École Centrale Lyon

MSO 3.7

Système temps réel, embarqué et mobile

Rapport

Rapport

Élèves :
Fabien Duranson
Julien Bronner

Enseignant :
Alexandre Saïdi

Introduction

Lors de ce cours et des différents BE associés, nous avons vu des méthodes d'accélération d'exécution via la parallélisation des calculs sur les différents coeurs du processeurs. L'intérêt de diminuer le temps de calcul est notamment l'utilisation de programme en temps réel, pour que la réponse soit instantanée, mais aussi dans le cas de systèmes embarqués, disposant souvent d'une faible puissance de calcul. Répartir les opérations permet donc d'optimiser cette puissance.

Principe de fonctionnement du programme principal

À l'exécution du fichier compilé, le programme principal attend une entrée de l'utilisateur pour savoir quel sous-programme exécuter.

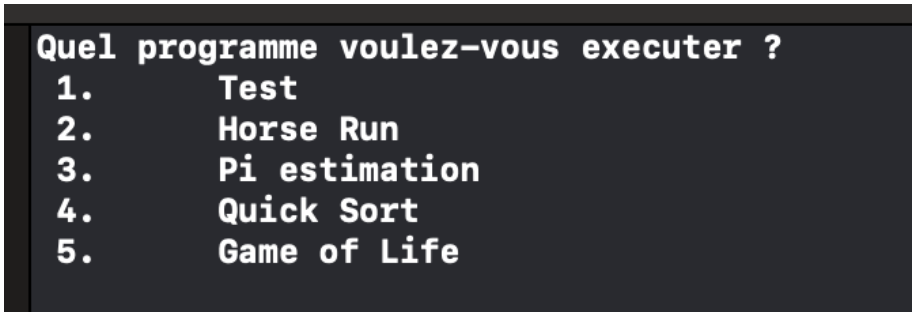

FIGURE 1 – Interface de sélection du sous-programme à executer

Il suffit de rentrer le chiffre correspondant au sous-programme voulu et de valider avec la commande retour à la ligne.

Le programme **Test** permet de vérifier que le mutli-threading fonctionne : si les lettres A, B, C et D apparaissent dans l'ordre (AAA...ABBBB...BCCC..CDDD), il n'y a **pas** de mutli-threading, sinon, ça fonctionne.

En fonction du programme choisi, d'autres inputs sont nécessaires lors de l'exécution.

Nous détaillons ci-après les différents programmes.

I. Méthode de calcul de Pi : Monte-Carlo

I.1 Principe

La méthode de Monte-Carlo permet d'approcher la valeur de pi en tirant au hasard un grand nombre de points 2D dans [0,1] x [0,1] et en comptant la proportion de points contenus dans le quart de cercle unité.

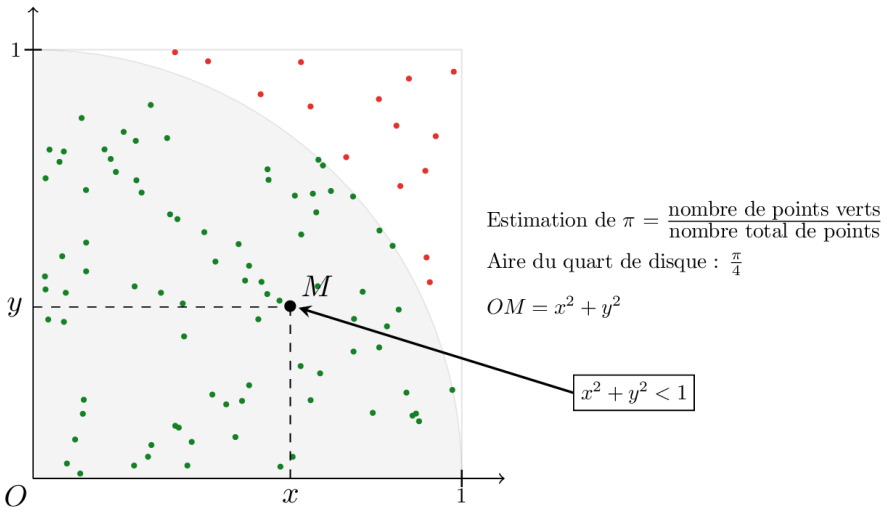

FIGURE 2 – Principe de la méthode de Monte-Carlo pour l'estimation de la valeur de *pi*

La parallélisation est basique : si on veut effectuer $m \times N$ tirages aléatoires et les répartir en m threads, on va simplement faire faire N tirage aléatoire à chaque thread puis faire la moyenne de toutes les proportions obtenues pour donner l'estimation finale de pi.

