**PROJECT REPORT**

**Parallel Sudoku**

Group Members:

* Ashmal anis

Objective:

Parallel programming on an SMP and a multicomputer, using OpenMP for Parallel and Distributed Computing course. For this purpose, it was developed a serial and parallel implementation of a sudoku solver.

Methodology:

The puzzle consists of a matrix of size n x n which is partially filled and the algorithm fills the matrix cells which are blank with values from 1 to n in such a way that no value is repeated more than once on each of n columns and n rows or n squares of size√n × √n on which the matrix is split.

The algorithm we are using can be summarized as follows:

1. **Apply heuristics** on the input grid. This gives the initial grid for creation of allotment list.
2. **Create a list of grids for allotment** among the threads by doing BFS level-by-level on a tree of intermediate grids that has the initial grid at the root. This is done till there are less than thread count many grids in the list.
3. **Each thread** gets a grid from the allotment list and uses a **local stack** to **execute brute force DFS** on it. It then **repeats** the following till either the solution is found or the stack is empty in which case it gets the next grid.
   1. Pop a grid from the search stack.
   2. Apply heuristics on it.
   3. Expand the tree by selecting the cell with least number and pushing the newly created grids into the stack.
   4. Whenever the solution is found the thread sets **a shared variable** indicating this and all the threads exit the parallel section.
4. After the parallel section, the value of the shared variable is checked to figure out if the solution was found.

The design decisions that we took in our implementations include:

1. **Possible values** for each cell in the grid store as a **bit mask of length 64**.
   * **O(1) addition, deletion and searching etc.** of possible values and hence is much faster than using an array.
   * Other operations such as **getting number of possible values etc.** also done in O(1) time using gcc’s builtin functions.
2. The application of heuristics is done as follows till none of the heuristics make any change:
3. Sequence in which the heuristics are applied: **Elimination -> Loneranger -> Twins**.
4. If the application of a heuristics causes some change, then the sequence is repeated from the beginning. This ensures that elimination and lone rangers being more useful are applied more frequently.
5. **Static allocation** of workload among the threads as dynamic allocation didn’t seem to be useful.

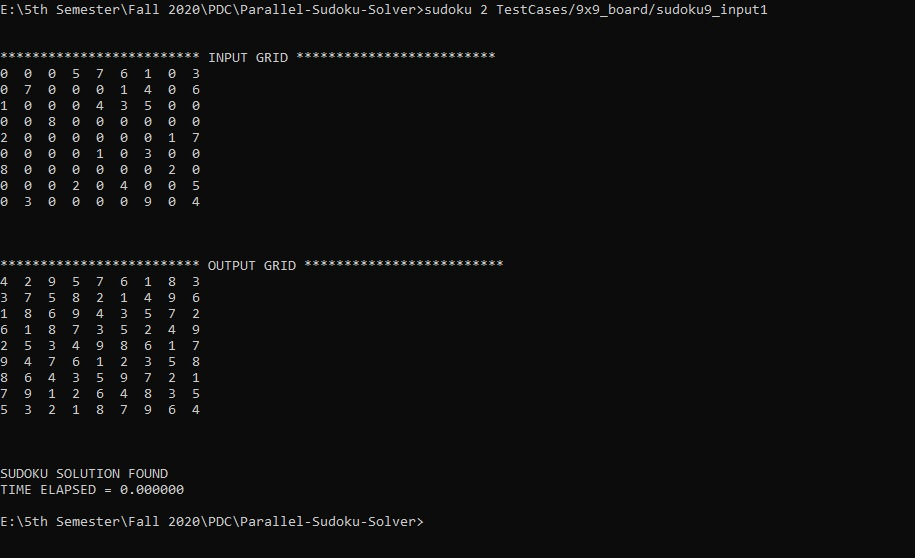
a. Grids assigned in **round-robin fashion** for similar workloads on all threads.

