Compte rendu sur la parallélisation du calcul de l'ensemble de Mandelbrot et du produit matriciel

Pather Stevenson et Benedictus-Kent Rachmat

Enseignant : Fortin Pierre

L3 ICHP - Semestre VI - 2022

Table des matières

Calcul de l'ensemble de Mandelbrot

1.1 Présentation

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

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

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

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

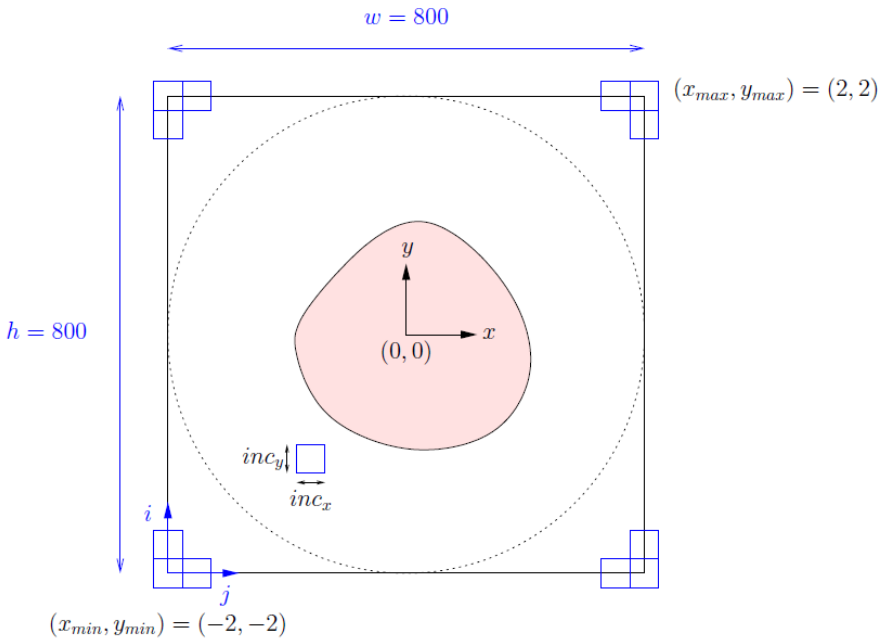
avec les conditions initiales $x_0 = y_0 = 0$.

On peut montrer que s'il existe un entier n tel que $|z_n| > 2$ (soit $|z_n|^2 = (x_n^2 + y_n^2) > 4$), alors le schéma (1) diverge.

Pour cela, nous allons devoir procéder à deux approximations :

— approximation en temps : nous n'allons pas pouvoir itérer à l'infini la suite des (z_n), et nous allons donc fixer un nombre d'itérations maximal (une profondeur) pour déterminer si la suite des $(|z_n|)$ dépasse ou non 2.

— approximation en espace : nous n'allons pas déterminer pour chaque point de l'espace entre $(x_{min}\,;\,y_{min})$ et $(x_{max}\,;\,y_{max})$ s'il appartient à l'ensemble car il y a une infinité de tels points. Nous allons donc faire le calcul pour le centre de chacun des $h \times w$ pixels, chaque pixel ayant une taille de $inc_x \times inc_y$ dans $\mathbb{R}^2$ (voir figure suivante).



avec $inc_x = \frac{x_{max} - x_{min}}{w-1}$ et $inc_y = \frac{y_{max} - y_{min}}{h-1}$.

L'algorithme séquentiel est donc le suivant :

- Pour le centre de chacun des $h \times w$ pixels de l'image, faire (fonction xy2color) :

(a) calculer le nombre d'itérations pour lequel le schéma itératif diverge (i.e. $|zn| > 2$) (nombre maximal d'itérations limité à une profondeur donnée) ;

(b) mettre à jour la valeur du pixel correspondant suivant :
— si profondeur atteinte : couleur_pixel $\leftarrow 255$
(soit la couleur correspondant à l'ensemble de Mandelbrot)

