l'image totale
17: **Pour** s allant de 0 à NP-1 **faire**
18: **Si** s n'est pas root **Alors**
19: Réception de la bande de taille $h' \times w$ du processus s à l'adresse : image_totale + $w * h' * s$
20: **Fin si**
21: **Fin pour**
22: **Sinon**
23: Envoi de sa bande locale de taille $h' \times w$ au processus root
24: **Fin si**
25:
26: **Si** processus root **Alors**
27: Sauve l'image totale dans un fichier
28: Affiche le temps total de calcul /* car root termine en dernier ici */
29: **Sinon**
30: Affiche temps de calcul local
31: **Fin si**

Voici l'algorithme pour Mandelbrot avec équilibrage de charge statique et sans l'hypothèse : $h\%NP = 0$ (non demandé, uniquement pour les meilleurs étudiants)

1: Chaque processus récupère NP (nombre de processus) et son rang P.

2: Chaque processus détermine la bande de pixels qu'il doit traiter, ainsi que les coordonnées correspondantes (à partir de la ligne de commande), et vérifie les hypothèses de travail.
Dimensions de la bande de pixels : h'_l lignes $\times$ w colonnes,
avec $h'_c \leftarrow h/NP$ (h'_c : «h' commun»)
et $h'_l \leftarrow h'_c + (p < NP - 1\ ?\ 0\ :\ h\%NP)$ (h'_l : «h' local»).
Et : $x'_{min} \leftarrow x_{min}$ $\quad$ $y'_{min} \leftarrow y_{min} + p * h'_c * inc_y$
Avec : $inc_x \leftarrow \frac{x_{max}-x_{min}}{w-1}$ $\quad$ $inc_y \leftarrow \frac{y_{max}-y_{min}}{h-1}$
3: Allocation mémoire de la bande locale de taille $h' \times w$.
4: **Si** processus root **Alors**
5: Allocation de l'image totale
6: **Fin si**
7:
8: **Pour** i (resp. y) allant de 0 (resp. y'_{min}) à '$h'_l - 1$ par pas de 1 (resp. inc_y) **faire**
9: **Pour** j (resp. x) allant de 0 (resp. x'_{min}) à $w - 1$ par pas de 1 (resp. inc_x) **faire**
10: pixel(i,j) ← calcul_couleur(x, y, profondeur)
11: **Fin pour**
12: **Fin pour**
13:
14: /*Récupération de l'image globale soit avec MPI_Gatherv(), soit via :*/
15: **Si** processus root **Alors**
16: Copie de la bande locale dans l'image totale
17: **Pour** s allant de 0 à NP-1 **faire**
18: **Si** s n'est pas root **Alors**
19: Réception de la bande de taille $h'_c + (s == NP - 1\ ?\ h\%NP\ :\ 0) \times w$ du processus s à l'adresse : image_totale + $w * h'_c * s$
20: **Fin si**
21: **Fin pour**
22: **Sinon**
23: Envoi de sa bande locale de taille $h'_l \times w$ au processus root
24: **Fin si**
25:
26: **Si** processus root **Alors**
27: Sauve l'image totale dans un fichier
28: Affiche le temps total de calcul /* car root termine en dernier ici */
29: **Sinon**
30: Affiche temps de calcul local
31: **Fin si**

Remarque : en fait, il est plus simple pour la réception finale au niveau du root d'avoir le dernier processus avec moins d'éléments que les autres processus (on peut alors faire des MPI_Recv() avec la même taille maximum).

3. Comparer les temps d'exécution pour chaque processeur avec différents nombres de processeurs, ainsi que les efficacités parallèles obtenues. On étudiera en particulier le cas à 8 processeurs avec les paramètres par défaut.

Enseignants :
Problème à constater : on a réparti de façon équitable le nombre de pixels entre tous les processus, mais pas la charge de calcul. Or c'est la répartition équitable de la charge de calcul qui est importante pour obtenir des efficacités parallèles optimales. Avec les paramètres par défaut : l'image est symétrique, donc on doit

obtenir :
- pour 2 processus : $\sim$100% d'efficacité
- pour 4 processus : $\sim$50% d'efficacité
- pour 8 processus : $\sim$25% d'efficacité

4. L'équilibrage de charge précédent possède un gros défaut. Lequel ? Proposer un autre équilibrage de charge pour résoudre ce problème.