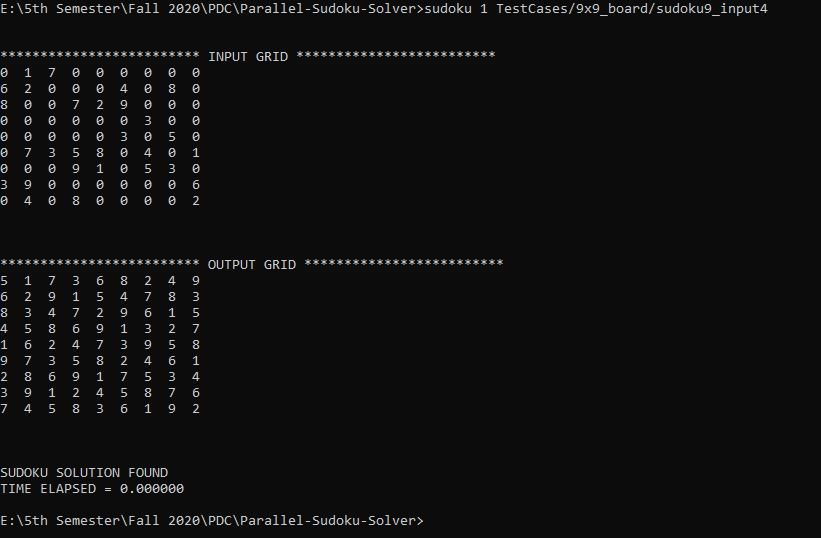
1. **Prune the DFS** tree i.e., ignore the branch whenever:
2. A cell has no possible values
3. A number doesn’t occur in the possible values of any cell in a row/column/box.
4. Stacks store any grid that gets freed so that a new grid doesn’t need to be allocated from scratch.
5. **Parallelized only the DFS section**, as each application of the heuristics doesn't take much time and so the overheads of parallelizing them would have been too much.
6. Triplets not used as it isn’t efficient enough.

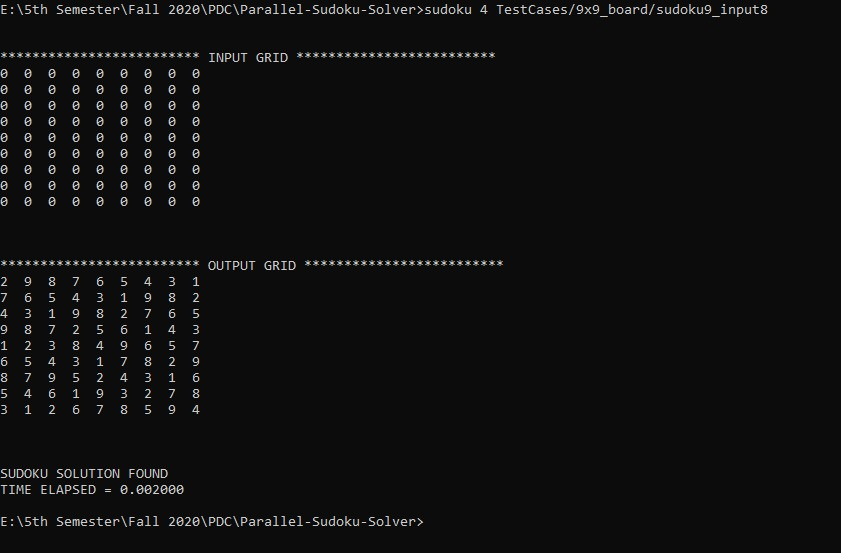
Input and Output:

There are 2 types of grids 9x9 and 16x16. The digit highlighted in the image below after sudoku shows the number of threads. The 0 in the input grid shows blank space.





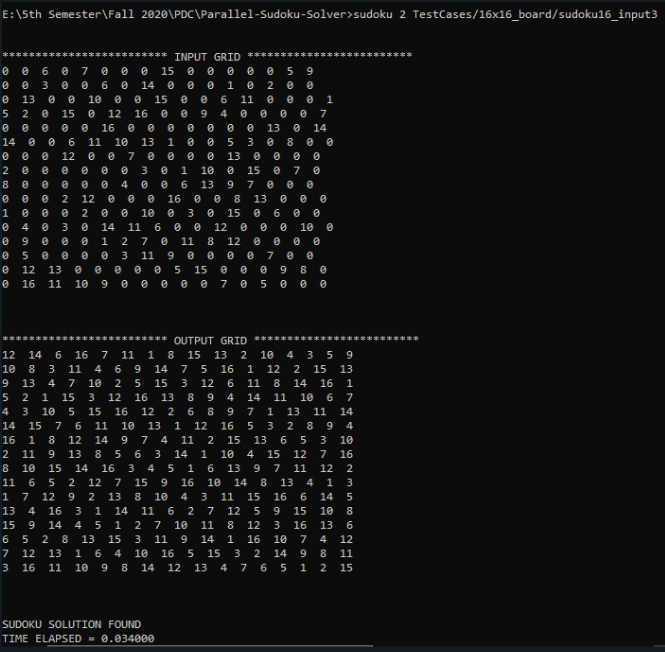




Comparison w.r.t threads and time

By increasing number of threads, the time taken is reduced which can be seen in the following images.