— sinon : couleur_pixel NombreIterations % 255

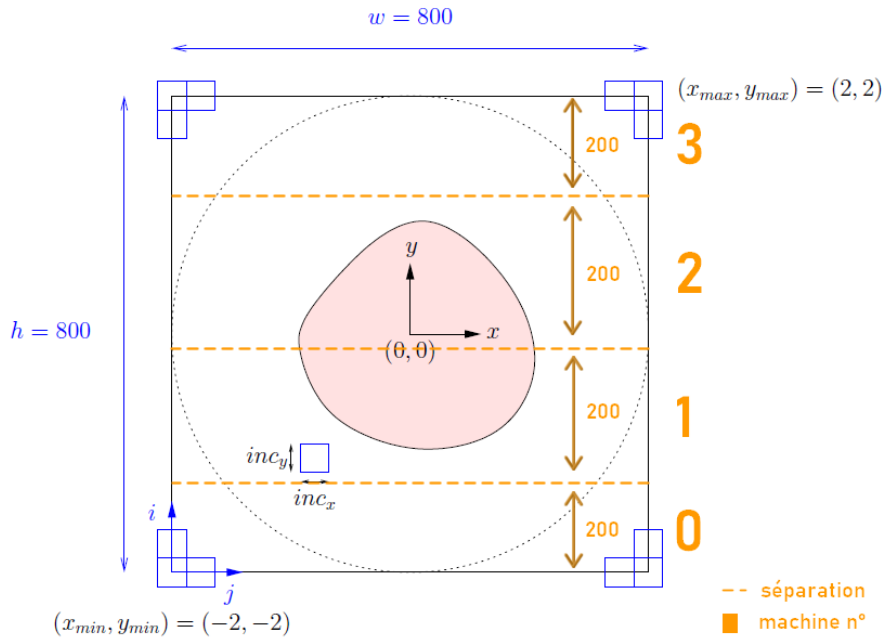
1.2 Questions

1.2.1 Quelles sont les boucles parallèles de l'algorithme ?

Le code séquentiel du programme contient deux boucles qui nous permettent de parcourir les pixels. Effectivement le coin inférieur gauche de l'image dans la présentation correspond au pixel de coordonnées $(i, j) = (0, 0)$. Il répresente donc également le premier pixel du tableau des pixels en mémoire. Ainsi on retrouve une première boucle qui itère sur les i allant de 0 à $h - 1$, dans laquelle on y trouve la seconde boucle qui itère sur les j allant de 0 à $j - 1$ dans laquelle on appelle la fonction **xy2color()** pour le pixel actuellement traité.

Ainsi il est nécessaire ici de paralléliser la boucle qui itère sur les i de façon à ce que l'on puisse lancer de manière parallèle le traitement de tous les pixels pour un i donné, donc une pour une ligne donnée. Ce traitement sera attribué a un seul thread qui effectuera le calcul pour tous les pixels de la ligne i qui lui est attribué. Par exemple si un thread doit traité tous les pixels de la ligne $i = 0$ alors ce thread effectuera le traitement pour tous les pixels de coordonnées $(0, j)$ avec j allant de 0 à $w - 1$.

Ainsi on propose ici une distribution des données en 1D par bloc. Par exemple cette distribution peut être illustrée comme ce qui suit avec 4 threads :



Dans notre exemple cette distribution par bloc donne un traitement des pixels par les threads avec cette affection :

- thread 0 : pixel de coordonnées $(0, 0)$ à $(0, 199)$,
- thread 1 : pixel de coordonnées $(0, 200)$ à $(0, 399)$,
- thread 2 : pixel de coordonnées $(0, 400)$ à $(0, 599)$,
- thread 3 : pixel de coordonnées $(0, 600)$ à $(0, 799)$

Cette distribution nécessiste donc l'hypothèse suivante :

$$h \mod |T| = 0 \tag{1.1}$$

où $h = 800$ est la hauteur et $|T|$ la cardinalité de l'ensemble des thread utilisés.

1.2.2 Comment paralléliser ce calcul en OpenMP en mode SPMD ?

Pour permettre une telle parallélisation de ce calcul en OpenMP en mode SPMD. Nous devons utiliser la directive **parallel** avec sa clause **private**

pour rendre privé deux variables x et y car nous sommes en SPMD ainsi nous sommes dans le cas de mémoire distribuée.

