Week 4 - Applying OpenMP for Your Project

CED19I026

KOTAMARTHI MOHAN HIMANSHU

Project Description:

Solving a large number of NxN sudoku(1000-1500).We have taken N=9.

Sudoku is a logic-based, combinatorial number-placement puzzle. In classic Sudoku, the objective is to fill a 9 × 9 grid with digits so that each column, each row, and each of the nine 3 × 3 subgrids that compose the grid contain all of the digits from 1 to 9.

The sudoku dataset has been sourced from Kaggle.
Link: https://www.kaggle.com/datasets/rohanrao/sudoku

The basic code for the project has been sourced from Leetcode.
Link: Java: Generate, validate and solve NxN Sudoku puzzle with visualization, tracking and 100% readable code. - LeetCode Discuss

The algorithm we use in general makes use of backtracking for each position,checking if the correct character is in place.This is then continued for all the available positions in the sudoku.

This is a brute force algorithm and it takes exponential time complexity to solve..

Profiling Inference

```
❯ cat analysis.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 23.81      0.05     0.05    38904     1.29     1.80  printBoard(char**, int)
 21.43      0.10     0.04   616757     0.07     0.15  canPutChar(char**, int, int, char, int)
 19.05      0.14     0.04   707574     0.06     0.06  checkSudokuSubarray(char*, int)
  9.52      0.15     0.02   155616     0.13     0.13  printHorizontalBorder(char**, int)
  9.52      0.17     0.02                             _init
  4.76      0.18     0.01   394562     0.03     0.03  getHorizontalSubArray(char**, int, int)
  4.76      0.20     0.01   190765     0.05     0.05  getVerticalSubArray(char**, int, int)
  4.76      0.20     0.01   122247     0.08     0.08  getMxMSubArray(char**, int, int)
  2.38      0.21     0.01      100    50.00    82.36  getRandomBoard(int)
  0.00      0.21     0.00    38704     0.00     2.31  solveBoard(char**, int, int, int)
  0.00      0.21     0.00     8976     0.00     2.65  isValidSudoku(char**&, int)
  0.00      0.21     0.00     7553     0.00     2.65  isBoardSolved(char**, int)
  0.00      0.21     0.00      101     0.00     0.00  __gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type std::sqrt<int>(int)
  0.00      0.21     0.00      100     0.00     0.00  init_board_properties(int)
  0.00      0.21     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

```
                        Call graph


granularity: each sample hit covers 4 byte(s) for 4.76% of 0.21 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     90.5    0.00    0.19                 main [1]
                0.04    0.14     100/100         solveBoard(char**, int, int, int) <cycle 1> [4]
                0.01    0.00     100/100         getRandomBoard(int) [14]
                0.00    0.00     200/8976        isValidSudoku(char**&, int) [7]
                0.00    0.00     200/38904       printBoard(char**, int) [5]
                0.00    0.00     100/7553        isBoardSolved(char**, int) [10]
-----------------------------------------------
[2]     86.0    0.04    0.14     100+655361  <cycle 1 as a whole> [2]
                0.04    0.05  616757             canPutChar(char**, int, int, char, int) <cycle 1> [3]
                0.00    0.09   38704             solveBoard(char**, int, int, int) <cycle 1> [4]
-----------------------------------------------
                              616757             solveBoard(char**, int, int, int) <cycle 1> [4]
[3]     43.5    0.04    0.05  616757         canPutChar(char**, int, int, char, int) <cycle 1> [3]
                0.03    0.00  489705/707574      checkSudokuSubarray(char*, int) [6]
                0.01    0.00  318451/394562      getHorizontalSubArray(char**, int, int) [11]
                0.01    0.00  119684/190765      getVerticalSubArray(char**, int, int) [12]
                0.00    0.00   51570/122247      getMxMSubArray(char**, int, int) [13]
                               38604             solveBoard(char**, int, int, int) <cycle 1> [4]
-----------------------------------------------
                               38604             canPutChar(char**, int, int, char, int) <cycle 1> [3]
                0.04    0.14     100/100         main [1]
[4]     42.6    0.00    0.09   38704         solveBoard(char**, int, int, int) <cycle 1> [4]
                0.05    0.02   38704/38904       printBoard(char**, int) [5]
                0.00    0.02    7453/7553        isBoardSolved(char**, int) [10]
                              616757             canPutChar(char**, int, int, char, int) <cycle 1> [3]
-----------------------------------------------
                0.00    0.00     200/38904       main [1]
                0.05    0.02   38704/38904       solveBoard(char**, int, int, int) <cycle 1> [4]
[5]     33.3    0.05    0.02   38904         printBoard(char**, int) [5]
                0.02    0.00  155616/155616      printHorizontalBorder(char**, int) [8]
```

From this we get to know that almost 60% of the execution time is spent in the top 3 functions.

- printBoard()
- canPutChar()
- checkSudokuSubarray()

printBoard() can't be parallelized,so we have to reduce the amount of times its called in the program.We have tried to do this by only printing the board when completely solved and not in between.

So, if we are able to parallelize these functions,we will be able to reduce the execution time by a large amount.
The checkSudokuSubarray() function makes calls to 3 sub functions, which each get the horizontal row,the column and the 3x3 smaller matrix.We have tried to parallelize each of them by creating threads to fill each subarray.However this increases the time taken with overhead of creating the thread and communicating between threads for maintaining the shared variable.

This issue is resolved by also parallelizing the main loop where we solve for a particular sudoku.The loop runs for each member of the 1000-1500 sudoku that we run as part of the test cases.

We have also optimized the code slightly by reducing the nested loop for getting each member of 3x3 subarray and replacing it with index modulo and division operations.

OpenMP Code