No. of Threads = 2



No. of Threads = 4

Code:

Main.c

#include <stdio.h>

#include <stdlib.h>

#include "sudoku.h"

#include <string.h>

#include <omp.h>

// Number of threads initialised.

int threadCount = 4;

//Returns a 2D array from an input file containing the Sudoku in space separated format (empty cells are 0)

int \*\* readInput(char \*filename){

FILE \*inputFile;

inputFile = fopen(filename, "r");

int i, j;

char dummyline[SIZE+1];

char dummy;

int val;

int \*\*sudokuGrid = (int\*\*)malloc(sizeof(int\*)\*SIZE);

// Initialises the dynamic memory with 0's.

for (i = 0; i < SIZE; i++){

sudokuGrid[i] = (int\*)malloc(sizeof(int)\*SIZE);

for (j = 0; j < SIZE; j++)

sudokuGrid[i][j] = 0;

}

// Initialises dynamically with the input values of the grid.

for (i = 0; i < SIZE; i++){

for (j = 0; j < SIZE; j++){

// Checking if number of rows is less than SIZE

if (feof(inputFile)){

if (i != SIZE){

printf("The input puzzle has less number of rows than %d. Exiting.\n", SIZE);

exit(-1);

}

}

fscanf(inputFile, "%d", &val);

if(val >= 0 && val <= SIZE)

sudokuGrid[i][j] = val;

else{

printf("The input puzzle is not a grid of numbers (0 <= n <= %d) of size %dx%d. Exiting.\n", SIZE, SIZE, SIZE);

exit(-1);

}

}

// To remove stray \0 at the end of each line.

fscanf(inputFile, "%c", &dummy);

// Checking if row has more elements than SIZE.

if (j > SIZE){

printf("Row %d has more number of elements than %d. Exiting.\n", i+1, SIZE);

exit(-1);

}

}

return sudokuGrid;

}

/\*Checks if solution is a valid solution to original one

i.e. all originally filled cells match, and that solution is a valid grid\*/

int checkValid(int \*\*original, int \*\*solution){

int valuesChecked[SIZE],i,j,k;

//check all rows

for (i = 0; i < SIZE; i++){

for (k = 0; k < SIZE; k++) valuesChecked[k] = 0;

for (j = 0; j < SIZE; j++){

if (solution[i][j]==0) return 0;

if ((original[i][j])&&(solution[i][j] != original[i][j])) {

printf("Values mismatch at (%d, %d) \n", i, j);

return 0;

}

int v = solution[i][j];

if (valuesChecked[v-1]==1){

printf("Repeatition of values %d in row at (%d, %d) \n", v, i, j);

return 0;

}

valuesChecked[v-1] = 1;

}

}

//check all columns

for (i = 0; i < SIZE; i++){

for (k = 0; k < SIZE; k++) valuesChecked[k] = 0;

for (j = 0; j < SIZE; j++){

int v = solution[j][i];

if (valuesChecked[v-1]==1){

printf("Repeatition of values %d in column at (%d, %d) \n", v, j, i);

return 0;

}

valuesChecked[v-1] = 1;

}

}

//check all minigrids

//check all rows

for (i = 0; i < SIZE; i = i+MINIGRIDSIZE){

for (j = 0; j < SIZE; j = j+MINIGRIDSIZE){

for (k = 0; k < SIZE; k++) valuesChecked[k] = 0;

int r,c;

for (r = i; r < i+MINIGRIDSIZE; r++)

for (c = j; c < j+MINIGRIDSIZE; c++){

int v = solution[r][c];

if (valuesChecked[v-1]==1) {

printf("repeat of val %d in box at (%d, %d) \n", v, i, j);

return 0;

}

valuesChecked[v-1] = 1;

}

}

}

return 1;

}

int main(int argc, char \*argv[]){

if (argc<3){

// Catches erroneous input

printf("Usage: ./sudoku <threadCount> <inputFile>\n");

exit(0);

}

int \*\*originalGrid = readInput(argv[2]);

int \*\*gridToSolve = readInput(argv[2]);

threadCount = atoi(argv[1]);

if (threadCount<=0){

printf("Usage: Thread Count should be positive\n");

}

omp\_set\_num\_threads(threadCount);

int i,j;

printf("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* INPUT GRID \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

for (i=0;i<SIZE;i++){

for (j=0;j<SIZE;j++){

printf("%d ",originalGrid[i][j]);

}

printf("\n");

}

printf("\n\n\n");

double start = omp\_get\_wtime();

int \*\*outputGrid = solSudoku(originalGrid);

double finish = omp\_get\_wtime();

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* OUTPUT GRID \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

for (i=0;i<SIZE;i++){

for (j=0;j<SIZE;j++)

printf("%d ",outputGrid[i][j]);

printf("\n");

}

printf("\n\n\n");

if (checkValid(originalGrid,outputGrid)){

printf("SUDOKU SOLUTION FOUND\nTIME ELAPSED = %lf\n",(finish-start));

}

else{

printf("NO SOLUTION OF SUDOKU FOUND\nTIME ELAPSED =%lf\n",(finish-start));

}

}

Sudoku.c

// Libraries defined.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <omp.h>

#include <limits.h>

#include <stdint.h>

#include <math.h>

#include <assert.h>

#include "sudoku.h"