I.2 Implémentation

Le fichier dédié est *pi_ circle.hpp*.
Voici les deux fonctions que contient ce fichier :

```
1  double randFloat() { return (double) std::rand() / (double)
       RAND_MAX; }
2
3  double estimate_pi(int n_iter_subfunc)
4  {
5      int count = 0;
6      for (int i = 0; i < n_iter_subfunc; i++)
7      {
8          double x = randFloat();
9          double y = randFloat();
10         if (x*x + y*y < 1) count++;
11     }
```

```
12      return 4. * (double) count / (double) n_iter_subfunc;
13  }
```

La fonction **estimate_pi** est la fonction que va executer chaque thread. À noter qu'on multiplie par 4 le résultat avant de l'envoyer car la méthode l'algorithme tel quel estime la valeur de $\frac{\pi}{4}$.

Dans le fichier *main.cpp*, c'est la fonction **piCircle** qui se charge de répartir l'exécution en threads et de récupérer les résultats :

```
1  double piCircle(int n_point, int n_thread)
2  {
3      double pi = 0.;
4      std::future<double> results[n_thread];
5      for(int i = 0; i < n_thread ; i++)
6          results[i] = std::async(std::launch::deferred,
       estimate_pi, n_point / n_thread);
7      for(int i = 0; i < n_thread ; i++)
8          pi += results[i].get();
9      return pi / n_thread;
10 }
```

On utilise `std::future` ici et non `std::threads` car on doit stocker et récupérer un résultat de type **double** à la fin de l'exécution d'un thread.

I.3 Résultats

Exécution avec 10^7 points et 1 thread : *Résultat : Pi = 3.14116 Temps pour 1 threads = 178ms*

Exécution avec 10^7 points et 8 threads : *Résultat : Pi = 3.14211 Temps pour 8 threads = 180ms*

On remarque ici que le temps de calcul est le même alors que les threads sont bien toutes lancés et que les tâches sont bien réparties. Ceci peut s'expliquer par certaines fonctions qui ne peuvent peut être pas être parallélisées et sont donc obligées de se faire séquentiellement malgrès le mutli-thread.

II. Méthode de Tri : Quick Sort

II.1 Principe

L'algorithme de tri *Quick Sort* fonctionne sur le principe **Diviser pour régner**. En effet, pour trier une liste, on choisit un pivot (en principe souvent le premier élément), et on trie les autres éléments de la liste en deux autres suivant s'ils sont plus ou moins grands que le pivot. On trie ensuite récursivement ces deux listes via ce principe (jusqu'à obtenir une liste avec un seul élément qui est donc par essence triée), et on fusionne dans l'ordre, la liste plus petite que le pivot, le pivot et celle plus grande. On obtient ainsi une liste triée.

Dans le cas de la parallélisation, l'idée est répartir sur des threads différents le tri de chaque sous-liste, puis de fusionner les résultats. Pour cela on sépare la liste de taille n de base en m listes, où là k-ème sous-liste contiendra les éléments qui seront entre les places $(k-1)*m$ et $k*m-1$; c'est-à-dire que la liste est triée par paquet, et que les éléments de chaque paquet resteront dans une place de ce paquet. Ensuite, on applique sur chaque sous-liste un tri *Quick Sort*, en utilisant un thread à chaque fois. Enfin, on fusionne chaque liste triée, et l'on obtient la liste de base triée, puisque chaque sous-liste était à sa place.

II.2 Implémentation

Le fichier dédié est *quick_sort.hpp*.

Il y a tout d'abord les deux fonctions composant le tri rapide, **combine** qui fusionne deux listes et un pivot, et **quick_sort** qui exécute un Quick Sort classique. Il y a donc l'étape de séparation en deux listes par rapport au pivot, puis dans le *return*, il a les étapes de tri récursives, et la fusion via la fonction **combine**.

```
1  std::vector<double> combine(std::vector<double> v_inf, double
       pivot, std::vector<double> v_sup)
2  {
3      v_inf.push_back(pivot);
4      v_inf.insert( v_inf.end(), v_sup.begin(), v_sup.end() );
5      return v_inf;
6  }
7
8  std::vector<double> quick_sort(std::vector<double> v)
9  {
10     if (v.size() <= 1)
11         return v;
12     double pivot = v[0];
13     std::vector<double> inf;
14     std::vector<double> sup;
15     for (int i = 1; i < v.size(); i++)
16     {
17         if (v[i] < pivot) {
```

```
            inf.push_back(v[i]);
        } else {
            sup.push_back(v[i]);
        }
    }
    return combine(quick_sort(inf), pivot, quick_sort(sup));
}
```