Enseignants :

Première solution : décomposition statique avec distribution cyclique 1D des lignes («modulo»)

Avec NP processus : la ligne numéro i est traitée par le processus $i\%NP$. Cette solution devrait être efficace en pratique pour Mandelbrot avec les hypothèses suivantes :
- deux lignes consécutives ont à peu près la même charge de calcul (c'est plutôt le cas pour Mandelbrot car l'ensemble est formé de «tâches connexes») ;
- le nombre de processus n'est pas trop grand (i.e. chaque processus récupère au moins une partie de l'ensemble : ceci est aussi nécessaire pour les solutions basées sur un équilibrage de charge dynamique (voir plus loin)).

Mais cette solution ne peut pas être optimale dans le cas général d'un calcul dont la charge n'est pas prévisible statiquement : on ne retient donc pas cette solution ici.

De plus, à la fin de cette solution chaque processus envoie toutes ses lignes en 1 seul message au processus root. Le processus root a alors besoin de recopies supplémentaires pour recopier chaque ligne à sa place dans l'image totale, ce qui implique un surcoût (autre alternative : envoyer chaque ligne séparément et précédée de son numéro de ligne (comme dans les versions avec équilibrage de charge dynamique) (ou utiliser type MPI avec «saut» (stride) ? ?)).

Deuxième solution : équilibrage de charge dynamique de type «maître-esclaves» (ou ««maître-ouvriers»)

On utilise pour cela des «bandes fines» de `nlines` lignes consécutives, et on ajoute les deux paramètres supplémentaires suivants sur la ligne de commande :
- `nlines` : nombre de lignes dans chaque bloc élémentaire (valeur par défaut : 1) $\rightarrow$ grain de calcul
- `r_master` : rang du maître (valeur par défaut : 0)

On va utiliser 2 messages pour renvoyer chaque résultat d'un esclave au maître (voir algorithmes plus loin), et donc se baser sur les 3 étiquettes (*tags*) suivants :
- TAG_REQ : message contenant un numéro de bloc (requête de travail ou réponse à une requête de travail)
- TAG_DATA : message contenant les données (pixels) d'un bloc
- TAG_END : message de terminaison

Autres alternatives :
- mettre le numéro de bloc dans le tag du message des données : pas «propre» et limité par la norme MPI à 32757 ...
- mettre le numéro de bloc dans le message avec les données (via une structure C en MPI par exemple) : implique une recopie supplémentaire des données pour chaque esclave à l'émission et pour le maître à la réception, et donc des surcoûts... (Remarque : pour les esclaves à l'émission, on pourrait allouer le buffer local directement dans la structure, mais ceci impose que $nlines$ et w soient fixés à la compilation, et on ne résoud pas le problème des recopies supplémentaires côté maître...) ;
- maintenir dans le maître un tableau pour stocker le dernier numéro de bloc envoyé à chaque esclave : bien, mais un peu moins pratique pour mettre en place l'optimisation basée sur la «2^{ieme} vague» (voir plus loin) ...

Voici l'algorithme **pour les esclaves** avec équilibrage de charge dynamique de type «maître-esclaves», sans recouvrement des communications par le calcul, et avec les hypothèses suivantes : $h\%nlines = 0$ et $nbloc \geq NP - 1$ (NP : nombre de processus total, soit 1 processus maître et $NP - 1$ processus esclaves, et donc cette dernière hypothèse implique que chaque esclave reçoit au moins un bloc).

1: Allocation mémoire d'un buffer local de taille $nlines \times w$
2: **Tant que** vrai **faire**
3: Attendre un message contenant un numéro de bloc de la part du maître (NB : attente sur *n'importe quel tag*)
4: **Si** tag == TAG_END **Alors**
5: Sortir de la boucle
6: **Fin si**
7: Faire le calcul pour le bloc correspondant de `nlines` lignes
8: Envoyer au maître le numéro de bloc (message de tag TAG_REQ) puis un message contenant les données du bloc (message de tag TAG_DATA)
9: **Fin tant que**
10: Libération mémoire du buffer local

Remarque : si les esclaves commencent par demander du travail au maître, ce message envoyé par les esclaves est en fait inutile car le maître sait quand il doit donner du travail à un esclave (au début pour tous les esclaves, et ensuite à chaque esclave qui vient de lui remettre son travail).