```
#include "stdlib.h"
#include "iostream"
#include "math.h"
#include "omp.h"
#include <fstream>
#include <string>
#include <vector>
#include <chrono>
using namespace std::chrono;

using namespace std;

int WIDTH_9X9 = 9;
int BOARD_WIDTH = WIDTH_9X9;
int SUB_WIDTH = ((int)sqrt(BOARD_WIDTH));
char START_CHAR = '1';

bool solveBoard(char **board, int rStart, int cStart, int n);
char *getHorizontalSubArray(char **board, int ix, int n)
{
   char *subarray = new char[n];
#pragma omp parallel for

   for (int i = 0; i < n; i++)
   {
       subarray[i] = board[ix][i];
   }

   return subarray;
}

char *getVerticalSubArray(char **board, int ix, int n)
{
   char *subarray = new char[n];
#pragma omp parallel for
   for (int i = 0; i < n; i++)
   {
       subarray[i] = board[i][ix];
```

```
    }
    return subarray;
}

char *getMxMSubArray(char **board, int ix, int n)
{
    char *subarray = new char[n];

    int cOffset = SUB_WIDTH * (ix % SUB_WIDTH);
    int rOffset = SUB_WIDTH * (ix / SUB_WIDTH);

    #pragma omp parallel for
    for(int i=0; i<n; i++){
        subarray[i] = board[rOffset+(i/3)][cOffset+(i%3)];
    }

    return subarray;
}

bool checkSudokuSubarray(char *array, int n)
{
    int nBOARD_WIDTH = n;
    bool *temp = new bool[nBOARD_WIDTH];

    for (int i = 0; i < nBOARD_WIDTH; i++)
    {
        // cout<<array[i]<<" ";
        temp[i] = false;
    }
    bool res = true;
#pragma omp parallel for shared(res,temp)
    // {
        for (int i = 0; i < nBOARD_WIDTH; i++)
        {

                if ((array[i] >= START_CHAR) && (array[i] <= (START_CHAR +
nBOARD_WIDTH)))
            {
                int iPos = (array[i] - START_CHAR);
                if (false == temp[iPos])
```

```
                {
                    temp[iPos] = true;
                }
                else
                {
                    // cout<<"Why u do this? "<<array[i]<<endl;
                    // return false;
                    res = false;
                }
            }
            else if (array[i] == '.')
            {
                continue;
            }
            else
            {
                // cout<<"Wrong at "<<array[i]<<endl;
                // return false;
                res = false;
            }
        }
    // }
    // cout<<endl;
    return res;
}

bool isValidSudoku(char **&board, int n)
{
    if (nullptr == board)
    {
        cout << "board is null." << endl;
        return false;
    }
    if (n <= 0)
    {
        cout << "board.length is <= 0." << endl;
        return false;
    }