Il y a ensuite les fonctions dédiées au multi-threading. Ainsi, **merge** permet de fusionner une liste de liste, dans l'ordre de celle-ci. Cela permet de reconstruire la liste triée à partir de chaque paquet trié par chaque thread. Ensuite, la fonction **split** permet de diviser une liste en sous-listes, et dont les éléments resteraient au sein du paquet pour la liste triée. Ainsi, la liste est triée par bloc, les blocs ne sont pas triés, mais le maximum d'un bloc est plus petit que le minimum du bloc suivant.

```
std::vector<double> merge(std::vector<std::vector<double>>
   v_list)
{
    std::vector<double> v;
    for (int i = 0; i < v_list.size(); i++)
    {
        std::vector<double> v2 = v_list[i];
        v.insert( v.end(), v2.begin(), v2.end() );
    }
    return v;
}

std::vector<std::vector<double>> split(std::vector<std::
   vector<double>> v_list, int depth, int maxDepth)
{
    if (depth >= maxDepth)
        return v_list;
    std::vector<std::vector<double>> v_list_return;
    for (std::vector<double>& v : v_list)
    {
        double pivot = v[0];
        std::vector<double> inf;
        std::vector<double> sup;
        for (int i = 1; i < v.size(); i++)
        {
            if (v[i] < pivot) {
                inf.push_back(v[i]);
            } else {
                sup.push_back(v[i]);
            }
        }
```

```
30          sup.push_back(pivot);
31          v_list_return.push_back(inf);
32          v_list_return.push_back(sup);
33      }
34      return split(v_list_return, depth + 1, maxDepth);
35  }
```

Enfin, la fonction générale **quick_sort_multi_threading** utilise les fonctions présentées précédemment. On force l'utilisation d'une puissance de deux en nombre de thread, en faisant $2^{log2(n\_thread\_input)}$, avec $log2$ le logarithme en base 2, et n_thread_input le nombre de thread demandé par l'utilisateur.

Ensuite, on sépare en sous-liste via la fonction **split** pour obtenir la liste de sous-liste triée par paquet, on ouvre un thread pour chaque paquet, et on applique la fonction **quick_sort** sur chaque thread, et donc pour chaque paquet. Une fois que chaque thread a fini, on peut fusionner chaque paquet retourné en une seule liste via **merge** et ainsi avoir la liste triée.

```
1  std::vector<double> quick_sort_multi_thread(std::vector<
       double> v, int *nThreads)
2  {
3      int maxDepth = floor(log2(*nThreads));
4      *nThreads = pow(2,maxDepth);
5      if (v.size() < 2 * *nThreads)
6      {
7          std::cout << "Pas de mutli-thread ici." << std::endl;
8          return (quick_sort(v));
9      }
10     else
11     {
12         std::vector<std::vector<double>> v_list; v_list.
       push_back(v);
13         std::vector<std::vector<double>> splitted = split(
       v_list, 0, maxDepth);
14
15         std::vector<std::future<std::vector<double>>>
       splitted_sorted_future(splitted.size());
16         std::vector<std::vector<double>> splitted_sorted(
       splitted.size());
17
18         for(int i = 0; i < splitted.size() ; i++)
19             splitted_sorted_future[i] = std::async(std::
       launch::deferred,quick_sort, splitted[i]);
20         for(int i = 0; i < splitted.size() ; i++)
21             splitted_sorted[i] = splitted_sorted_future[i].
       get();
22         return merge(splitted_sorted);
```

```
23        }
24    }
```

II.3 Résultats

Le programme créé par défaut une liste de taille 10^6 (modifiable dans le code) dont les valeurs sont comprises entre -10^6 et 10^6. Ensuite il demande à l'utilisateur de rentrer le nombre de threads qu'il veut affecter au calcul.

On affiche la liste triée et le temps de calcul nécessaire à l'obtention du résultat :

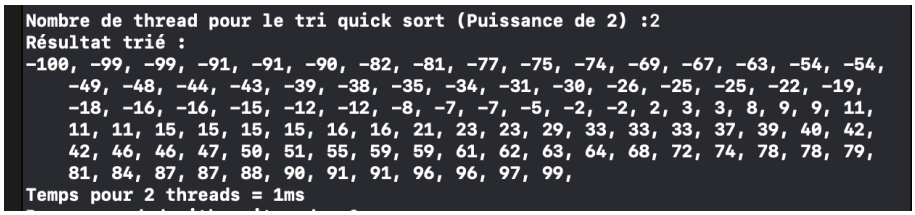

FIGURE 3 – Résultat du programme Quick Sort sur un échantillon de donnée de taille 100

Voici le résultat pour :

— 100 élements et 2 threads : 1ms
— 10^6 élements et 2 threads : 4944ms
— 10^6 élements et 8 threads : 2783ms

La liste est toujours correctement triée.