Les valeurs de x et y représentent les coordonnées réellées (x, y) dans le plan à chaque étape en considérant les dimensions x_{inc} et y_{inc} d'un pixel. Ainsi au départ du programme $y = y_{min}$ et à chaque nouvelle itération de la boucle i, $x = x_{min}$. Ce qui permet de les donner en argument à la fonction **xy2color()**. Ainsi comme nous donnons la directive suivante :

 # pragma omp parallel private(x,y)

Le traitement attribué de chaque thread peut être lancé en parallèle sans que leur variables x et y soient partagées. La directive complète est donc donné avant la boucle qui itère sur i et avant l'initialisation de $y = y_{min}$.

Dans ce contexte la boucle qui itère sur i doit être modifiée de façon à permettre à chaque thread de ne pas utiliser la même valeur de i et/ou faire un chevauchement sur la distribution d'un thread "voisin" dans le schéma. Dans une distribution 1D par bloc une telle boucle itère donc pour des valeurs comprises entre :

$$t \times \frac{h}{|T|} \leq i < (t+1) \times \frac{h}{|T|} \tag{1.2}$$

avec t le numéro de thread (exemple si $|T| = 4$ alors $0 \leq t < 3$).

1.2.3 Quelles sont les efficacités parallèles obtenues pour différents nombres de threads ?

Nous avons pu produire à l'aide de notre programme et d'un script pour génrer les graphiques suivants qui nous permettent d'exposer le temps de cacul. Mais aussi l'accélération et l'éfficacité par rapport à une exécution par un seul thread du traitement. Nous rappelons que :

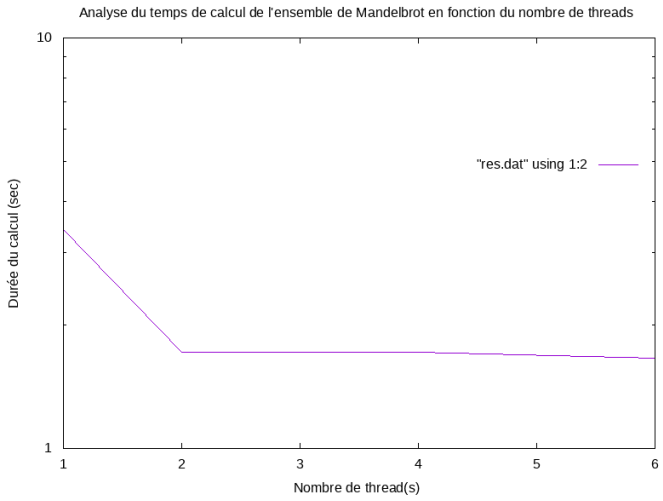
$$acc = S_p = \frac{T_1}{T_p} \tag{1.3}$$

où T_i est le temps d'exécution de notre programme parallèle sur i unités de calculs.

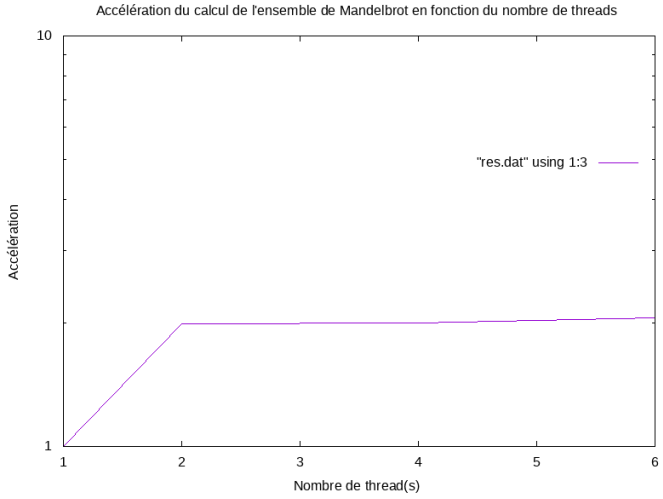
Pour l'efficacité :