    // check rows
```

```
    for (int i = 0; i < n; i++)
    {
            if (false == checkSudokuSubarray(getHorizontalSubArray(board, i,
n), n))
        {
            cout << "Invalid Horizontal Subarray" << i << endl;
            return false;
        }
    }
    // check columns
    for (int i = 0; i < n; i++)
    {
          if (false == checkSudokuSubarray(getVerticalSubArray(board, i, n),
n))
        {
            cout << "Invalid vertical subarray " << i << endl;
            return false;
        }
    }
    // check 3x3
    for (int i = 0; i < n; i++)
    {
        if (false == checkSudokuSubarray(getMxMSubArray(board, i, n), n))
        {
            cout << "Invalid subarray" << endl;
            return false;
        }
    }

    return true;
}

void printHorizontalBorder(char **board, int n)
{
    for (int c = 0; c < n; c++)
    {
        if (0 == (c % SUB_WIDTH))
        {
            cout << "-";
        }
```

```
        cout << "--";
    }
    cout << "-" << endl;
}

void printBoard(char **board, int n)
{
    if (nullptr == board)
        return;
    cout << "\n";
    for (int r = 0; r < n; r++)
    {
        if (0 == (r % SUB_WIDTH))
            printHorizontalBorder(board, n);
        for (int c = 0; c < n; c++)
        {
            if (0 == (c % SUB_WIDTH))
            {
                cout << ("|");
            }
            cout << " " << board[r][c];
        }
        cout << "|" << endl;
    }
    printHorizontalBorder(board, n);
}

bool canPutChar(char **board, int r, int c, char digit, int n)
{
    if ((r >= 0) && (r < n))
    {
        if ((c >= 0) && (c < n))
        {
            if ('.' == board[r][c])
            {
                board[r][c] = digit;
                if (checkSudokuSubarray(getHorizontalSubArray(board, r, n),
n) &&
                      checkSudokuSubarray(getVerticalSubArray(board, c, n),
n) &&
```

```
                     checkSudokuSubarray(getMxMSubArray(board, SUB_WIDTH *
(r / SUB_WIDTH) + c / SUB_WIDTH, n), n) &&
                   solveBoard(board, r, c + 1, n))
               {
                   return true;
               }
               else
               {
                   board[r][c] = '.';
                   return false;
               }
           }
           else
           {
               // already contains a potentially valid digit
               return true;
           }
       }
   }
   return true;
}

bool isBoardSolved(char **board, int n)
{
   bool isSolved = true;
   #pragma omp parallel for collapse(2)
   for (int r = 0; r < n; r++)
   {
       for (int c = 0; c < n; c++)
       {
           if ('.' == board[r][c])
               // #pragma omp cancel parallel
               isSolved = false;
               // return false;
       }
   }
   return isSolved && isValidSudoku(board, n);
}

bool solveBoard(char **board, int rStart, int cStart, int n)
```

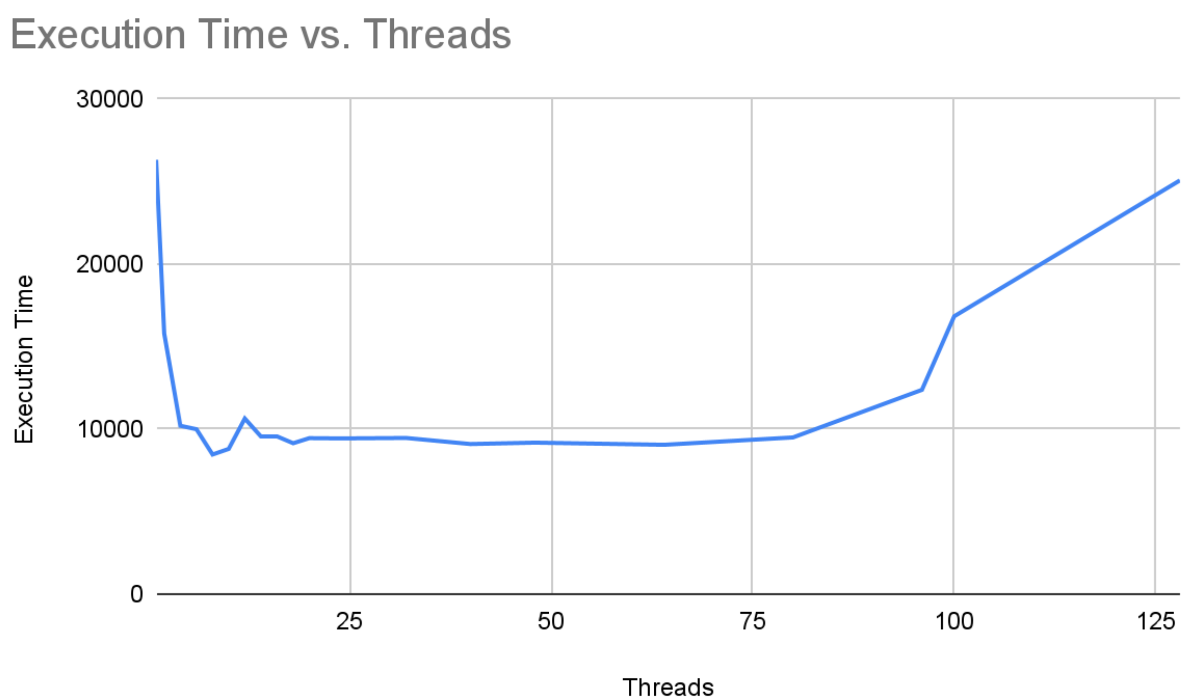
```
{
   if (cStart >= n)
   {
       // roll over to the next row
       cStart = 0;
       rStart++;
   }

