Programmation parallèle et distribuée(2012)

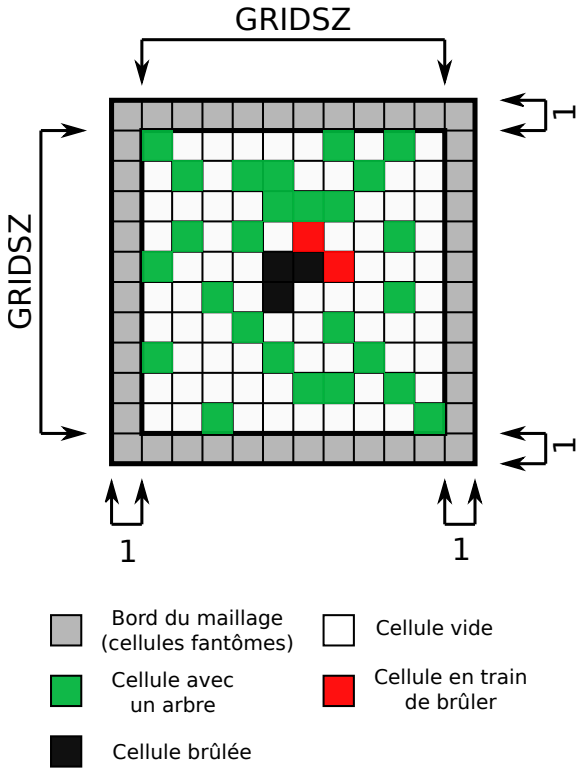
MPI : Message Passing Interface
Mini-projet

sylvain.didelot@exascale-computing.eu
jean-yves.vet.ocre@cea.fr

I Burning-Trees : un simulateur de feu de forêt

Dans le cadre de ce mini-projet, vous devrez travailler sur un programme simulant un feu de forêt. Les sources du programme séquentiel vous sont fournies. Vous devrez à partir de cette version séquentielle, réaliser une version parallèle en utilisant la bibliothèque MPI. Le but de ce mini-projet est de vous emmener pédagogiquement à réaliser cette version parallèle en répondant à des questions. Le travail sera à réaliser seul et un mini-projet sera à rendre par étudiant. Vous trouverez tous les détails de la notation à la subsubsection 2.5

1 Description du simulateur



Vous disposez d'un domaine (grille) carré de taille GRIDSZ. Cette grille contient GRIDSZ*GRIDSZ cellules. Chaque cellule du domaine peut avoir 4 états possibles qui sont les suivants :

– Cellule vide (*empty, blanche*) : la cellule est vide, elle n'a pas d'arbre.
– Cellule avec arbre (*tree, verte*) : la cellule contient un arbre.
– Cellule en train de brûler (*burning, rouge*) : la cellule contient un arbre en train de brûler.
– Cellule brulée (*burned, noire*) : la cellule contient un arbre brûlé.

Au début du programme, chaque cellule du domaine a une probabilité p de contenir un arbre ($p = 0,60$). Le temps du simulateur est discrétisé (avancement de la simulation suivant un pas de temps). Une fois le nombre d'itérations MAX_STEPS atteint, la simulation se termine et le programme quitte. Au temps t_0 une case est mise en feu au centre du domaine. Le feu est soumis à des règles strictes : il ne peut se déplacer que de proche en proche, de gauche à droite et de haut en bas. Il ne peut pas se déplacer sur les diagonales.

Au cours du simulation, le feu peut être amené à se déplacer suivant les règles suivantes :
– Une cellule ne brûle que si elle contient un arbre qui n'a pas encore été brûlé.
– Une cellule se met à brûler au temps *t+1* si au moins une cellule voisine est en train de brûler au temps *t*.
– Une cellule en train de brûler au temps t change d'état en cellule brulée au temps *t+1*
– Une cellule brulée au temps t ne brûlera pas au temps *t+1*, même si une cellule voisine est en train de brûler au temps *t*.

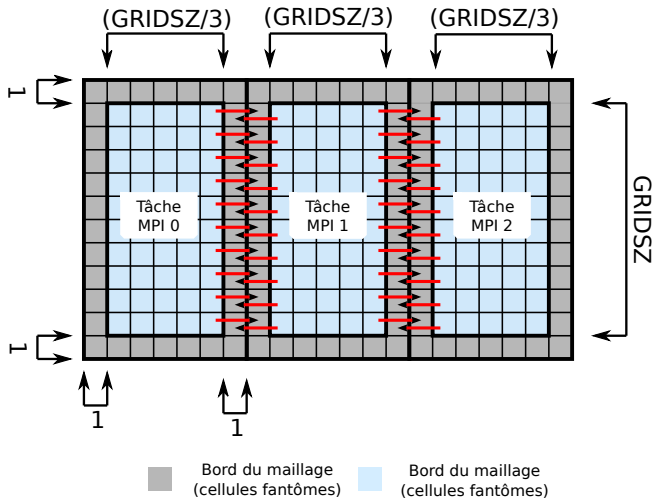
A chaque bord du domaine se trouvent des cellules fantômes de taille 1. Vous constaterez que ces cellules fantômes sont déjà présentes et utilisées dans la version séquentielle. Dans le code sequentiel, ces cellules fantômes sont initialisées comme des cellules vides (sans arbre) et demeurent vides durant toute la simulation. Cela permet à la fonction *update_cells* de rester simple et ne nécessite pas de traitement particulier pour les cellules au bord du domaine. Dans le code parallèle, ces cellules fantômes serviront à recevoir les cellules situées au bord des sous-domaines des tâches voisines.

