

MPI_Rapport

Problème de la vérification d'une grille sudoku

Prénom : Aymerick

Nom : LAURETTA-PERONNE

Objectif : Proposez trois versions parallèles du problème de résolution d'une grille de sudoku à l'aide des librairies `MPI` , `OpenCL` et `OpenMP` .

Résultat : Pour ce problème, 2 des 3 librairies sont utiliser : `MPI` et `OpenMP` .

[Lien GitHub](#) | [Lien vers le PDF](#) | [Version Web](#)

Projet de fin d'année d'étude de 3ème année de *Licence Informatique* en Information Distribué et Parallèle. Pour ce projet, j'ai décider de choisir le Problème de résolution d'une grille de sudoku . Le sudoku est un puzzle de placement de nombres combinatoire basé sur la logique qui est populaire depuis la fin du 20e siècle. La résolution de grilles $n^2 * n^2$ de $n * n$ blocs tend à devenir de plus en plus difficile en raison de l'explosion combinatoire. Par conséquent, une implémentation séquentielle d'un résolveur de puzzle Sudoku peut devenir à la fois gourmande en données et en calcul. Dans ce travail, nous construisons des implémentations parallèles de l'algorithme de résolution d'énigmes Sudoku en utilisant la propagation de contraintes, en stockant des grilles déjà explorées et en stockant des grilles à explorer. Les économies de temps d'exécution résultent des différentes méthodes de gestion des threads et des méthodes de partage des données. Les threads tentent de résoudre le puzzle en utilisant la recherche en profondeur d'abord.

Nom du processeur : Intel Core i5 double cœur; Vitesse du processeur : 2,3 GHz; Nombre de processeurs : 1; Nombre total de cœurs : 2; Technologie Hyper-Threading : Activé; Mémoire : 8 Go;

Nom du processeur : Intel Core i5-8300H; Vitesse du processeur : 2,3 GHz; Nombre de processeurs : 1; Nombre total de cœurs : 4; Processeurs logiques : 8; Technologie Hyper-Threading : Activé; Mémoire : 16 Go;

Présentation des algorithmes

Définition de notre structure `sudoku` contenant :

- un pointer de pointer de grid nous permettant la création d'un tableau en 2D de type `entiger`

- une `size` nous permettant de définir la taille de notre tableau lors de notre allocation

Structure "sudoku"

```
typedef struct sudoku { // Structure Sudoku
    int **grid; // Grille du Sudoku
    int size; // Taille de la grille de Sudoku
} sudoku;
```

Vérification sudoku

Vérification ligne

On commence par vérifier si la valeur est déjà dans la ligne, pour ce faire nous allons créer une fonction `is_exist_row` prenant en paramètre notre `sudoku *s`, et deux entiers `row` et `value` :

```
int is_exist_row(sudoku *s, int row, int value) {
```

On commence notre vérification :

```
    for (int i = 0; i < s->size; i++) {
```

Si la valeur est déjà présente dans la ligne on retourne -1 afin d'indiquer que la valeur est déjà présente :

```
        if (s->grid[row][i] == value) {
            return 1;
        }
    }
```

Sinon on retourne -0, et si c'est le cas la valeur est pas dans la ligne :

```
    return 0; // Return 0 if the value is not in the row
}
```

Vérification colonne

Pour la vérification de la colonne, c'est relativement le même procédé que pour la vérification de ligne :

```
int is_exist_col(sudoku *s, int col, int value) {
    for (int i = 0; i < s->size; i++) { // Vérifiez si la valeur est déjà dans la
        colonne
            if (s->grid[i][col] == value) { // Si la valeur est déjà dans la colonne
```

```

        return 1; // Renvoie 1 pour indiquer que la valeur est déjà dans la
        colonne
    }
}
return 0; // Renvoie 0 si la valeur n'est pas dans la colonne
}

```

Vérification d'un carrée

La dernière vérification consiste à regarder si nous disposons d'une et une seul fois le même nombre dans un carré.