#define true 1

#define false 0

typedef struct {

char\* matrix;

uint64\_t\* mask;

int i;

} gridStruct;

typedef gridStruct\* grid\_t;

#define CurrValue(grid) (grid->matrix[grid->i])

#define idx(x, y) ( (x) \* SIZE + (y) )

#define valCount(x) ( \_\_builtin\_popcountll(x) )

#define valSmallest(x) ( \_\_builtin\_ctzll(x) + 1 )

typedef struct {

grid\_t\* list;

int list\_sz;

int list\_alloc;

} stackStruct;

typedef stackStruct\* stack\_t;

typedef grid\_t stackelem\_t;

const int INIT\_STACK\_SIZE = 128;

const int M = SIZE, N = MINIGRIDSIZE;

extern int threadCount;

int solFound = 0;

char\* solMatrix;

stack\_t avail\_stack;

stack\_t allotment\_stack;

// Grid Methods

void printMatrix(int \*\*matrix);

int checkValidGrid(grid\_t grid);

void moveToNextUnsetCell(grid\_t grid);

int getLeastUnsureCell(grid\_t grid);

void Copy\_grid(grid\_t grid1, grid\_t grid2);

void Populate\_grid(grid\_t grid, int \*\*matrix);

grid\_t Alloc\_grid(stack\_t avail);

void Free\_grid(grid\_t grid, stack\_t avail);

// Stack Methods

stack\_t Alloc\_stack(void);

void Free\_stack(stack\_t stack);

int Empty\_stack(stack\_t stack);

void Push(stack\_t stack, grid\_t grid);

void Push\_copy(stack\_t stack, grid\_t grid, stack\_t avail);

grid\_t Pop(stack\_t stack);

// Humanistic approach methods

void printPossibleValues(grid\_t grid, int i);

int eliminate(grid\_t grid);

void setCellPossibleValues(grid\_t grid, int i);

int setLoneRangersRow(grid\_t grid);

int setLoneRangersColumn(grid\_t grid);

int setLoneRangersBox(grid\_t grid);

int setTwinsRow(grid\_t grid);

int setTwinsColumn(grid\_t grid);

int setTwinsBox(grid\_t grid);

void updatePeers(grid\_t grid, int i);

// Top level methods

int solveLogical(grid\_t grid);

int Prepare\_allotment\_stack(grid\_t grid);

int \*\*solSudoku(int \*\*);

/\*-----------------------------------------------------------------------------------------------------------------------\*/

void printMatrix(int \*\*matrix) {

int i, j;

for (i = 0; i < SIZE; i++) {

for (j = 0; j < SIZE; j++) {

printf("%2d ", matrix[i][j]);

}

printf("\n");

}

printf("\n");

}

void printGrid(grid\_t grid) {

int j;

for(j = 0; j < SIZE \* SIZE; j++) {

if(j != grid->i){

printf("%2d ", grid->matrix[j]);

} else {

printf("[%d]", grid->matrix[j]);

}

if(j % SIZE == SIZE-1) {

printf("\n");

}

}

printf("\n");

}

void printStack(stack\_t stack) {

printf("STACK: size = %d\n", stack->list\_sz);

int i;

for(i = 0; i<stack->list\_sz; i++)

printGrid(stack->list[i]);

}

int checkValidGrid(grid\_t grid) {

int i = grid->i;

int x = i / M, y = i % M;

int v = grid->matrix[i];

int bx = (x / N) \* N, by = (y / N) \* N;

int j, ox, oy;

for (j = 0; j < SIZE; j++) {

ox = j / N;

oy = j % N;

// check row

if (y != j && v == grid->matrix[x \* SIZE + j]) return false;

// check column

if (x != j && v == grid->matrix[j \* SIZE + y]) return false;

// check box

if (i != ((bx + ox) \* SIZE + by + oy)

&& v == grid->matrix[(bx + ox) \* SIZE + by + oy])

return false;

}

return true;

}

void moveToNextUnsetCell(grid\_t grid){

while(grid->i < SIZE \* SIZE && CurrValue(grid)) {

grid->i++;

}

}

int getLeastUnsureCell(grid\_t grid) {

int idx = 0, min = SIZE;

int i;

for(i = 0; i < SIZE \* SIZE; i++) {

if(grid->matrix[i] == 0 && valCount(grid->mask[i]) < min) {

idx = i;

min = valCount(grid->mask[idx]);

// printf("min = %d\n", min);

}

}

return idx;

}

void Copy\_grid(grid\_t grid1, grid\_t grid2) {

memcpy(grid2->matrix, grid1->matrix, SIZE\*SIZE\*sizeof(char));

memcpy(grid2->mask, grid1->mask, SIZE\*SIZE\*sizeof(uint64\_t));

grid2->i = grid1->i;