$$eff = \frac{S_p}{p} = \frac{T_1}{p \times T_p} \tag{1.4}$$

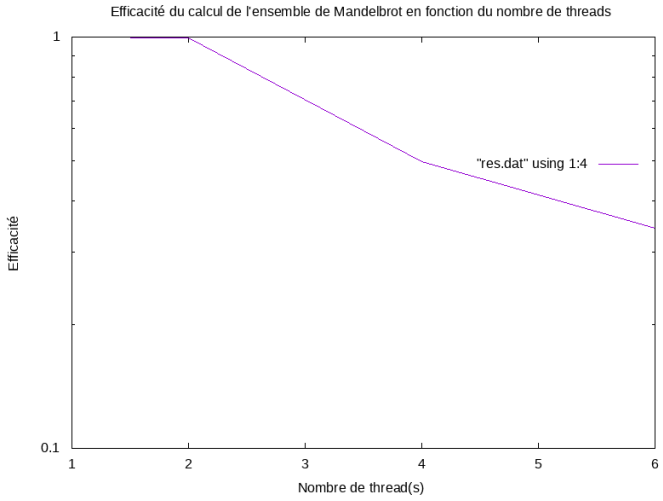
Ainsi voici les résultats obtenus pour notre solution proposée :
Durée du calcul :



Accélération :



Efficacité :



Ainsi on peut constater une perte de performance quand le nombre de threads augmente avec une accélération qui tend à être constante. Ce qui représente ici une efficacité parallèle de 33% pour 6 threads contre 100% pour 2 threads. Nous nous attendions pas à ce résultat mais en observant mieux la répartition par bloc du calcul de l'ensemble nous avons pu nous tourner sur une piste pour améliorer les performances dans la section suivante. Effectivement il y a possiblement un déséquilibre dans la répartition de la charge du traitement de l'ensemble à calculer.

1.2.4 Comment améliorer les performances obtenues ? (toujours en mode SPMD)

Nous pouvons constater une perte de performance dans l'efficacité de parallélisation quand on faisait augmenter le nombre de threads. Ce qui nous a paru étonnant dans un premier temps. Mais ceci est dû à une mauvaise répartition de charge. Effectivement un pixel qui se trouve au centre de l'ensemble obligera la fonction **xy2color()** a itérer jusqu'à une profondeur maximale qui est fixée à 10000 dans notre programme. Ainsi certains threads auront une charge de calcul moins importante s'ils ont la tranche la plus haute ou la plus basse dans la distribution par bloc. Car c'est là où il y figure le moins de pixels qui sont dans l'ensemble, c'est-à-dire qui diverge avant le nombre d'itération maximal de notre contexte.

Ce que nous pouvons faire est un équilibrage de charge dynamique centralisé au niveau des lignes. Cela consiste à lancer en parallèle le traitement de t_x lignes qui seront traitées par t_x threads. Avec au plus un thread par ligne. Mais dès qu'un thread aura terminé le traitement de sa ligne alors il commencera le traitement de la ligne suivante non traitée la plus proche. Ici par non traitée l'on veut dire non traitée et non en cours de traitement par un autre thread. Ce qui va permettre aux threads qui auraient eus bloc avec un nombre de pixels très peu présent dans l'ensemble. Et qui donc auraient terminés avant les autres threads qui auraient eu eux beaucoup de pixels présent dans l'ensemble. De ne pas être en attente et de ne pas utiliser cette perte de capacité pour permettre de terminer plus rapidement le traitement de tous les pixels. Le déséquilibr est donc ici réduire à une ligne et non à des blocs de lignes.