Pour ce faire on crée comme pour les précédente vérification une fonction `is_exist_square` avec pour paramètre :

- sudoku *s
- col de type integer
- row de type integer
- value de type integer

```

int is_exist_square(sudoku *s, int row, int col, int value) {

```

Square size est la racine carrée de la taille du sudoku :

```

int square_size = sqrt(s->size);

```

Square row est la ligne du carré où se trouve la valeur :

```

int square_row = row - (row % square_size);

```

Square col est la colonne du carré où la valeur est :

```

int square_col = col - (col % square_size);

```

Et la vérification relativement identique également :

```

    /// Vérifie si la valeur est déjà dans the square
    for (int i = square_row; i < square_row + square_size; i++) {
        // Vérifie si la valeur est déjà dans le carré
        for (int j = square_col; j < square_col + square_size; j++) {
            if (s->grid[i][j] == value) { // Si la valeur est déjà dans le carré
                return 1; // Renvoie 1 pour indiquer que la valeur est déjà dans le
                carré
            }
        }
    }
}

```

```
}  
return 0; // Retourne 0 si la valeur n'est pas dans le carré
```

Solver sudoku (extra)

```
int solve_sudoku_rec(sudoku *s, int row, int col) {  
    int square_size = sqrt(s->size);  
    if (row == s->size) { // Si la ligne est la dernière sure  
        return 1;  
    }  
    // Si l'emplacement est déjà attribué, passez au suivant  
    if (s->grid[row][col] != UNASSIGNED) {  
        if (col == s->size - 1) {  
            // Aller à la ligne suivante  
            return solve_sudoku_rec(s, row + 1, 0);  
        } else {  
            // Aller à la colonne suivante  
            return solve_sudoku_rec(s, row, col + 1);  
        }  
    }  
    // Essayer toutes les valeurs possibles  
    for (int i = 1; i <= s->size; i++) {  
        // Si la valeur est sûre, affectez-la à la grille  
        if (is_safe_number(s, row, col, i)) {  
            s->grid[row][col] = i;  
            // Si la colonne est la dernière, passe à la ligne suivante  
            if (col == s->size - 1) {  
                // Si le sudoku est résolu, retourne 1  
                if (solve_sudoku_rec(s, row + 1, 0)) {  
                    // Retourne 1 si le sudoku est résolu  
                    return 1;  
                }  
            } else {  
                // Si le sudoku est résolu, retourne 1  
                if (solve_sudoku_rec(s, row, col + 1)) {  
                    // Retourne 1 si le sudoku est résolu  
                    return 1;  
                }  
            }  
            // Si le sudoku n'est pas résolu, réinitialiser la grille  
            s->grid[row][col] = UNASSIGNED;  
        }  
    }  
    // Retourne 0 si le sudoku n'est pas résolu  
    return 0;  
}
```

Output :

Size: 9
Solving Sudoku:

0	0	0		0	0	0		0	1	0
4	0	0		0	0	0		0	0	0
0	2	0		0	0	0		0	0	0

0	0	0		0	5	0		4	0	7
0	0	8		0	0	0		3	0	0
0	0	1		0	9	0		0	0	0

3	0	0		4	0	0		2	0	0
0	5	0		1	0	0		0	0	0
0	0	0		8	0	6		0	0	0

6	9	3		7	8	4		5	1	2
4	8	7		5	1	2		9	3	6
1	2	5		9	6	3		8	7	4

9	3	2		6	5	1		4	8	7
5	6	8		2	4	7		3	9	1
7	4	1		3	9	8		6	2	5

3	1	9		4	7	5		2	6	8
8	5	6		1	2	9		7	4	3
2	7	4		8	3	6		1	5	9

Solve in 35.668896 s

Output :

Size: 12
Start solving...

	0	3	8	0		0	10	7	0		0	6	4	0	
	4	7	0	11		0	0	0	0		10	0	8	5	
	0	0	0	0		0	0	0	0		0	0	0	0	

	0	1	11	5		3	0	0	12		8	4	2	0	
	0	10	0	0		4	0	0	5		0	0	12	0	
	0	12	0	0		2	0	0	3		0	0	9	0	

	0	2	10	4		6	0	0	8		1	12	3	0	
	0	0	0	0		0	0	0	0		0	0	0	0	
	0	0	3	12		0	0	0	0		4	11	0	0	

	5	8	0	10		0	0	0	0		9	0	1	6	
	0	6	1	0		0	9	5	0		0	10	7	0	
	0	0	0	0		0	0	0	0		0	0	0	0	