Voici l'algorithme **pour le maître** avec équilibrage de charge dynamique de type «maître-esclaves», sans recouvrement des communications par le calcul, et avec les hypothèses suivantes : $h\%nlines = 0$ et $nbloc \geq NP - 1$.

1: Allocation de l'image totale et vérificatien des hypothèses.
Si échec : envoi de TAG_END à tous les esclaves.
2: $nbloc \leftarrow h/nlines$
3: Envoi d'un numéro de bloc (de `nlines` lignes) à tous les processus esclaves (TAG_REQ)
4: **Tant que** il reste des blocs **faire**
5: Attente d'un message contenant un numéro de bloc de n'importe qui (TAG_REQ)
6: Récupération du rang de l'émetteur (et du numéro de bloc)
7: Réception des données pour cet émetteur (message TAG_DATA)
8: Envoi d'un nouveau numéro de bloc à cet émetteur (TAG_REQ)
9: **Fin tant que**
10: **Pour** tous les processus esclaves **faire**
11: Attente d'un message contenant un numéro de bloc de n'importe qui (TAG_REQ)
12: Récupération du rang de l'émetteur (et du numéro de bloc)
13: Réception des données pour cet émetteur (message TAG_DATA)
14: Envoi d'un message de terminaison à cet émetteur (TAG_END)
15: **Fin pour**
16: Ecriture de l'image totale dans un fichier
17: Libération mémoire de l'image totale.

Demander nouveaux temps de calcul et nouvelles efficacités (à éventuellement optimiser en fonction du nombre de lignes par bloc (`nlines`)).

Améliorations possibles :

- si $nbloc < NP - 1$: il faut compter le nombre de blocs envoyés et envoyer des messages de terminaison aux processus qui ne recoivent pas de bloc
- si $h\%nlines \neq 0$: il faut utiliser MPI_Probe() dans le code du maître pour connaitre le nombre de lignes dans chaque bloc (message TAG_DATA) et pour ne pas introduire de recopies supplémentaires.

Remarque : MPI_Probe() peut être inutile si on a des messages de `nlines` ou moins (i.e. $nbloc = (h\%nlines\ ?\ h/nlines\ :\ h/nlines + 1)$) car alors il suffit juste que l'esclave calcule et envoie le nombre de lignes exact ... (à vérifier...)

Mise en place du recouvrement comms/calcul :

- Version 1 : on garde les communications bloquantes et on se base sur le fait que les messages (de tag TAG_REQ) sont courts (1 entier) ⇒ envoi immédiat. Le maître a donc juste à envoyer une deuxième «vague» de numéros de blocs à tous les esclaves immédiatement après la première vague (le reste du code ne change pas). Gain léger mais notable en pratique.
- Version 2 (à coupler avec la Version 1) : passer les émissions des esclaves en non-bloquant pour leur faire commencer leur travail suivant (deuxième vague de la version 1) au plus tot. Peu de gain en pratique, mais recouv. comms/calcul revu en détails au TDTP4 donc inutile de creuser plus ici.
- Version 3 (très facultative) : passer les réceptions des esclaves en non bloquant et gérer deux réceptions à la fois (double buffering + besoin de MPI_Cancel()). Peu ou pas de gain en pratique.

Remarque : pas de recouv. comms/calcul côté maître car le maître ne dispose pas de calcul à faire pour recouvrir ses comms.

5. Implémenter en C l'algorithme correspondant à ce nouvel équilibrage de charge.

Références

[1] The Fractal Geometry of the Mandelbrot Set, *Robert L. Devaney* `http://math.bu.edu/DYSYS/FRACGEOM/`

[2] The Spanky Fractal Database, *Noël Giffin*, `http://spanky.triumf.ca/www/welcome1.html`