COMP 429/529: Project 1

Berkay Barlas

March 22, 2019

Date Performed: March 17, 2019
Instructor: Didem Unat

In this assignment I developed my parallel implementations on top of given serial version for two different applications; an image blurring algorithm and sudoku solver using OpenMP.
While the first application in data parallelism, the second application in task parallelism.

In this assignment I have completed

- Parallel Version of Image Blurring
- Performance Study for Part I
- Parallel Version of Sudoku Part A, Part B, Part C
- Performance Study for Part II

1 Part I: Image Blurring

In the first part of this assignment I implemented a parallel version of a simple image blurring algorithm with OpenMP which takes an input image and outputs a blurred image.

I used #pragma omp for collapse() in getGaussian(), loadImage(), saveImage(), applyFilter(), averageRGB() methods since all of have nested for loops that can be parallelized. The biggest nested for loop is in applyFilter() and it can be parallelizable as below. collapse(5) can not be used because last two for loops depends on previous loops.

```
1    #pragma omp parallel for collapse(3)
2    for (d=0 ; d<3 ; d++) {
3     for (i=0 ; i<newImageHeight ; i++) {
4      for (j=0 ; j<newImageWidth ; j++) {
5       for (h=0 ; h<filterHeight ; h++) {
6        for (w=0 ; w<filterWidth ; w++) {
7            newImage[d][i][j] += filter[h][w]*image[d][h+i][w+j];
8    }}}}}
```

1.1 Stability Test

Serial version execution time:

Coffee Image: 13.64
Strawberry Image: 27.56

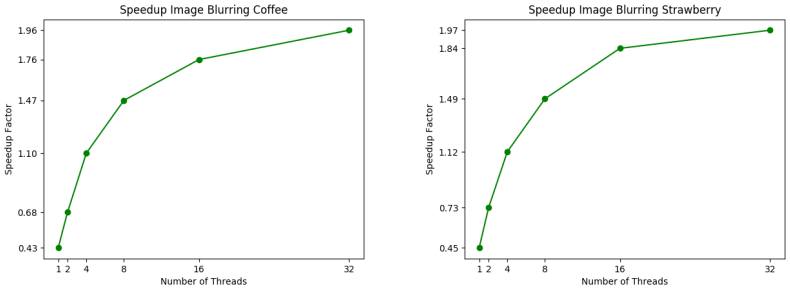
Paralel version with single thread execution time:

Coffee Image: 30.88
Strawberry Image: 55.93

Which thread number gives the best performance?

32 thread count gives the best performance for both blurring applications. The reason of serial version performs better than parallel version with 1 thread is Parallization overhead. The difference between them caused by the execution time of parallization.

Results



(a) Speedup results for the blurring on coffee image.

(b) Speedup results for the blurring on strawberry image.

Figure 1: Speedup figures for image blurring application

Explanation of Speedup Curve

Due to parallization overhead a speedup with value bigger than 1 is observed after 4 threads for both applications.

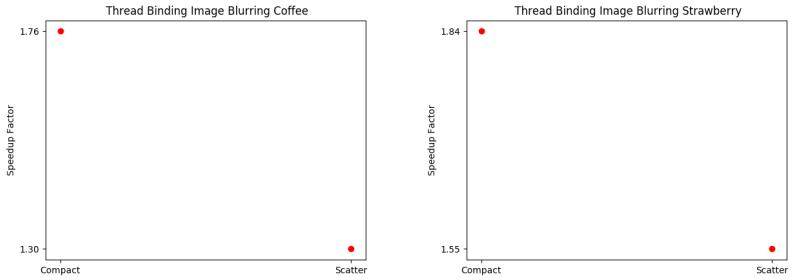
Even though, a linear/perfect speedup is not expected the results are actually worse than what is expected. Even the parallel version with 32 thread on 32 core cluster gives only around 2x speedup.
One of reasons of that is intel compiler optimizations in serial version which is already very fast.

1.2 Thread Binding Test

In the compact mapping, multiple threads are mapped as close as possible to each other, while in the scatter mapping, the threads are distributed as evenly as possible across all cores.