	1	3	8	2		5	10	7	9		11	6	4	12	
	4	7	2	11		1	3	12	6		10	9	8	5	
	11	9	6	7		12	8	1	4		5	2	10	3	

	9	1	11	5		3	6	10	12		8	4	2	7	
	3	10	7	9		4	1	6	5		2	8	12	11	

```
| 6 12 5 8 | 2 11 4 3 | 7 1 9 10 |
-----
| 7 2 10 4 | 6 5 11 8 | 1 12 3 9 |
| 12 11 4 1 | 9 2 3 10 | 6 7 5 8 |
| 8 5 3 12 | 10 7 9 2 | 4 11 6 1 |
-----
| 5 8 12 10 | 11 4 2 7 | 9 3 1 6 |
| 2 6 1 3 | 8 9 5 11 | 12 10 7 4 |
| 10 4 9 6 | 7 12 8 1 | 3 5 11 2 |
-----
```

Solve in : 41.339593 s

Le temps nécessaire à la résolution d'un sudoku dépend de sa taille et de sa difficulté (nombre de nombres de départ)

MPI

Adaptation de la structure :

```
typedef struct {
    int tabs[9][9]; // Sudoku struct definition
    int i; // i is the index of the current thread
    int j; // j is the index of the current column
} sudoku;

sudoku *grids;
int pointer;
```

```
long int nsol = 0; // Nombre de solutions
long int sol; // Nombre de solutions
int id, n_procc; // id est l'identifiant du processus courant, n_procc est le
nombre de processus
```

Initialisation de MPI :

```
MPI_Init(NULL, NULL);
```

On récupère l'id du processus :

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
```

On récupère le nombre de processus :

```
MPI_Comm_size(MPI_COMM_WORLD, &n_procc);
```

Autres initialisation :

```
    grids = (sudoku *)malloc(sizeof(sudoku) * TABSIZE); // On alloue la mémoire pour
les grilles
    pointer = 0; // On initialise le pointeur de grilles à 0
    init_grids(0, 0, 0); // On initialise les grilles
    int num_elements_per_proc = pointer / n_procc; // On calcule le nombre d'éléments
par processus
    int id_limit = pointer % n_procc; // On calcule le nombre d'éléments restants
    int i = 0; // On initialise le compteur de grilles à 0
    int j = 0; // On initialise le compteur de processus à 0
```

Si on est pas dans le bon processus :

```
while (i < id) { // Tant que l'on est pas sur le bon processus
    if (i < id_limit) // Si il reste des éléments à envoyer
        j += num_elements_per_proc + 1; // On ajoute un élément
    else
        j += num_elements_per_proc; // Sinon on ajoute pas d'élément
    i++; // On passe au prochain processus
}
```

On envoie les grilles aux autres processus :

```
if (id < id_limit) // Si il reste des éléments à envoyer
    // On ajoute un élément
    pointer = j + num_elements_per_proc + 1;
else
    // Sinon on ajoute pas d'élément
    pointer = j + num_elements_per_proc;
// Tant qu'il reste des grilles à envoyer
for (; j < pointer; j++) {
    // On teste les grilles
    nsol += check(grids[j].i, grids[j].j, grids[j].tabs);
}
```

```
// On affiche le nombre de solutions trouvées
printf("node %d found %ld solutions\n", id, nsol);

// On crée une opération pour additionner les résultats
MPI_Op op;

// On crée l'opération pour additionner les résultats
MPI_Op_create((MPI_User_function *)addem, 1, &op);

// On réduit les résultats pour obtenir le total
MPI_Reduce(&nsol, &sol, 1, MPI_LONG_INT, op, 0, MPI_COMM_WORLD);

if (id == 0) {
```

```

printf("For tab Size %d \n", TABSIZE);
printf("Number of solutions : %ld\n", sol);
}

// End timer MPI
double end_time = MPI_Wtime();
double total_time = (end_time - start_time);
printf("node %d total time: %f\n", id, total_time);

// On attend que tous les processus aient fini
MPI_Barrier(MPI_COMM_WORLD);

// On termine MPI
MPI_Finalize();

// On quitte le programme
exit(0);

```

Résultat MPI

![clipboard.png](inkdrop://file:HskwfsAx4)

OpenMP

Vérifier que l'élément z peut être placé en position (x, y) du sudoku :