    cout  <<  "\nSolved  :"  <<  ((rStart  *  BOARD_WIDTH  +  cStart)  *  100)  /
(BOARD_WIDTH * BOARD_WIDTH);
   // printBoard(board, n);

   bool bPutChar = false;
   for (int r = rStart; r < n; r++)
   {
       for (int c = cStart; c < n; c++)
       {
           for (char i = 0; i < n; i++)
           {
                bPutChar = canPutChar(board, r, c, (char)(START_CHAR + i),
n);
               if (bPutChar)
                   break; // potentially solved !
           }
           if (false == bPutChar)
               return false; // exhausted all possibilities
       }
       cStart = 0; // for next cycle cStart starts from zero.
   }
   return isBoardSolved(board, n);
}

int main()
{
   auto start = high_resolution_clock::now();
   fstream newfile;
   vector<string> sudokulist;
    newfile.open("nsodoku.txt",  ios::in);  //  open  a  file  to  perform  read
operation using file object
   cout << "Hi" << endl;
```

```
    if (newfile.is_open())
    { // checking whether the file is open
        string tp;
        while (getline(newfile, tp))
        { // read data from file object and put it into string.
            sudokulist.push_back(tp);
        }
        newfile.close(); // close the file object.
    }
    #pragma omp parallel for
    for (int i = 0; i < sudokulist.size(); i++)
    {
        char **board = new char *[WIDTH_9X9];
        for (int i = 0; i < WIDTH_9X9; i++)
        {
            // Declare a memory block of size n
            board[i] = new char[WIDTH_9X9];
        }
        string testcase = sudokulist[i];