Different mapping strategies; Compact and Scatter

Results



(a) Results for the blurring on coffee image.

(b) Results for the blurring on strawberry image.

Figure 2: Speedup figures for image blurring application

Which Mapping Gives Better Performance, Why?

Compact gives better performance for both images because when neighbouring threads are accessing the same(Temporal Locality) or nearby data(Spatial Locality); the data which is brought into the cache by one thread can be used by the other, avoiding a costly memory access. If the tasks were longer scatter could perform better.

2 PART II: Parallel Sudoku Solver

In the second part of this assignment, I parallelized a serial sudoku solver with OpenMP which takes a sudoku problem as an input and finds all possible solutions from it using a brute force search for searching by all possible solutions to the problem.

Part A

In this Part, I defined parallel section while calling solveSudoku() method. I used #pragma omp task in different places inside solveSudoku() method, after several run and experiments using I selected the one that gives best performance.

```
if(matrix[row][col] != EMPTY) {
//#pragma omp task firstprivate(col, row)
    if (solveSudoku(row, col+1, matrix, box_sz, grid_sz)) {
            printMatrix(matrix, box_sz);
    }
} else {
int num;
  for (num = 1; num <= box_sz; num++)
  {
   if (canBeFilled(matrix, row, col, num, box_sz, grid_sz))
   {
    #pragma omp task firstprivate(num, col, row)
    {
     int tempMatrix[MAX_SIZE][MAX_SIZE];
     int i;
     int j;
     for(i=0; i<box_sz; i++) {
         for( j=0; j<box_sz; j++){
             tempMatrix[i][j] = matrix[i][j];
         }
     }
     tempMatrix[row][col] = num;
     if (solveSudoku(row, col+1, tempMatrix, box_sz, grid_sz))
         printMatrix(tempMatrix, box_sz);
    }
   }
  }
}
```

Part B

In this Part, I changed funtion signature to pass the task depth. Every time, a new task is created the depth value is incereased by 1 and passed to recursive method as parameter.

To find a good cutoff value I run the program with different cutoff parameters. Overall cutoff value 30 gived the best result.

```
if (canBeFilled(matrix, row, col, num, box_sz, grid_sz))
{
 #pragma omp task firstprivate(num, col, row, depth) if (depth <
    MAX_DEPTH)
 {
  depth++;
  int tempMatrix[MAX_SIZE][MAX_SIZE];
  int i;
  int j;
  for(i=0; i<box_sz; i++) {
      for( j=0; j<box_sz; j++){
          tempMatrix[i][j] = matrix[i][j];
      }
  }
  tempMatrix[row][col] = num;
  if (solveSudoku(row, col+1, tempMatrix, box_sz, grid_sz, depth))
      printMatrix(tempMatrix, box_sz);
 }
}
```

Part C

In this Part, I created a shared variable found and passed it as parameter to solveSudoku function which stops priting solution and creation of new tasks.

```
#pragma omp parallel shared(found)
{
  #pragma omp single
  {
  solveSudoku(0, 0, matrix, box_sz, grid_sz, &found);
  }
}
```

I also used #pragma omp critical when I changing value of found.

```
    ...
    #pragma omp critical
        *found = 1;
    ...
```

2.1 Scalability Test

2.1.1 Part A

Serial version execution time: 48.10

Paralel version with single thread execution time: 78.69

Which thread number gives the best performance?
32 thread count gives the best performance.

