Romain
Corbeau
M2 IMIS

Compte rendu TP 4 : VTK + MPI

Pour générer un rendu en parallèle, on doit indiquer des groupes de données différentes à chaque processus. Pour se faire, on divise la lecture des données en modifiant le *zStart* et le *zEnd* de chaque processus.
Cette modification s'effectue dans la fonction « *ParallelReadGrid* ».
Ensuite, on partage le *zBuffer* entre tous les processus en appliquant un opérateur de réduction minimal puis on recompose l'image sur le processus root.
Cette modification s'effectue dans la fonction « *CompositeImage* »

I) ParallelReadGrid

```
vtkRectilinearGrid* ParallelReadGrid() {
    int zCount = parRank < gridSize % parSize
                 ? (gridSize / parSize) + 1
                 : gridSize / parSize;
    int zStart = parRank < gridSize % parSize
                 ? ((gridSize / parSize) + 1) * parRank
                 : ((gridSize / parSize) + 1) * (gridSize % parSize) +
                   (gridSize / parSize) * (parRank - (gridSize % parSize));
    int zEnd = zStart + zCount;

    if (parRank == parSize - 1)
        zEnd -= 1;

    return ReadGrid(zStart, zEnd);
}
```

Ici, on calcul le *zCount* qui correspond au nombre d'élément de la grille devant être lu pour chaque processus. Grâce à la condition ternaire, on détermine le nombre précis d'élément, y compris si la taille de la grille (*gridSize*) n'est pas divisible par le nombre de processus (*parSize*).

Ensuite, on calcul le *zStart* qui correspond à l'index du tableau de donnée auquel le processus doit commencer sa lecture. A nouveau, on vérifie si la taille de la grille (*gridSize*) est divisible par le nombre de processus. Si ce n'est pas le cas, on doit d'abord commencer par compter le nombre d'élément prit par les premiers processus (processus auxquels on a attribue un élément de plus) puis le nombre d'élément prit par les processus suivant.

Enfin, on calcul le *zEnd* qui correspond simplement au *zStart + zCount* à l'exception du dernier processus où on doit enlever 1 au *zCount* afin de ne pas dépasser la taille des données.

II) CompositeImage

```
bool CompositeImage(const float *rgbaIn, float *zBuffer, float *rgbaOut, int imageWidth, int imageHeight) {
    int nPixels = imageWidth * imageHeight;
    auto *zBufferMin = new float[nPixels];

    MPI_Allreduce(zBuffer, zBufferMin, nPixels, MPI_FLOAT, MPI_MIN, MPI_COMM_WORLD);

    auto *rgbaTmp = new float[4 * nPixels];
    for (int i = 0; i < nPixels; ++i) {
        if (zBuffer[i] <= zBufferMin[i]) {
            rgbaTmp[i * 4 + 0] = rgbaIn[i * 4 + 0];
            rgbaTmp[i * 4 + 1] = rgbaIn[i * 4 + 1];
            rgbaTmp[i * 4 + 2] = rgbaIn[i * 4 + 2];
            rgbaTmp[i * 4 + 3] = rgbaIn[i * 4 + 3];
        }
    }

    MPI_Reduce(rgbaTmp, rgbaOut,  count: 4 * nPixels, MPI_FLOAT, MPI_MAX,  root: 0, MPI_COMM_WORLD);

    delete[] zBufferMin;
    delete[] rgbaTmp;

    return true;
}
```

Dans cette fonction, on doit d'abord appliquer une réduction avec un opérateur minimal sur l'ensemble des données contenus dans la variable « *zBuffer* » de chaque processus. Une fois cette opération faite par l'intermédiaire de la communication collective « *MPI_Allreduce* », on peut effectuer une comparaison entre les différents *zBuffer* (le zBuffer donnée en entrée de la fonction et celui réduit sur chaque processus).

Pour ce faire, on parcourt chaque pixel si ce dernier possède un *zBuffer* inférieur au *zBuffer* réduit, alors on sait que ce pixel doit être au premier plan donc on affecte chaque valeur du tableau de sortie local « *rgbaOut* » par la valeur contenu dans le tableau d'entrée « *rgbaIn* ».

Lorsque toutes les données sont organisées en fonction de la profondeur à laquelle elles sont affecté, on réduit toutes les données locales de chaque processus vers le processus root en appliquant un opérateur maximal. On aurait également pu faire un « *MPI_Gather* » à la place.

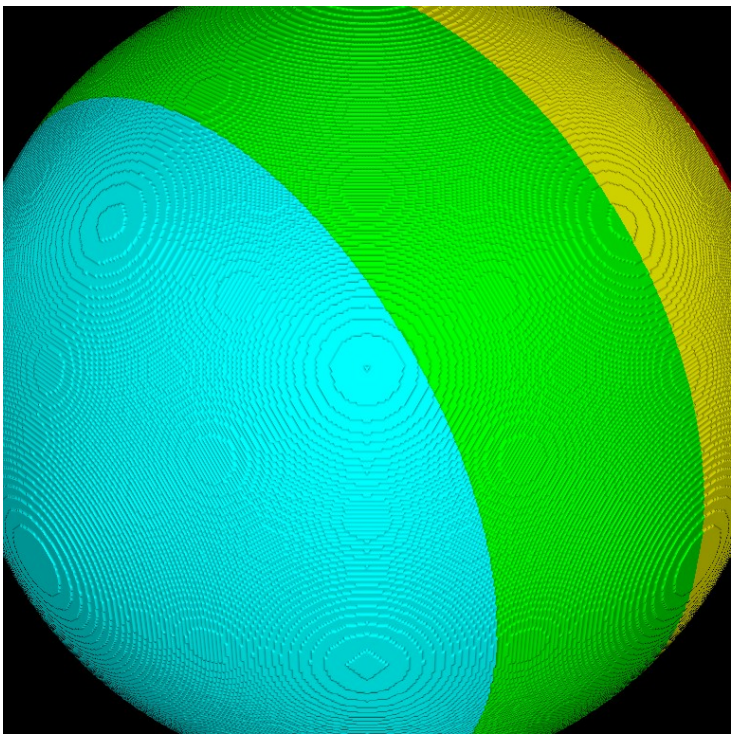
Pour finir, on libère la mémoire utilisée localement dans chaque processus par le « *rgbaTmp* » et le « *zBufferMin* ».