}

void Populate\_grid(grid\_t grid, int \*\*matrix) {

int i;

for(i = 0; i< SIZE\*SIZE; i++) {

grid->matrix[i] = matrix[i/SIZE][i%SIZE];

if(grid->matrix[i] > 0) {

grid->mask[i] = (1 << (grid->matrix[i] - 1));

} else {

// all values from 1 to SIZE possible for now.

grid->mask[i] = SIZE == 64 ? ULLONG\_MAX : ((1 << SIZE) - 1);

}

}

}

grid\_t Alloc\_grid(stack\_t avail) {

grid\_t tmp;

if (avail == NULL || Empty\_stack(avail)) {

tmp = (grid\_t) malloc(sizeof(gridStruct));

tmp->matrix = (char \*)malloc(SIZE\*SIZE\*sizeof(char));

tmp->mask = (uint64\_t \*)malloc(SIZE\*SIZE\*sizeof(uint64\_t));

return tmp;

} else {

return Pop(avail);

}

}

void Free\_grid(grid\_t grid, stack\_t avail) {

if(grid != NULL) {

if (avail == NULL) {

free(grid->matrix);

free(grid->mask);

free(grid);

} else {

// push the grid in the avail stack only if it wouldn't cause

// reallocation of the stack

Push(avail, grid);

}

}

}

/\* Stack Methods \*/

stack\_t Alloc\_stack() {

stack\_t tmp;

tmp = (stack\_t)malloc(sizeof(stackStruct));

tmp->list = (grid\_t \*)malloc((INIT\_STACK\_SIZE) \* sizeof(grid\_t));

tmp->list\_sz = 0;

tmp->list\_alloc = INIT\_STACK\_SIZE;

return tmp;

}

void Free\_stack(stack\_t stack) {

if(stack != NULL) {

if(!Empty\_stack(stack)) {

int i = 0;

for(; i<stack->list\_sz; i++) {

Free\_grid(stack->list[i], NULL);

}

}

free(stack->list);

free(stack);

}

}

int Empty\_stack(stack\_t stack) {

return stack == NULL || !stack->list\_sz;

}

void Push(stack\_t stack, grid\_t grid) {

if(stack->list\_sz == stack->list\_alloc) {

grid\_t \*tmp = (grid\_t \*)malloc((stack->list\_alloc) \* 2 \* sizeof(grid\_t));

memcpy(tmp, stack->list, (stack->list\_sz)\*sizeof(grid\_t));

free(stack->list);

stack->list = tmp;

stack->list\_alloc \*= 2;

// printf("stack->list\_alloc = %d\n", stack->list\_alloc);

}

stack->list[stack->list\_sz++] = grid;

}

void Push\_copy(stack\_t stack, grid\_t grid, stack\_t avail) {

grid\_t tmp = Alloc\_grid(avail);

Copy\_grid(grid, tmp);

Push(stack, tmp);

}

grid\_t Pop(stack\_t stack) {

if(Empty\_stack(stack))

return NULL;

stack->list\_sz--;

return stack->list[stack->list\_sz];

}

/\* Stack Methods \*/

int Prepare\_allotment\_stack(grid\_t grid) {

grid\_t curr\_grid = Alloc\_grid(avail\_stack);

// // Temporary stacks for work assignment

stack\_t stack1 = Alloc\_stack(), stack2 = Alloc\_stack();

stack\_t allot\_stack1, allot\_stack2;

grid->i = 0;

Push\_copy(stack1, grid, avail\_stack);

int j = 0, idx, v;

uint64\_t vals;

do {

if(j % 2) {

allot\_stack1 = stack2;

allot\_stack2 = stack1;

} else {

allot\_stack1 = stack1;

allot\_stack2 = stack2;

}

// pop from allot\_stack1, expand next level, push into allot\_atack2

while(!Empty\_stack(allot\_stack1)) {

curr\_grid = Pop(allot\_stack1);

if(0) printGrid(curr\_grid);

curr\_grid->i = 0;

while(curr\_grid->i < SIZE \* SIZE && CurrValue(curr\_grid))

curr\_grid->i++;

if(curr\_grid->i == SIZE \* SIZE) {

// solution has been found

solFound = 1;

solMatrix = curr\_grid->matrix;

return 1;

}

idx = getLeastUnsureCell(curr\_grid);

vals = curr\_grid->mask[idx];

v = 1;

do {

if(vals % 2){

curr\_grid->i = idx;

curr\_grid->matrix[idx] = v;

Copy\_grid(curr\_grid, grid);

updatePeers(grid, idx);

if(checkValidGrid(grid)){

Push\_copy(allot\_stack2, grid, avail\_stack);

}

}

vals /= 2;

v++;

} while(vals > 0);

Free\_grid(curr\_grid, avail\_stack);

}

if(Empty\_stack(allot\_stack2)) {

// no solution possible

return -1;