Il est nécessaire de modifier la directive donnée en OpenMP en mode SPMD. La boucle qui itère sur i se voit supprimé au profit d'une variable compteur i qui est initialisé à 0. Puis nous lançons une boucle tant que, qui s'arrêtera quand notre variable compteur sera égale à h donc 800. Nous venons maintenant inscrire une première directive pour permettre de paralléliser le traitement des lignes :

 # pragma omp parallel private(x,y,i_{cp})

les variables x et y ont déjà été introduites précédemment. la variable i_{cp}

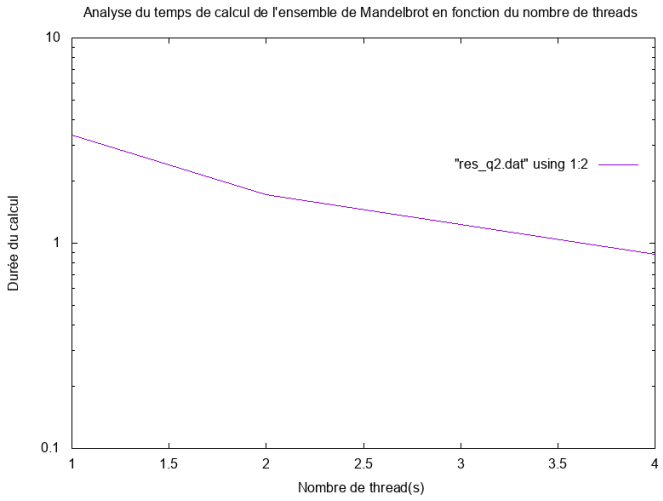
représente une copie de la variable i qui correspond au numéro de ligne. Nous déclarons ensuite une autre directive :

 # pragma omp critical

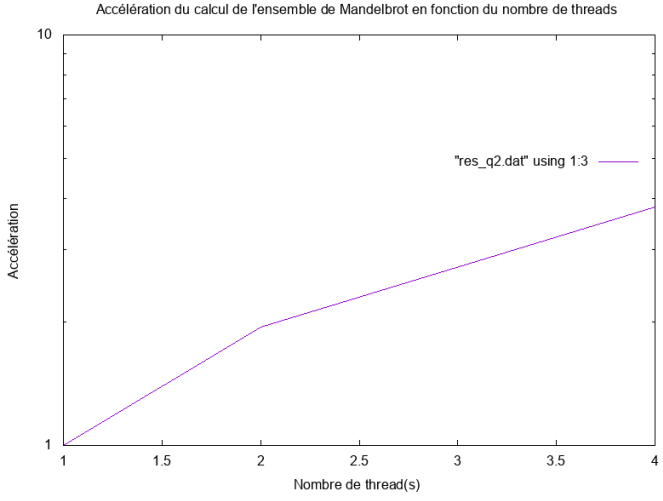
Pour permettre la déclaration d'une section critique. Effectivement dans cette section critique nous allons permettre aux threads d'effectuer une copie de la variable i puis d'incrémenter i pour indiquer que la ligne i va être en cours de traitement et que la ligne la plus proche non traitée et non en cours de traitement sera la ligne $i+1$ au moment de la sortie de la zone critique d'un thread qui y sera rentré. Cette directive critique permet donc un système de verrous, si un thread rentre dans cette zone critique alors cette zone ne peut être exécutée par aucun autre thread qui souhaiterait y rentrer. Et ceux tant que le premier thread qui y soit rentré n'y soit pas sorti. Ainsi c'est pour cela que nous avons dû ajouter la variable i_{cp} en private dans la directive de parallélisation. En sortie de cette zone critique les variables y et x sont initialisés, y devient égale à $y_{min} + (i_{cp} \times y_{inc})$ et x reste initialisé à x_{min} comme nous sommes à ce moment dans le cas d'un thread qui commencera une ligne. Vient-ensuite la boucle qui itère sur j qui elle ne change pas et dans laquelle on retrouve toujours l'appele à la fonction **xy2color()**.

Ainsi nous avons pu produire les résultats de notre nouvelle approche de parallélisation du calcul de cet ensemble :

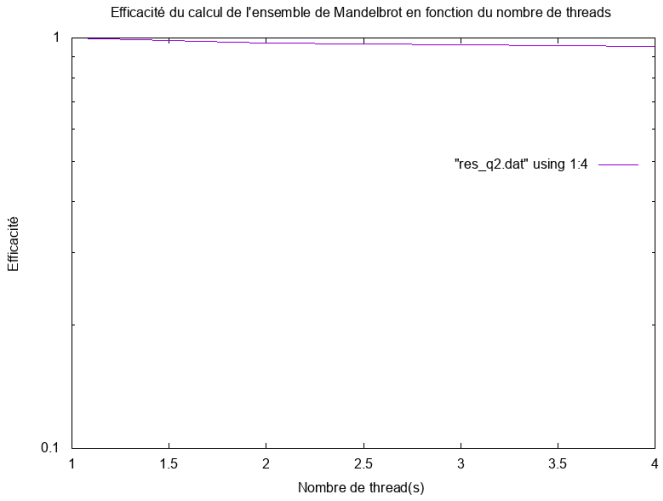
Durée du calcul :