III. Horse run

III.1 Principe

Il s'agit ici de simuler une course hippique entre différents chevaux. Chaque cheval sera désigné par une lettre et aura une ligne attribuée (cf Figure 4).

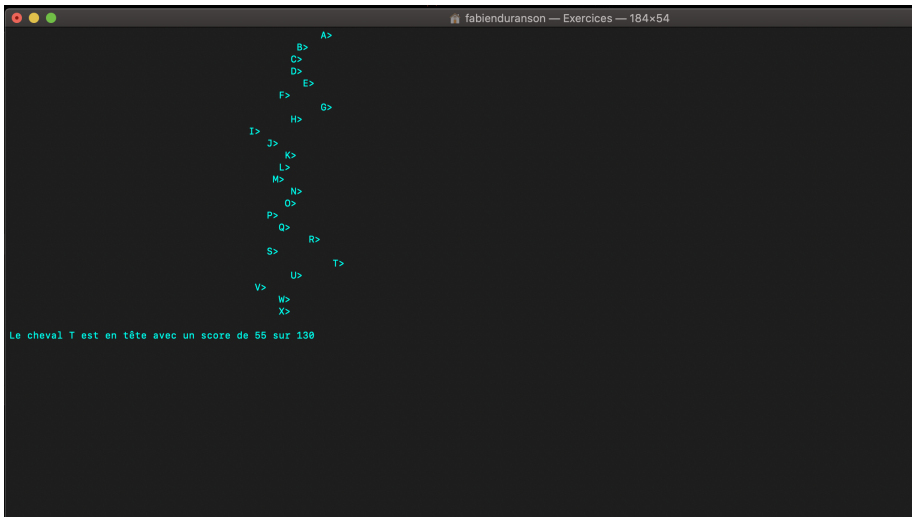

FIGURE 4 – Capture d'écran de l'exécution du programme Horse Run

Chaque cheval sera commandé par un thread dédié qui le fera avancer après avoir attendu un temps aléatoire.

Un arbitre est également présent pour vérifier à tout moment quel cheval est en tête (un thread lui est dédié).

III.2 Implémentation

Le fichier dédié est *horse_run.hpp* :

```
1 char nth_letter(int n)
2 {
3     return "
  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"[n -
  1];
4 }
5 void moveto(int lig, int col) { printf("\033[%d;%df",lig, col
  ); };
6 void erase_scr() { printf("\033[2J");}
7 void erase_line() { printf("\033[1K");}
8 void set_curseur_visible() { printf("\x1B[?25h"); }
9 void set_curseur_invisible() { printf("\x1B[?25l"); }
```

```
10
11 void courir(int ma_ligne, int nColonnes, std::mutex *mtx, std
   ::vector<int> *scores) {
12     std::vector<int> &sc = *scores;
13     moveto(ma_ligne, 30);
14     char mon_signe[3] = {nth_letter(ma_ligne), '>', 0};
15     for (int i=0; i < nColonnes; i++)
16     {
17         mtx->lock();
18         moveto(ma_ligne, i);
19         erase_line();
20         moveto(ma_ligne, i);
21         puts(mon_signe);
22         sc[ma_ligne - 1] += 1;
23         mtx->unlock();
24         std::this_thread::sleep_for(std::chrono::milliseconds
   (50*(rand()%3+1)));
25     }
26 }
27
28 void arbitrer(int nHorses, int nColumns, std::mutex *mtx, std
   ::vector<int> *scores)
29 {
30     std::vector<int> &sc = *scores;
31     int ma_ligne = nHorses + 2;
32     int first_horse_id = 0;
33     int first_horse_score = 0;
34     while (*std::min_element(sc.begin(),sc.end()) < nColumns)
35     {
36         first_horse_id = std::max_element(sc.begin(),sc.end()
   ) - sc.begin();
37         first_horse_score = *std::max_element(sc.begin(),sc.
   end());
38         mtx->lock();
39         if (first_horse_score == nColumns){
40             moveto(ma_ligne + 1, 0);
41             std::cout << "Le cheval " << nth_letter(
   first_horse_id + 1) << " a gagné !" << std::endl;;
42             mtx->unlock();
43             return;
44         } else {
45             moveto(ma_ligne, 0);
46             erase_line();
47             moveto(ma_ligne, 0);
```

```
48              std::cout << "Le cheval " << nth_letter(
   first_horse_id + 1) << " est en tête avec un score de " <<
    first_horse_score << " sur " << nColumns;
49          }
50          mtx->unlock();
51          std::this_thread::sleep_for(std::chrono::milliseconds
   (100));
52      }
53      return;
54  }
```