Results

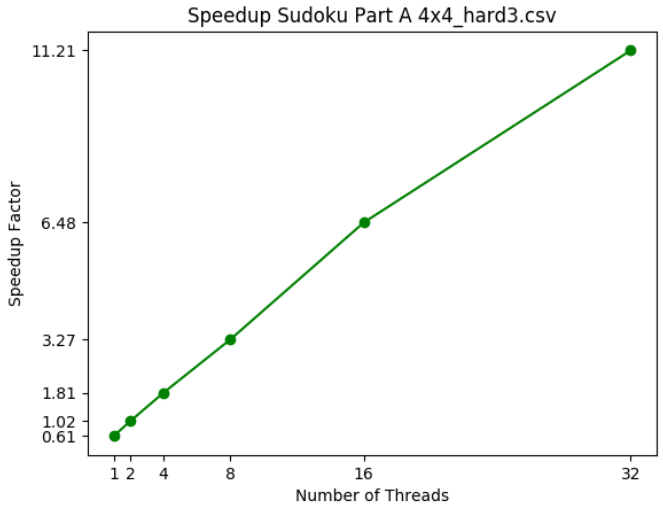


Figure 3: Results for the Sudoku 4x4hard3 using algorithm in.

Explanation of Speedup Curve

Due to parallization overhead a speedup with value bigger than 1 is observed after 2 threads.

The task Parallization of serial version results in the creation of different task causes to a great overhead. Therefore, the speedup results are lower than expected. Even with 32 thread speedup is just around 11.

2.1.2 Part B

Serial version execution time: 48.10

Paralel version with single thread execution time: 74.50

Which thread number gives the best performance?

32 thread count gives the best performance.

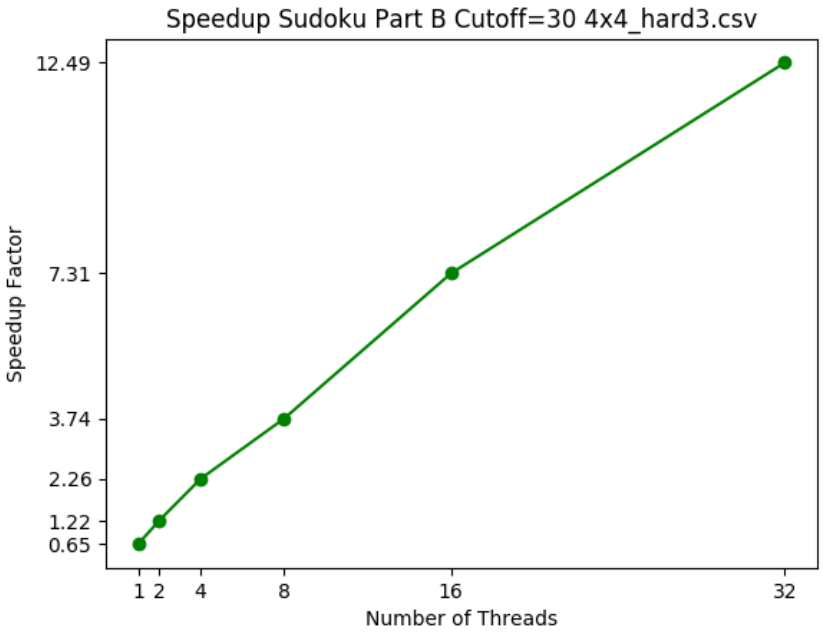


Figure 4: Results for the blurring on strawberry image.

Explanation of Speedup Curve

Due to parallization overhead a speedup with value bigger than 1 is observed after 2 threads.

In order to improve the performance of the previous parallel version, a cutoff parameter to limit the number of parallel tasks is needed.

I defined a variable called depth and passed it as parameter to recursive method to prevent task creation after certain depth in the call-path tree. After that depth switch to the serial execution and do not generate more tasks. To determine that cut off parameter, I executed parallel program with several different values.

The speedup curve is linear which is expected.

2.1.3 Part C

Serial version execution time: 0.33

Paralel version with single thread execution time: 0.57

Which thread number gives the best performance?
32 thread count gives the best performance.