Accélération :



Efficacité :



On peut observer que maintenant la courbe du temps de calcul et donc de l'accélération expose bien un gain en temps de calcul de l'ensemble quand le nombre de threads augmente. La courbe de l'efficacité nous montre également que notre solution pour permettre un meilleur équilibrage des charges a fonctionné. Ce qui représente ici un gain d'environ 73% en temps de calcul. Ainsi on peut conclure sur le fait que notre parallélisation est maintenant correcte est intéressante.

Calcul du produit matriciel

2.1 Présentation

On considère un code de produit matriciel du type $C = A \times B$ pour des matrices carrées d'ordre N. Pour rappel, le produit matriciel consiste à calculer chaque élément de la matrice C ainsi :

$$C_{i,j} = \sum_{k=0}^{N-1} A_{i,k} \times B_{k,j}, \ \ \forall (i,j) \in [\![0, N-1]\!]^2 \tag{2.1}$$

En d'autres termes, l'élément $C_{i,j}$ est le résultat de produit scalaire de la ligne i de A et de la colonne j de B.

2.2 Questions

2.2.1 Identifier la ou les boucle(s) parallèles du produit matriciel dans le code séquentiel.

Dans le code séquetiel on retrouve une boucle qui itère sur les i qui représente le numéro de ligne actuel du parcours de la matrice A. Puis nous avons une boucle qui itère sur les j qui représente donc le numéro de colonne actuellement parcouru de la matrice B. On trouve ensuite une boucle qui itère sur une variable k qui va représenté le numéro de case/l'index actuel dans la ligne i pour la matrice A et le numéro de case/l'index dans la colonne j de la matrice B. Donc le k de la matrice A sera en largueur et en hauteur pour la matrice B. Et enfin on trouve l'incrémentation du résultat du calcul

de la case de la matrice C de coordonées (i, j) qui est obtenu par la formule indiquée en présentation. Ici on a :

$$0 \leq i < N$$
$$0 \leq j < N$$
$$0 \leq k < N$$

avec N qui est l'ordre.

Ainsi dans ce contexte il est intéressant de paralléliser la boucle qui itère sur les i.

2.2.2 Etablir la stratégie de parallélisation multi-thread adaptée

Pour permettre la parallélisation du produit matriciel nous devons tout d'abord procéder à une modifcation du code séquentiel qui est nécessaire comme nous allons effectuer une parallélisation de la boucle i. Il est alors intéressant de boucler en deuxième sur k pour permettre pour une case de coordonées (i, k) dans la matrice A. De ne être parcouru plusieurs fois par un thread lors du calcul de la matrice C. C'est-à-dire d'assigner à chaque thread l'incrémentation habituellement dans les cases de C pour le calcul de la matrice C de la ligne i mais en faisant varié j à chaque fois avant k pour ajouter la valeur du produit entre la case (i, k) et la case (k, j). Ainsi on de cette façon les threads ne repassent pas deux fois sur la même case de la matrice i lors de leur calcul pour leur i donnés. Ce qui va permettre un gain de performance considérable. Ainsi on retrouve dans l'ordre :

- Boucle qui itère sur les i :

    - Boucle qui itère sur les k :

        - Boucle qui itère sur les j :

            - $C_{i \times N,j} \leftarrow C_{i \times N,j} + (A_{i \times N,k} * B_{k \times N,j})$

avec N l'ordre des matrices et $0 \leq i, j, k < N$.

Ce qui revient donc comme nous l'avons exposé à une inversion de l'ordre entre les boucles j et k du code séquentiel.