On définit ici des fonctions utilitaires pour afficher ce qu'on veut dans la console et les deux tâches à effectuer pour une thread : **courir** ou **arbitrer**. On peut noter ici l'utilisation de *mutex* qui permet de réserver l'utilisation des ressources à un seul thread pour éviter que plusieurs threads modifient la même ressource en même temps (ici on utilise mutex pour écrire dans la console et modifier le tableau de résultats intermédiaires).

Dans le fichier *main.cpp*, on répartit les tâches en différentes threads comme expliqué précédemment :

```
1  double horseRun(int nColonnes, int nHorses)
2  {
3      std::vector<std::thread> tab_id1(nHorses);
4      std::vector<int> tab_num_ligne_a_ecran(nHorses); // les
   valeurs envoy  ees  aux threads srand(time(NULL));
5      std::vector<int> scores(nHorses);
6      erase_scr(); // On efface  l  ecran
7      set_curseur_invisible();
8      moveto(nHorses + 2,1);
9      mtx.lock();
10     std::cout << "Creations des threads \n";
11     mtx.unlock();
12     for (int i = 0; i < nHorses; i++)
13     {
14         tab_num_ligne_a_ecran[i] = i + 1;
15         scores[i] = 0;
16         tab_id1[i] = std::thread(courir,
   tab_num_ligne_a_ecran[i], nColonnes, &mtx, &scores);
17     }
18     std::thread arbitre = std::thread(arbitrer,nHorses,
   nColonnes, &mtx, &scores);
19     for (int i = 0; i< nHorses; i++){
20         tab_id1[i].join();
21     }
22     arbitre.join();
23     moveto(nHorses + 4,2);
```

```
24      std::cout << "\n\n\n\n Fin de main\n";
25      set_curseur_visible();
26
27      return 0;
28  }
```

Étant donné que l'on attend aucun valeur de retour des fonctions **courir** et **arbitrer** (elle modifient directement l'affichage de la console), on utilise ici des *std : :thread*.

III.3 Résultats

N.B. L'exécution doit se faire dans une console à part car la console de l'IDE n'est peut être pas compatible avec les commandes d'affichage que l'on utilise.

La longueur de la course est fixée à 130 dans le code et le nombre de chevaux doit être entré par l'utilisateur lors de l'exécution (plus de 52 fera planter le programme à cause des lettres disponibles).

Lorsqu'un cheval arrive à 130, il est déclaré vainqueur par l'arbitre. Lorsque tous les chevaux sont arrivés le programme termine.

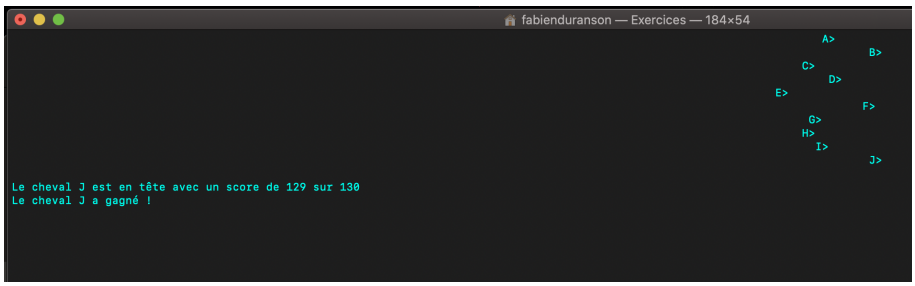

FIGURE 5 – Capture d'écran de la fin de la course programme Horse Run

En testant avec plusieurs valeurs, on remarque que le temps reste le même peut importe le nombre de chevaux ce qui montre que la parallélisation est efficace :

- *Temps pour 4 threads = 14487ms*
- *Temps pour 30 threads = 14988ms*

IV. Mini-projet : Jeu de la Vie

IV.1 Principe

Le jeu de la vie est un automate cellulaire, basé sur des règles simples, qui permet de faire apparaître des formes semblables à de la vie (d'où son nom). L'idée est de considérer une grille de jeu, composée de cellules mortes ou vivantes, et on avance par étape. À chaque étape, l'état d'une cellule dépend de son état précédent et de celui de ses voisines les plus proches (les huit qui l'entourent).