```

int can_be_assigned(int x, int y, int z, int thread) {
    int i, j, pi, pj;

    // Vérifiez la ligne de la position (x, y)
    for (i = 0; i < 9; i++) {

        // Si l'élément est déjà dans la ligne
        if (tabs[thread][x][i] == z)

            // L'élément ne peut pas être placé en position (x, y)
            return (FALS);

        // Si l'élément est déjà dans la colonne
        if (tabs[thread][i][y] == z)

            // L'élément ne peut pas être placé en position (x, y)
            return (FALS);
    }

    // Vérifier le carré
    pi = (x / 3) * 3; // La première ligne du carré
    pj = y - y % 3;   // The first column of the square

    // Vérifie le carré de la position (x, y)
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            // Si l'élément est déjà dans le carré

```



```

        if (tabs[thread][pi + i][pj + j] == z)
            return (FALS); // L'élément ne peut pas être placé en position (x, y)
        return (CERT); // L'élément peut être placé en position (x, y)
    }

```

On vérifie si l'élément peut être placé en position (i, j) :

```

int check(int i, int j, int thread) {
    int k;
    long int s = 0;

    if (tabs[thread][i][j]) // La valeur fixe ne doit pas être itérée
    {
        if (j < 8) // Vérifie la colonne suivante du tableau
            return (check(i, j + 1, thread)); // Appel récursif
        else if (i < 8)
            return (check(i + 1, 0, thread)); // Appel récursif
        else
            return (1); // Fin du tab
    } else // Il y a un 0 que nous devons essayer
    {
        for (k = 1; k < 10; k++) // // Essaye tous les nombres

            // S'il est possible de mettre k dans (i, j)
            if (can_be_assigned(i, j, k, thread)) {
                tabs[thread][i][j] = k; // Met k dans (i, j)
                if (j < 8) // Check the next column of the table

                    // Appel récursif pour vérifier la colonne suivante du tableau
                (j+1)
                    s += check(i, j + 1, thread);
                else if (i < 8) // Vérifie la ligne suivante du tableau
                    s += check(i + 1, 0, thread); // Appel récursif (i+1, 0)
                else // End of tab
                    s++; // Nous avons trouvé une solution
                tabs[thread][i][j] = 0; // Retire k de (i, j)
            }
    }
    return (s); // Renvoie le nombre de solutions
}

```

On vérifie si le sudoku est résolu :

```

int is_solve(int i, int j) {
    int k; // Compteur pour la boucle sur les nombres
    long int s = 0; // Compteur du nombre de solutions
    int thread = 0; // Le numéro de thread

    if (tabs[thread][i][j]) { // La valeur fixe ne doit pas être itérée
        if (j < 8) // Vérifier la colonne suivante
            return (firstcheck(i, j + 1));
        else if (i < 8) // Vérifie la ligne suivante
            return (firstcheck(i + 1, 0));
    }
}

```

```

        else
            return (1); //
    } else {
#pragma omp parallel firstprivate(thread)
    {
#pragma omp for reduction(+ \
                        : s)
        for (k = 1; k < 10; k++) { // Il y a un 0 que nous devons essayer
            clock_t start = clock();
            thread = omp_get_thread_num(); // Récupère le numéro du thread
            if (can_be_assigned(i, j, k, thread)) { // Vérifie si le numéro peut
être attribué

                tabs[thread][i][j] = k; // Affecte le numéro
                if (j < 8) // Vérifiez la colonne suivante
                    s += check(i, j + 1, thread); // Aller à la colonne suivante
                else if (i < 8) // Vérifiez la ligne suivante
                    s += check(i + 1, 0, thread); // Aller à la ligne suivante
                else
                    s++; // Incrémentation du compteur de solution
                tabs[thread][i][j] = 0; // Réinitialise la valeur de la cellule
                printf("%d Results using thread %d: %li\n", k, thread, s);
                clock_t end = clock();
                double time_spent = (double)(end - start) / CLOCKS_PER_SEC;
                printf("Time spent: %f\n", time_spent);
            }
        }
    }
}

return (s); // Retourne le nombre de solutions
}

```

![clipboard.png](inkdrop://file:uTvbQg-cN)

à revoir