2.2.3 Implémenter votre parallélisation en OpenMP

Nous donnons donc la directive OpenMP suivante qui se trouve avant la boucle qui itère sur i :

 # pragma omp parallel for private(j,k)

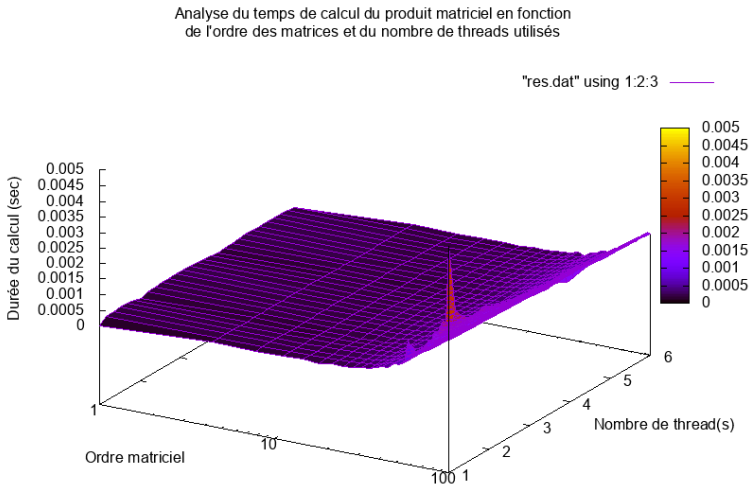
Ici il est nécessaire d'indiquer que chaque thread doit avoir sa propre valeur/variable de j et k de manière privée comme le traitement pour le(s) i donnés des threads sont lancés en parallèle. La directive for permet elle une distribution par bloc au niveau des lignes qui est décidé par OpenMP en fonction du nombre de threads disponibles et de leur états à chaque cylce. Par exemple avec 4 threads le premier peut par exemple faire les incrémentations dans les cases de la matrice C pour tous les i allant de 0 à $\frac{N}{4}$, le second les i allant de $\frac{N}{4} + 1$ à y, etc.. Avec N l'ordre des matrices qui sont carrées.

2.2.4 Calculer les efficacités parallèles obtenues avec différents nombres de threads, et analyser les performances obtenues.

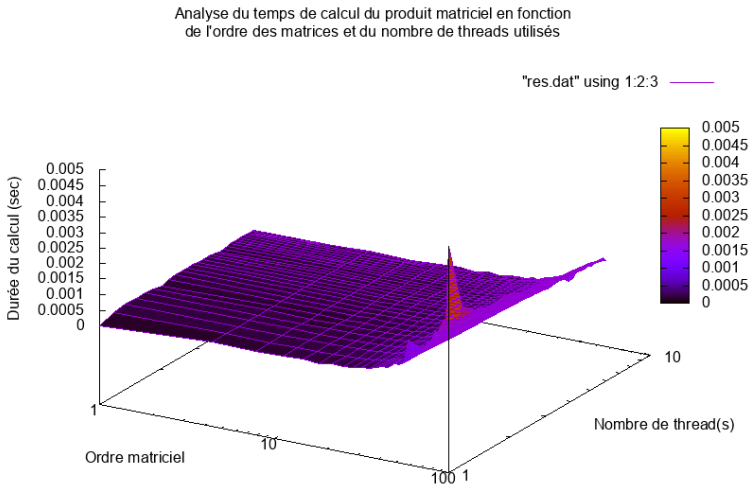
Après avoir vérifié que l'exécution parralèle est correcte au niveau des coins de la matrice. Nous lançons un script qui va nous permettre à l'aide de gnuplot de produire un graphe pour le temps de calcul du produit matriciel et du nombre d'opérations en virgule flottante par seconde (Gflops/s) en fonction de l'ordre de la matrice et du nombre de threads utilisés :

Pour le temps de cacul :

Avec logscale x :