Les règles sont :

— Une cellule morte entourée de exactement trois voisines vivantes devient vivante (elle naît) ;

— Une cellule vivante entourée de deux ou trois cellules vivantes reste vivante, sinon elle meurt.

L'idée est donc de générer ce système, et de voir son évolution via un affichage basique. Pour la parallélisation, nous avons décidé de séparer la grille en sous grille, et chacune sera calculée par un thread différent.

IV.2 Implémentation

Il y a dans l'implémentation, d'une part la gestion d'une grille de booléen pour connaître l'état de la grille à un instant t, et d'autre part son affichage.

IV.2.a Gestion de la grille

Puisque le nombre de voisins est une donnée essentielle, il y a tout d'abord une fonction comptant le nombre de voisins autour d'une cellule donnée.

```
1  int n_neighbors(std::vector<std::vector<bool>> grid, int i,
       int j)
2  {
3      // Indices de 1 à n car les bords sont à false
4      int countNeighbors = 0;
5      if (i < 1 || j < 1 || i > grid[0].size() - 1 || j > grid.
       size() - 1)
6          return 0;
7      // S
8      if (grid[j][i - 1])
9          countNeighbors++;
10     // SE
11     if (grid[j + 1][i - 1])
12         countNeighbors++;
13     // E
14     if (grid[j + 1][i])
15         countNeighbors++;
16     // NE
```

```
    if (grid[j + 1][i + 1])
        countNeighbors++;
    // N
    if (grid[j][i + 1])
        countNeighbors++;
    // NO
    if (grid[j - 1][i + 1])
        countNeighbors++;
    // O
    if (grid[j - 1][i])
        countNeighbors++;
    // SO
    if (grid[j - 1][i - 1])
        countNeighbors++;
    return countNeighbors;
}
```

Ensuite, il y a une fonction pour gérer le changement d'état. En regardant attentivement les règles, on constate qu'une cellule est en vie à coup sûr si au tour d'avant elle était entourée de trois cellules vivantes exactement, elle sera dans le même état que précédemment s'il y en avait deux, et sera morte sinon. C'est cette forme là qui sera utilisée dans la fonction **process_cell**.

```
void process_cell(int x, int y, std::vector<std::vector<bool
   >> grid, std::vector<std::vector<bool>> &grid_out, std::
   mutex *mtx)
{
    int width_out = grid_out[0].size();
    int height_out = grid_out.size();
    mtx->lock();
    int number_of_neighbors = n_neighbors(grid, x, y);
    switch (number_of_neighbors)
    {
        case 3:
            grid_out[(y - 1) % height_out][(x - 1) %
   width_out] = true;
            break;
        case 2:
            grid_out[(y - 1) % height_out][(x - 1) %
   width_out] = grid[y][x];
            break;
        default:
            grid_out[(y - 1) % height_out][(x - 1) %
   width_out] = false;
            break;
    }
```

```
    mtx->unlock();
}
```

Ensuite, puisque l'on va traiter la grille par zone via chaque thread, il y a la fonction **process_partial_grid** qui va appliquer la fonction précédente à chaque zone de la grille définie par les coin supérieur gauche et inférieur droit.

```
std::vector<std::vector<bool>> process_partial_grid(int x1,
   int x2, int y1, int y2, std::vector<std::vector<bool>>
   grid, std::mutex *mtx)
{
    std::vector<std::vector<bool>> grid_out((y2 - y1), std::
   vector<bool>(x2 - x1));

    for (int x = x1; x < x2; x++)
        for (int y = y1; y < y2; y++)
            process_cell(x, y, grid, grid_out, mtx);

    return grid_out;
}
```

IV.2.b Affichage de la grille

Nous avons utilisé OpenCV pour l'affichage de la grille du jeu de la vie et sa mise à jour à chaque étape. C'est donc une succession d'images qui permet de représenter chaque étape du jeu de la vie.

```
void show_state_img(int generation, std::vector<std::vector<
   bool>> grid)
{
    cv::Mat img(grid.size(), grid[0].size(), CV_8UC3, cv::
   Scalar(200,200, 200));

    for(int y=0; y < img.rows; y++)
    {
        for(int x=0; x < img.cols; x++)
        {
            if (grid[y][x])
            {
                // get pixel
                cv::Vec3b & color = img.at<cv::Vec3b>(y,x);
                color[1] = 0;
                color[2] = 0;
                // set pixel
                img.at<cv::Vec3b>(y,x) = color;
            }
        }
    }
```

```
20
21      cv::imshow("Game of life", img);
22      cv::resizeWindow("Game of life", 800, 400);
23      cv::waitKey(1);
24  }
```