}

j++;

} while(allot\_stack2->list\_sz < threadCount);

if(grid != NULL) {

Free\_grid(grid, avail\_stack);

}

if(curr\_grid != NULL) {

Free\_grid(curr\_grid, avail\_stack);

}

allotment\_stack = allot\_stack2;

return 0;

}

int solveLogical(grid\_t grid) {

int changes = false;

int i;

do {

// printf("Setting Possible Values\n");

for(i = 0; i < SIZE \* SIZE; i++) {

if(grid->matrix[i] == 0) {

setCellPossibleValues(grid, i);

if(grid->mask[i] == 0) return -1;

}

else

grid->mask[i] = (1 << (grid->matrix[i] - 1));

}

changes = eliminate(grid);

// printGrid(grid);

if(!changes) {

changes = setLoneRangersRow(grid);

if(!changes) {

changes = setLoneRangersColumn(grid);

if(!changes) {

changes = setLoneRangersBox(grid);

if(!changes) {

changes = setTwinsRow(grid);

if(!changes) {

changes = setTwinsColumn(grid);

}

}

}

}

}

// TODO : set solFound

} while(!solFound && changes);

return 0;

}

int \*\*solSudoku(int \*\* original\_matrix) {

avail\_stack = Alloc\_stack();

grid\_t init\_grid = Alloc\_grid(avail\_stack);

Populate\_grid(init\_grid, original\_matrix);

init\_grid->i = 0;

int r = solveLogical(init\_grid);

if(r < 0) {

goto end;

}

r = Prepare\_allotment\_stack(init\_grid);

if(r != 0){

goto end;

}

if(0) printStack(allotment\_stack);

// exit(0);

#pragma omp parallel shared(solFound)

{

int tid, i, nthrds;

int idx, v;

uint64\_t vals;

grid\_t curr\_grid, grid = Alloc\_grid(avail\_stack);

stack\_t avail\_stack\_local = Alloc\_stack();

stack\_t search\_stack\_local = Alloc\_stack();

tid = omp\_get\_thread\_num();

nthrds = omp\_get\_num\_threads();

for(i = tid; i < allotment\_stack->list\_sz && !solFound; i+=nthrds) {

curr\_grid = allotment\_stack->list[i];

do {

if(solveLogical(curr\_grid) < 0){

goto next;

}

curr\_grid->i = 0;

while(curr\_grid->i < SIZE \* SIZE && CurrValue(curr\_grid))

curr\_grid->i++;

if(curr\_grid->i == SIZE\*SIZE) {

#pragma omp critical (soln)

{

solFound = 1;

solMatrix = curr\_grid->matrix;

}

break;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*find the next cell to expand\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

idx = getLeastUnsureCell(curr\_grid);

vals = curr\_grid->mask[idx];

if(0) printPossibleValues(curr\_grid, idx);

v = 1;

do {

if(vals % 2){

curr\_grid->i = idx;

curr\_grid->matrix[idx] = v;

Copy\_grid(curr\_grid, grid);

updatePeers(grid, idx);

if(checkValidGrid(grid)){

Push\_copy(search\_stack\_local, grid, avail\_stack\_local);

}

}

vals /= 2;

v++;

} while(vals > 0);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*find the next cell to expand\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

next:

Free\_grid(curr\_grid, avail\_stack\_local);

curr\_grid = Pop(search\_stack\_local);

} while(curr\_grid != NULL && !solFound);

}

// Free\_stack(search\_stack\_local);

Free\_stack(avail\_stack\_local);

}

// printMatrix(solMatrix);

end: ;

// ^ need to put a ';'as labels can be followed only by statements.

// and declarations are not statements.

int i;

int\*\* ret\_matrix = (int \*\*)malloc(SIZE \* sizeof(int \*));

for(i = 0; i < SIZE; i++) {

ret\_matrix[i] = (int \*)malloc(SIZE \* sizeof(int));

}

// Free\_stack(allotment\_stack);

// Free\_stack(avail\_stack);

if(solFound) {

for(i = 0; i < SIZE \* SIZE; i++) {

ret\_matrix[i/SIZE][i%SIZE] = solMatrix[i];

}

}

else {

ret\_matrix = original\_matrix;

}

return ret\_matrix;

}

int eliminate(grid\_t grid) {

uint64\_t vals;

int changes = false;

int i;

for(i = 0; i < SIZE \* SIZE; i++) {

vals = grid->mask[i];

if(grid->matrix[i] == 0 && valCount(vals) == 1) {

changes = true;

grid->matrix[i] = valSmallest(vals);

updatePeers(grid, i);

if(0) printf("(%d, %d) set to %d\n", i/SIZE, i%SIZE, grid->matrix[i]);

}

}

return changes;

}

// Returns the possible values for current cell of grid.