        for (int i = 0; i < WIDTH_9X9; i++)
        {
            for (int j = 0; j < WIDTH_9X9; j++)
            {
                if (testcase[(i * 9) + j] == '0')
                {
                    board[i][j] = '.';
                }
                else
                {
                    board[i][j] = testcase[(i * 9) + j];
                }
            }
        }
        cout << ("\nProblem board:");
        printBoard(board, WIDTH_9X9);
        if (isValidSudoku(board, WIDTH_9X9))
        {
              cout << ("isValidSudoku() before solving returned true.") <<
endl;
```

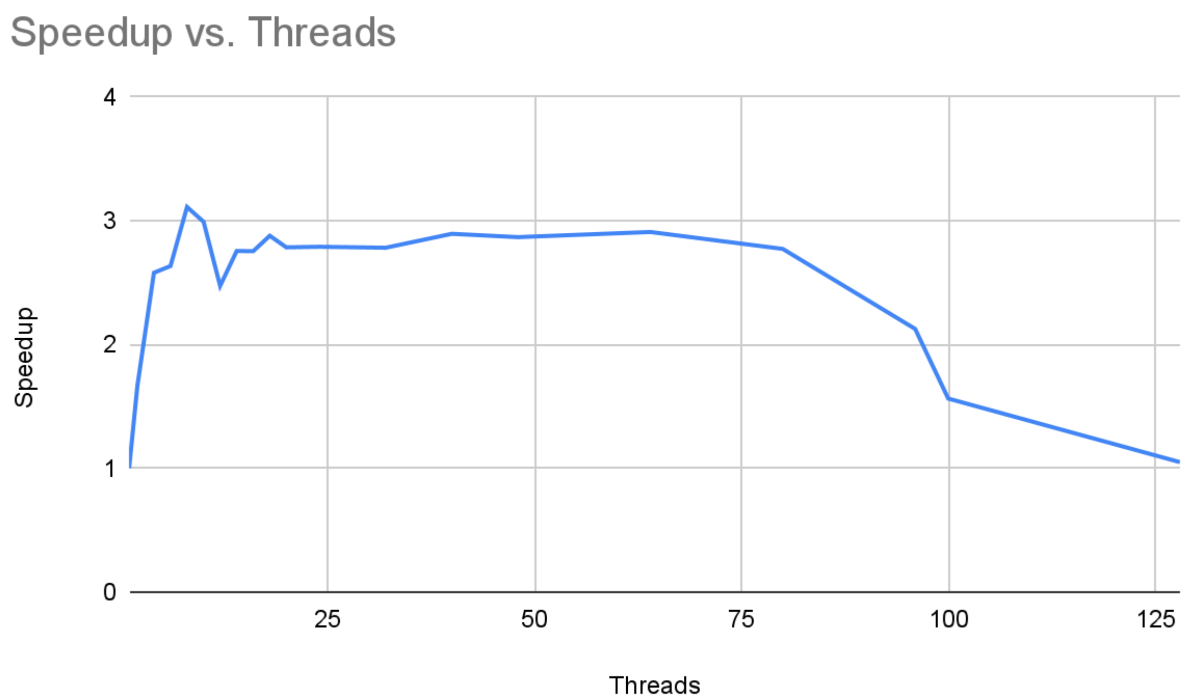
```
            solveBoard(board, 0, 0, WIDTH_9X9);
            // cout << ("\nSolved board:");
            // printBoard(board, WIDTH_9X9);
            if (isValidSudoku(board, WIDTH_9X9))
            {
                cout << ("isValidSudoku() after solving returned true.") <<
endl;
                if (isBoardSolved(board, WIDTH_9X9))
                {
                          cout << ("isBoardSolved() after solving returned
true.") << endl;
                }
                else
                {
                          cout << ("isBoardSolved() after solving returned
false.") << endl;
                }
            }
            else
            {
                 cout << ("isValidSudoku() after solving returned false.")
<< endl;
            }
        }
        else
        {
             cout << ("isValidSudoku() before solving returned false.") <<
endl;
        }
    }
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);

    // To get the value of duration use the count()
    // member function on the duration object
    cout << duration.count() << endl;
    cout<<"END ONE TIME" << endl;
    return 0;
}
```

Result

Threads vs Time



Speedup vs Processors



Parallelization Fraction

Threads	Execution Time	Speedup	Speedup %	Parallelization factor
1	26315	1	0	#DIV/0!
2	15791	1.666455576	66.6455576	0.7998479954
4	10200	2.579901961	157.9901961	0.8165178289
6	9989	2.634397838	163.4397838	0.7444879346
8	8458	3.111255616	211.1255616	0.7755272658
10	8796	2.991700773	199.1700773	0.7397133025
12	10637	2.473911817	147.3911817	0.649943862
14	9549	2.755785946	175.5785946	0.6861368918
16	9554	2.75434373	175.434373	0.679399582
18	9140	2.879102845	187.9102845	0.6910619083
20	9448	2.785245555	178.5245555	0.674700241
24	9432	2.789970314	178.9970314	0.6694677362
32	9459	2.782006555	178.2006555	0.6612100298
40	9091	2.894621054	189.4621054	0.6713144984
48	9176	2.867807323	186.7807323	0.6651590186

64	9045	2.909342178	190.9342178	0.6666968263
80	9491	2.772626699	177.2626699	0.6474239797
96	12376	2.126292825	112.6292825	0.5352736582
100	16831	1.563484047	56.34840473	0.3640432445
128	25068	1.049744694	4.974469443	0.04776055093
			Avg Parallelization	0.6413519134

Inference

From the Thread vs Time and Speedup vs Processors plot, we can see that the performance of the program has increased with increase in number of threads.The cost of the operation of getting the correct letter in the correct position is quite high, and therefore the context switches and other phenomenon were unable to affect the time taken to execute in parallel threads.

We also see the best performance, i.e. the best degree of parallelism was observed when the program was run using 8 threads.This was the optimal scenario where the context switches did not increase the runtime of the program by much and the effect of parallelism was able to take place and reduce the time needed to execute the program.8 threads was the best spot because our threads in the subarray functions required around 9 threads to return the 9 elements.This also lines up with executing upto 9 sudoku solving at a time.

We were able to get a maximum of 211% improvement in performance while using 8 threads.