IV.2.c Fonction principale

Enfin, il y a la fonction principale, rassemblant le multi-threading et l'affichage. Il faut également générer une grille de départ. Nous avons décidé de le faire de manière aléatoire, avec une certaine proportion de cellule en vie au départ. Nous pouvons également choisir la taille de la grille dans la fonction **generate_initial_grid**.

```
1  std::vector<std::vector<bool>> generate_initial_grid(int
       width, int height, double proportion)
2  {
3      std::vector<std::vector<bool>> grid_out;
4      std::vector<bool> border_line;
5      for (int i = -1; i < width + 1; i++)
6          border_line.push_back(false);
7      grid_out.push_back(border_line);
8      for (int i = 0; i < height; i ++)
9      {
10         std::vector<bool> line;
11         line.push_back(false);
12         for (int j = 0; j < width; j ++)
13         {
14             line.push_back( (double) std::rand() / RAND_MAX <
       proportion);
15         }
16         line.push_back(false);
17         grid_out.push_back(line);
18     }
19     grid_out.push_back(border_line);
20     return grid_out;
21 }
```

La fonction principale **game_of_life_multi_thread** crée donc une grille de la taille demandée, par commodité nous forçons le fait que la largeur soit supérieur à la hauteur, et que le nombre de thread demandé soit une puissance de deux.

On crée ensuite les threads, et l'on sépare la grille en sous-grilles de tailles égales, où chacune sera gérée par un thread différent. Il suffit ensuite de les recoller ensemble et l'on obtient la grille de l'étape suivante.

```
1  void game_of_life_multi_thread(int nThreads, int grid_width,
       int grid_height, double proportion, int max_iter)
2  {
3      cv::namedWindow("Game of life", cv::WINDOW_NORMAL);
```

```
4     if (grid_width < grid_height)
5         std::swap(grid_width, grid_height);
6     int columns = pow(2,ceil(log2(sqrt(nThreads))));
7     int lines = round(pow(2,floor(log2(nThreads))) / columns)
    ;
8
9     std::vector<std::vector<bool>> grid =
    generate_initial_grid(grid_width, grid_height, proportion)
    ;
10
11    show_state_img(0,grid);
12    //show_state_print(0,grid);
13    cv::resizeWindow("Game of life", 800, 400);
14
15    std::vector<std::future<std::vector<std::vector<bool>>>>
    threads(columns * lines);
16
17    std::vector<std::vector<std::vector<bool>>> results(
    columns * lines);
18
19    for (int iter = 0; iter < max_iter; iter++)
20    {
21        //std::vector<std::vector< std::vector<std::vector<
    bool>> >> partial_grids;
22        for (int column = 0; column < columns; column++)
23        {
24            for (int line = 0; line < lines; line++)
25            {
26                int x1 = grid_width / columns * column + 1;
27                int x2 = grid_width / columns * (column + 1)
    + 1;
28                int y1 = grid_height / lines * line + 1;
29                int y2 = grid_height / lines * (line + 1) +
    1;
30                threads[column * lines + line] = std::async(
    std::launch::deferred,process_partial_grid,x1, x2, y1, y2,
     grid, &mtx);
31                //results[column * lines + line] =
    process_partial_grid(x1, x2, y1, y2, grid, &mtx);
32            }
33        }
34        for (int column = 0; column < columns; column++)
35        {
36            for (int line = 0; line < lines; line++)
37            {
```

```
38                  int x1 = grid_width / columns * column + 1;
39                  int x2 = grid_width / columns * (column + 1)
    + 1;
40                  int y1 = grid_height / lines * line + 1;
41                  int y2 = grid_height / lines * (line + 1) +
    1;
42
43                  int i = column * lines + line;
44
45                  std::vector<std::vector<bool>> partial_grid =
     threads[i].get();
46                  //std::vector<std::vector<bool>> partial_grid
     = results[i];
47
48                  for (int x = x1; x < x2; x++)
49                  {
50                      for (int y = y1; y < y2; y++)
51                      {
52                          grid[y][x] = partial_grid[y - y1][x -
     x1];
53                      }
54                  }
55              }
56          }
57          //usleep(1000); // Sleep 1000 milliseconds
58          //show_state_print(iter + 1, grid);
59          show_state_img(iter + 1,grid);
60          /*int c = 0;
61          std::cout << "Next generation ? (Press 1)" << std::
    endl;
62          std::cin >> c;
63          if (c != 1)
64              break;*/
65          std::cout << "Generation " << iter + 1 << " done." <<
     std::endl;
66      }
67      cv::destroyWindow("A_good_name");
68  }
```

IV.3 Résultats

Le résultat obtenu est une animation montrant en bleu les cellules vivantes et en blanc les cellules mortes.