void setCellPossibleValues(grid\_t grid, int i) {

if(grid->matrix[i] == 0) {

// int i = grid->i;

int x = i / SIZE, y = i % SIZE;

// uint64\_t possValues = grid->mask[i];

// printPossibleValues(grid, i);

int bx = (x / N) \* N, by = (y / N) \* N;

int j, ox, oy, v;

for(j = 0; j < SIZE; j++) {

ox = j / N;

oy = j % N;

v = grid->matrix[idx(x,j)];

// check row

if (y != j && v > 0) {

v = grid->matrix[idx(x,j)];

// printf("row : %d\n", v);

grid->mask[i] &= ~(1 << (v - 1));

// printPossibleValues(grid, i);

}

v = grid->matrix[idx(j,y)];

// check column

if (x != j && v > 0) {

// printf("column : %d\n", v);

grid->mask[i] &= ~(1 << (v - 1));

// printPossibleValues(grid, i);

}

v = grid->matrix[idx(bx + ox, by + oy)];

// check box

if (i != idx(bx + ox, by + oy) && v > 0) {

// printf("box : %d\n", v);

grid->mask[i] &= ~(1 << (v - 1));

// printPossibleValues(grid, i);

}

}

} else {

grid->mask[i] = (1 << (grid->matrix[i] - 1));

}

// return possValues;

}

// set the values of the loners in each the row.

int setLoneRangersRow(grid\_t grid) {

int changes = false, totalChanges = false;

int x, y;

int i, j, v, count;

// loop over all rows

// repeat if change occurred.

do {

changes = false;

//row number

for(i = 0; i < SIZE; i++) {

// value

for(v = 0; v < SIZE; v++) {

count = 0;

// column number

for(j = 0; j < SIZE; j++) {

if(grid->matrix[idx(i, j)] == 0

&& (grid->mask[idx(i, j)] & (1 << v))) {

count++;

if(count > 1)

break;

x = i;

y = j;

}

}

if(count == 1) {

// this value occured in only one cell in row. Set it.

grid->matrix[idx(x, y)] = v + 1;

if(0) printf("(%d, %d) set to %d\n", x, y, grid->matrix[idx(x, y)]);

updatePeers(grid, i);

// no need to updatePeers() as they won't be touched in this call

changes = true;

totalChanges = true;

}

// TODO: if count = 0, then NO POSSIBLE SOLUTION!

}

}

} while(changes);

return totalChanges;

}

int setLoneRangersColumn(grid\_t grid) {

int changes = false, totalChanges = false;

int x, y;

int i, j, v, count;

// loop over all columns

// repeat if change occurred.

do {

changes = false;

// column number

for(j = 0; j < SIZE; j++) {

// value

for(v = 0; v < SIZE; v++) {

count = 0;

// row number

for(i = 0; i < SIZE; i++) {

if(grid->matrix[idx(i, j)] == 0

&& (grid->mask[idx(i, j)] & (1 << v))) {

count++;

if(count > 1)

break;

x = i;

y = j;

}

}

if(count == 1) {

// this value occured in only one cell in column. Set it.

grid->matrix[idx(x, y)] = v + 1;

if(0) printf("(%d, %d) set to %d\n", x, y, grid->matrix[idx(x, y)]);

updatePeers(grid, i);

// no need to updatePeers() as they won't be touched in this call

changes = true;

totalChanges = true;

}

// TODO: if count = 0, then NO POSSIBLE SOLUTION!

}

}

}while(changes);

return totalChanges;

}

int setLoneRangersBox(grid\_t grid) {

int changes = false, totalChanges = false;

int x, y;

int i, j, bi, bj, v, count;

// loop over all boxes

// repeat if change occurred.

do {

changes = false;

// value

for(v = 0; v < SIZE; v++) {

// vertical box index

for(bi = 0; bi < SIZE; bi+=N) {

// horizontal box index

for(bj = 0; bj < SIZE; bj+=N) {

count = 0;

// look for a loner in this box

for(i = bi; i < bi + N; i++) {

for(j = bj; j < bj + N; j++) {

if(grid->matrix[idx(i, j)] == 0

&& (grid->mask[idx(i, j)] & (1 << v))) {

count++;

if(count > 1) {

// value v is not a loner in this box.

break;

}

x = i;

y = j;

}

}

if(count > 1)

break;

}

if(count == 1) {

// v is a loner in this box. Set it.

grid->matrix[idx(x, y)] = v + 1;

if(0) printf("(%d, %d) set to %d\n", x, y, grid->matrix[idx(x, y)]);

updatePeers(grid, i);

// no need to updatePeers() as they won't be touched in this call

changes = true;

totalChanges = true;

}

// TODO: if count = 0, then NO POSSIBLE SOLUTION!