III) Conclusion

Pour conclure, le résultat obtenu est semblable au TP précédent car les calculs ont simplement étaient parallélisés sur chaque processus. Le temps nécessaire pour obtenir l'image finale est légèrement inférieur pour cet exemple mais les données ne sont pas suffisantes pour obtenir un résultat de valeur avec une parallélisation.
De plus, il est compliqué de réaliser une vraie comparaison car les machines exécutant le code produit ne sont pas adaptée pour réaliser une parallélisation optimale.

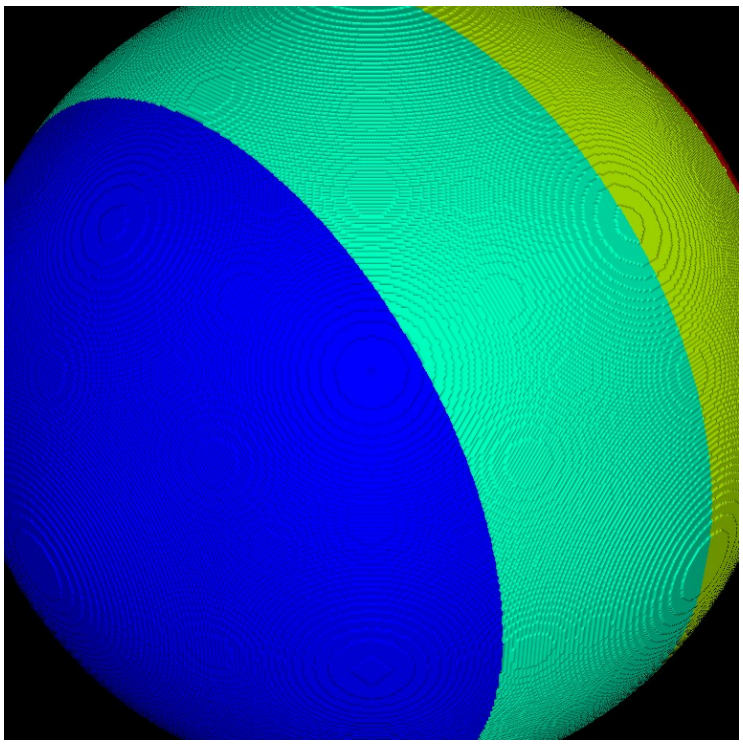
Résultat issus du TP 3 :

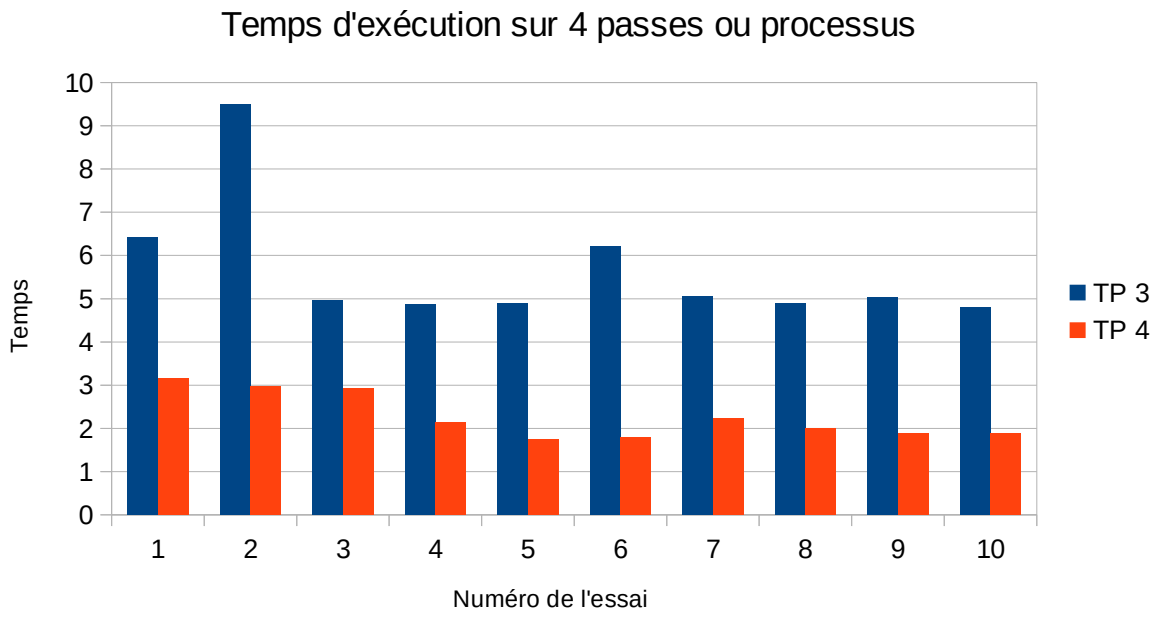
Temps total moyen sur 4 passes pour 10 essais: 5,66 secondes



Résultat issus du TP 4 :

Temps total moyen sur 4 passes pour 10 essais: 2,27 secondes




Temps d'exécution sur 4 passes ou processus
Temps
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
Numéro de l'essai
TP 3
TP 4