L'initialisation est aléatoire mais on peut tout à fait imaginer une initialisation à partir

d'une matrice connue (l'import de cette matrice est à coder).

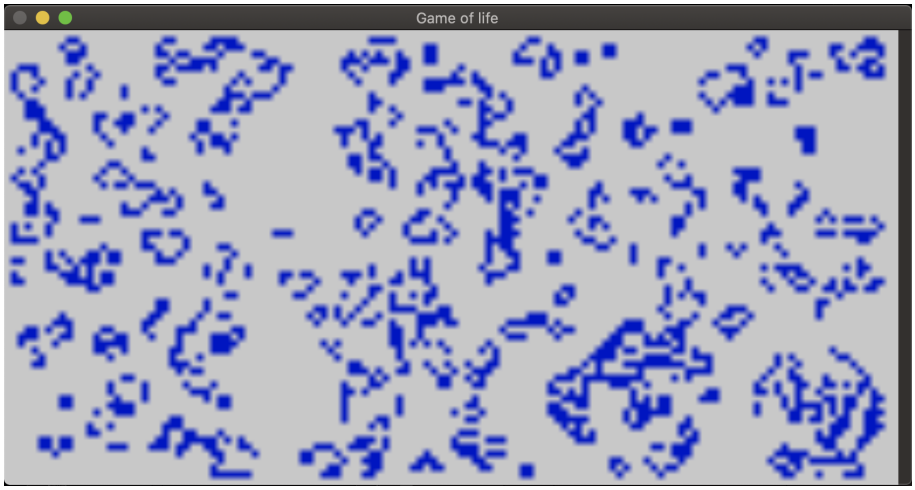

FIGURE 6 – Affichage d'une des premières génération du jeu de la vie avec initialisation aléatoire

On remarque qu'à partir d'un nombre d'itération assez court (< 1000), des formes très caractéristiques et connues du jeu de la vie se forment. Ce sont des formes stables qui soit n'évoluent plus, soit alternent entre une forme et une autre de manière stable (les oscillateurs) (cf Figure 7).

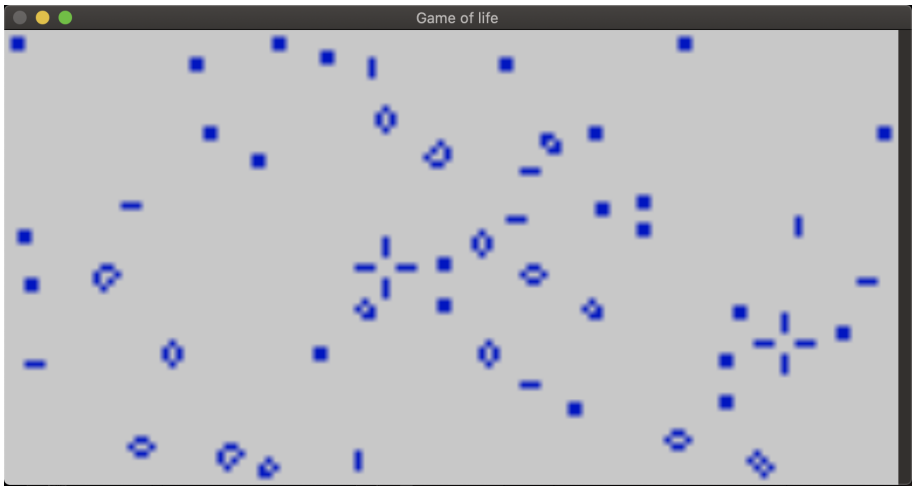

FIGURE 7 – Points fixes du jeu de la vie

La parallélisation en 4 threads permet de calculer et afficher une génération de la grille de 128x64 en 0,4s.

Conclusion

Dans cette application pratique du cours de *Système temps réel, embarqué et mobile*, nous avons pu implémenter différents algorithmes de programmation concurrente permettant de booster les performances de nos algorithmes.

Nous avons vu comment utiliser les threads en **C++**. Cette optimisation permet de réduire le temps de calcul tout en n'utilisant qu'un seul coeur.

D'autres optimisation existent, notamment celles utilisant différents processeurs de l'ordinateur pour effectuer certaines tâches en parallèle.

Ce cours nous a aussi permis de nous familiariser avec la programmation en **C++**.