Une fois la simulation terminée, vous verrez apparaitre dans votre répertoire de travail un fichier *output_ ?.gif*. À chaque itération, le simulateur crée une nouvelle image qu'il rajoute à la fin de ce fichier GIF. Vous pouvez ouvrir le fichier pour vérifier comment votre simulation s'est déroulée.

– Si vous souhaitez reprendre les TPs chez vous, il vous faudra installer la librairie *GD*. Cette bibliothèque est utilisée dans le but de générer les fichiers de sortie GIF. Sur un système Ubuntu/Linux, vous pouvez l'installer de la manière suivante : *sudo apt-get install libgd2-xpm-dev*

2 Réalisation du programme parallèle

2.1 Echange des cellules fantômes



Nous allons maintenant réaliser le programme parallèle de notre simulateur en utilisant la bibliothèque MPI. La décomposition du sous-domaines se fera en 1 dimension avec deux voisins : un à gauche, le second à droite. Nous découperons le domaine comme indiqué sur le schéma ci-dessus.

Les cellules fantômes détaillées précédemment vont permettre de recevoir des données venant des tâches MPI voisines. Ainsi au moment de parcourir la grille, chaque tâche MPI connaitra l'état des cellules voisines, même si celles-ci ne font pas partie du sous-domaine courant de la tâche MPI.

1. Remplissez convenablement la fonction *initialize_mpi* afin que chaque tâche MPI soit bien initialisée.
2. Remplissez convenablement la fonction *finalize_mpi* afin que chaque tâche MPI se termine correctement.
3. Modifiez le Makefile des sources afin de compiler le programme avec MPI.
4. Divisez le domaine en sous-domaines. Faites en sorte que chaque tâche MPI ait une sous-partie du domaine à calculer et que le fichier de sortie généré soit de la même taille que le sous-domaine calculé (modifiez l'appel à la fonction *display_render_step*).
5. Modifiez le contenu de la fonction *exchange_ghost_cells* afin que chaque tâche MPI reçoive les données de ses voisines dans les cellules fantômes. Faites bien attention à l'ordre des *send* et des *recv* pour éviter les deadlocks.

2.2 Gestion du GIF de sortie

Actuellement, chaque tâche MPI génère son propre fichier GIF de sortie (i.e : *output_0.gif, output_1.gif*, etc..) Nous souhaitons maintenant que seule la tâche MPI de rang 0 génère un fichier GIF de sortie et que celui-ci contienne tous les sous-domaines de toutes les tâches MPI.

1. Modifiez la fonction *gather_grid* afin qu'un seule tâche construise une grille qui contienne tous les sous-domaines de toutes les tâches MPI. Modifiez ensuite l'appel à *display_render_step* de manière à ce qu'un seul fichier de sortie contenant tout le domaine soit généré.

2.3 Condition de fin de la simulation

Jusqu'à maintenant, la simulation ne s'arrêtait qu'au moment où le nombre d'itérations atteignait la valeur de *MAX_STEPS*. Modifiez la fonction *check_final_conditions* de manière à ce que la simulation s'arrête si et seulement si plus aucune cellule de la simulation n'a besoin d'être mise à jour.

2.4 Temps d'exécution du simulateur

À la fin de la simulation, toutes les tâches affichent leur temps passé à calculer. Modifiez la fonction *print_time* de manière à ce que seule la tâche 0 affiche le temps total (de toutes les tâches) passé à calculer.

2.5 Consignes pour la notation

- Le projet est à réaliser seul et un projet par étudiant devra être rendu.
- N'hésitez pas à modifier les valeurs des constantes *MAX_STEPS, GRIDSZ et RATIO* suivant vos besoins.
- Vous n'avez pas à modifier une seule ligne du code contenu dans le fichier *mpi_display.c*. Ce fichier ne s'occupe que de la génération du fichier de sortie et ne comporte aucune partie MPI. Pour plus d'informations, regardez les commentaires du fichier *mpi_display.h*.

3 Annexes

3.1 Algorithme séquentiel du programme

```
function UPDATE_NEIGHBOR(x, y, nb_burning_cells)
    if grid[x][y] = tree then
        if nb_burning_cells > 0 then
            grid_next[x][y] = burning
        else
            grid_next[x][y] = tree
    else if grid[x][y] = burning then
        grid_next[x][y] = burned
    else if grid[x][y] = burned then
        grid_next[x][y] = burned

function UPDATE_SELF(x, y, nb_burning_cells)
    if grid[x][y] = burning then
        nb_burning_cells ++

function UPDATE_CELLS
    nb_burning_cells
    for x = 1 to x < GRIDSZ do
        for y = 1 to y < GRIDSZ do
            left = x - 1
            right = x + 1
            top = y - 1
            bottom = y + 1

            UPDATE_NEIGHBOR(left, y, nb_burning_cells)
            UPDATE_NEIGHBOR(right, y, nb_burning_cells)
            UPDATE_NEIGHBOR(x, top, nb_burning_cells)
            UPDATE_NEIGHBOR(x, bottom, nb_burning_cells)
```

```
        UPDATE_SELF(x, y, nb_burning_cells)

function MAIN
    Initialisation grid
    Initialisation grid_next
    for t = 0 to MAX_STEPS do
        UPDATE_CELLS
        Echanger grid avec grid_next
```