}

}

}

} while(changes);

return totalChanges;

}

int setTwinsRow(grid\_t grid) {

int changes = false;

int i, j, k, l;

uint64\_t vals;

//row number

for(i = 0; i < SIZE; i++) {

// first cell with just two possible values

for(j = 0; j < SIZE; j++) {

vals = grid->mask[idx(i, j)];

if(valCount(vals) == 2) {

// second cell with just two possible values

for(k = j + 1; k < SIZE; k++) {

if(vals == grid->mask[idx(i, k)]) {

// twin found. Remove the pair of possible values from

// all unset cells in row.

for(l = 0; l < SIZE; l++) {

if (l != j && l != k && grid->matrix[idx(i, l)] == 0 &&

(grid->mask[idx(i, l)] & vals)) {

grid->mask[idx(i, l)] &= ~vals;

changes = true;

if(0) {

printPossibleValues(grid, idx(i, l));

}

// leaving the elimination part for eliminate() as that will do

// update peers as well.

}

}

}

}

}

}

}

return changes;

}

int setTwinsColumn(grid\_t grid) {

int changes = false;

int i, j, k, l;

uint64\_t vals;

//column number

for(i = 0; i < SIZE; i++) {

// first cell with just two possible values

for(j = 0; j < SIZE; j++) {

vals = grid->mask[idx(j, i)];

if(valCount(vals) == 2) {

// second cell with just two possible values

for(k = j + 1; k < SIZE; k++) {

if(vals == grid->mask[idx(k, i)]) {

// twin found. Remove the pair of possible values from

// all unset cells in row.

for(l = 0; l < SIZE; l++) {

if (l != j && l != k && grid->matrix[idx(l, i)] == 0 &&

(grid->mask[idx(l, i)] & vals)) {

grid->mask[idx(l, i)] &= ~vals;

changes = true;

if(0) {

printPossibleValues(grid, idx(l, i));

}

// leaving the elimination part for eliminate() as that will do

// update peers as well.

}

}

}

}

}

}

}

return changes;

}

int setTwinsBox(grid\_t grid) {

int changes = false;

int j, k, l, bi, bj;

int idx1, idx2, idx3;

//box number

for(bi = 0; bi < N; bi++) {

for(bj = 0; bj < N; bj++) {

// first cell with just two possible values

for(j = 0; j < SIZE; j++) {

idx1 = idx(bi + j / N, bj + j % N);

if(valCount(grid->mask[idx1]) == 2) {

// second cell with just two possible values

for(k = j + 1; k < SIZE; k++) {

idx2 = idx(bi + k / N, bj + k % N);

if(grid->mask[idx1] == grid->mask[idx2]) {

// twin found. Remove the pair of possible values from

// all unset cells in box.

for(l = 0; l < SIZE; l++) {

if (l != j && l != k) {

idx3 = idx(bi + k / N, bj + k % N);

if(grid->matrix[idx3] == 0 &&

(grid->mask[idx3] & grid->mask[idx1])) {

grid->mask[idx3] &= ~(grid->mask[idx1]);

changes = true;

// leaving the elimination part for eliminate() as that will do

// update peers as well.

}

}

}

}

}

}

}

}

}

return changes;

}

void printPossibleValues(grid\_t grid, int i) {

printf("(%d, %d) has possible values : ", i/SIZE, i%SIZE);

uint64\_t vals = grid->mask[i];

int v = 1;

do {

if(vals % 2)

printf("%2d ", v);

vals /= 2;

v++;

} while(vals > 0);

printf("\n");

}

void updatePeers(grid\_t grid, int i) {

int v = grid->matrix[i];

int x = i / M, y = i % M;

int bx = (x / N) \* N, by = (y / N) \* N;

int j, ox, oy;

for (j = 0; j < SIZE; j++) {

ox = j / N;

oy = j % N;

// check row

if (y != j)

grid->mask[idx(x, j)] &= ~(1 << (v - 1));

if (x != j)

grid->mask[idx(j, y)] &= ~(1 << (v - 1));

// check box

if (i != idx(bx + ox, by + oy))

grid->mask[idx(bx + ox, by + oy)] &= ~(1 << (v - 1));

}

}

Sudoku.h

#ifndef SUDOKU\_H

#define SUDOKU\_H

// Macros for MINIGRIDSIZE

#ifndef MINIGRIDSIZE

#define MINIGRIDSIZE 3

#endif

#define SIZE ((MINIGRIDSIZE)\*(MINIGRIDSIZE))

//Returns a 2D array from an input file containing the Sudoku in space separated format (empty cells are 0)

int \*\*readInput(char \*);

/\*Checks if solution is a valid solution to original one

i.e. all originally filled cells match, and that solution is a valid grid\*/

int checkValid(int \*\*, int \*\*);

// Produces solution to the puzzle.

int \*\*solSudoku(int \*\*);

#endif