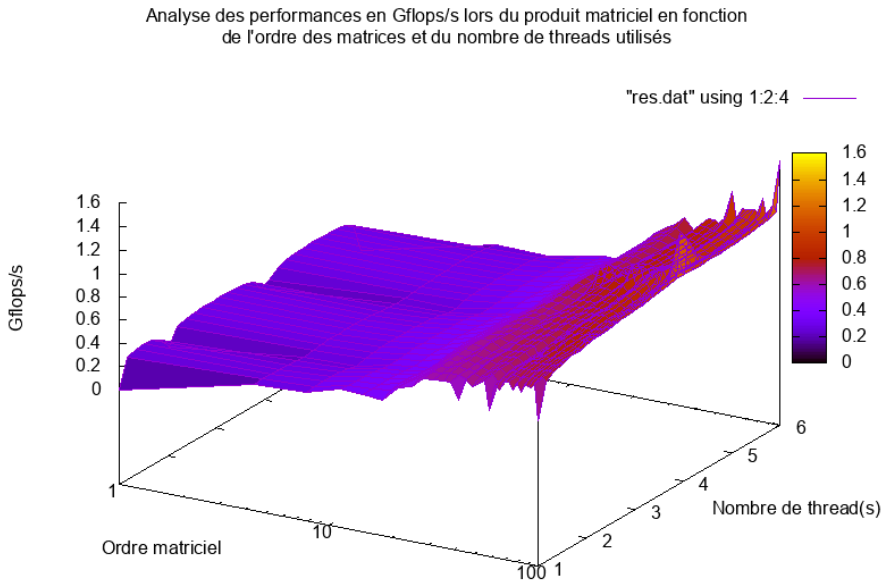
Avec logscale x et y :



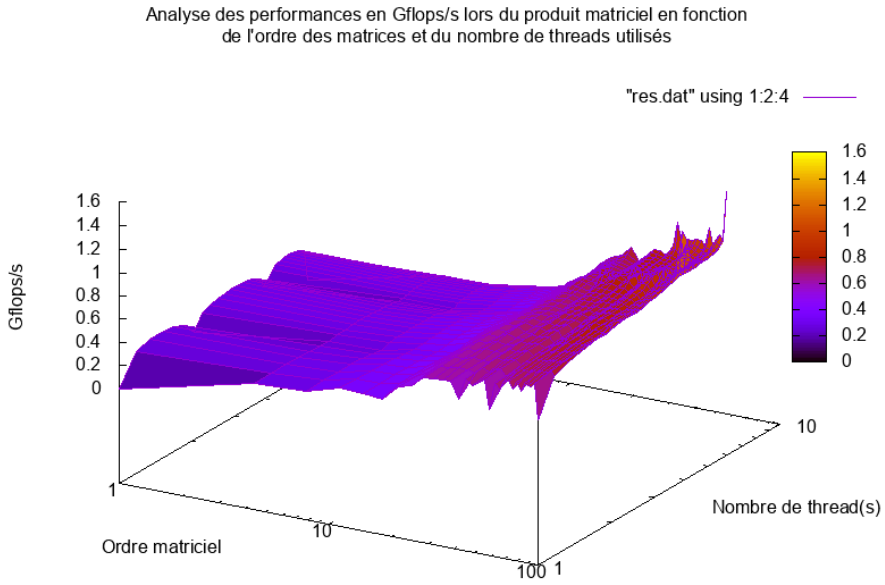
On constate facilement ici que l'on obtient un résultat élevé en temps de calcul quand l'ordre augmente vers 100. Et que ce temps est diminué par l'augmentation du nombre de threads. Ici on obtient une réduction du temps de calcul d'environ 71% entre 1 thread utilisé contre 6 threads utilisés pour un ordre de matrice $N = 100$.

Pour le nombre de flops/s :

Avec logscale x :



Avec logscale x et y :



On peut constater que concernant le taux de Gflops/s celui-ci augmente naturellement quand l'ordre des matrices augmente. Mais on peut observer qu'il augmente aussi lorsque le nombre de threads augmente. Notamment par exemple pour $N = 100$, on observe nettement des piques positifs d'augmentations avec un nombre de threads plus elévés et des piques négatifs d'augmentations lorsque ce nombre de threads diminue.

Nous pouvons donc en conclure que nos résultats de parallélisation du produit matriciel en terme de performances sont satisfaisants et cohérents.