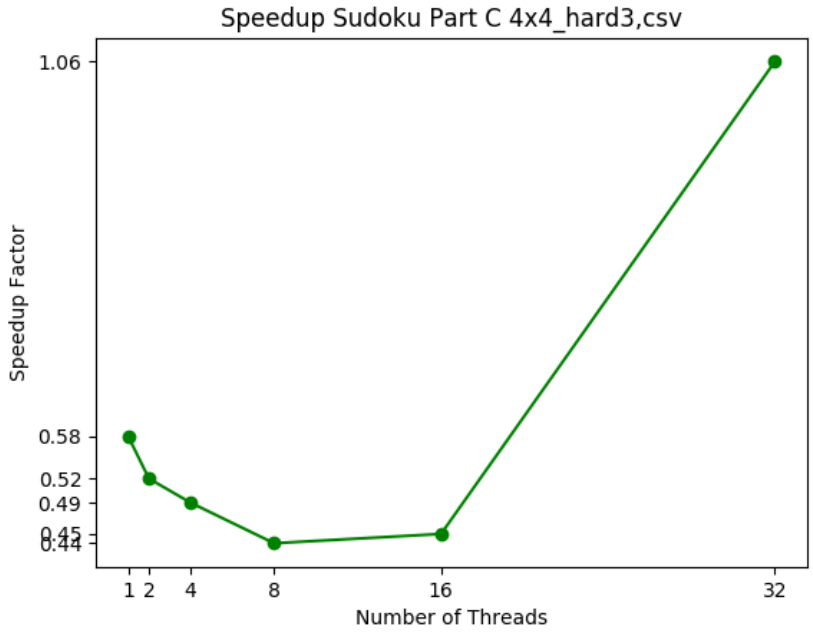


Figure 5: Results for the blurring on strawberry image.

Explanation of Speedup Curve

Stopping the execution after finding a solution is very easy for serial version which can be done by returning a different value inside of for loops when a solution is found. In order to guraantee single solution in parallel version a shared variable 'found' which will stop further task creation and execution can be defined. In this application, parallized application results are really poor because a solution might be found by one of the tasks but other tasks that are created previously will continue execution and this creates a great overhead compared to serial version.

Also, from 1 thread to 16 thread, increasing the number of threads decreases to speedup. This is most likely to be caused by increased number of total task created and needs to be executed, even though, one of the tasks already find a solution.

2.2 Thread Binding Test

2.2.1 Part A

Different mapping strategies; Compact and Scatter

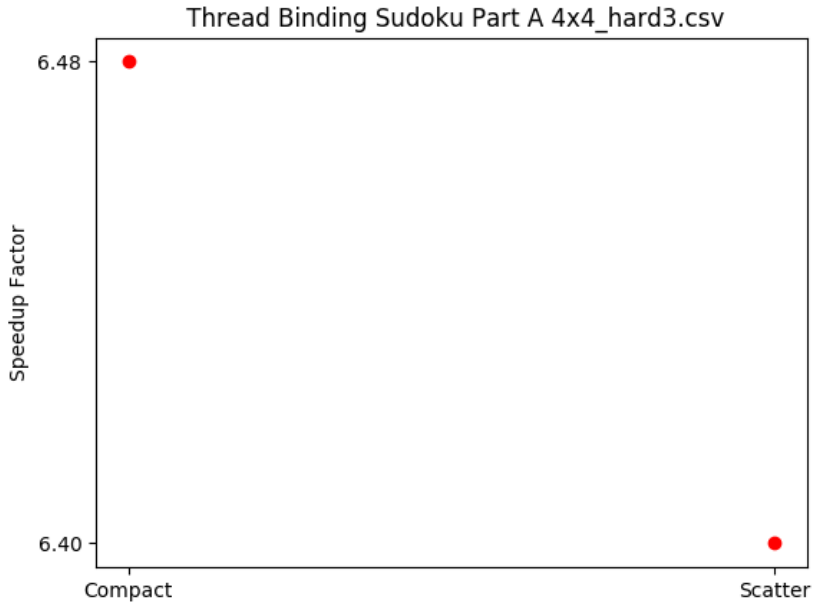


Figure 6: Results for the blurring on strawberry image.

Which Mapping Gives Better Performance, Why?

Compact gives better performance for both images because when neighbouring threads are accessing the same(Temporal Locality) or nearby data(Spatial Locality); the data which is brought into the cache by one thread can be used by the other, avoiding a costly memory access. If the tasks were longer scatter could perform better.

2.2.2 Part B

Different mapping strategies; Compact and Scatter

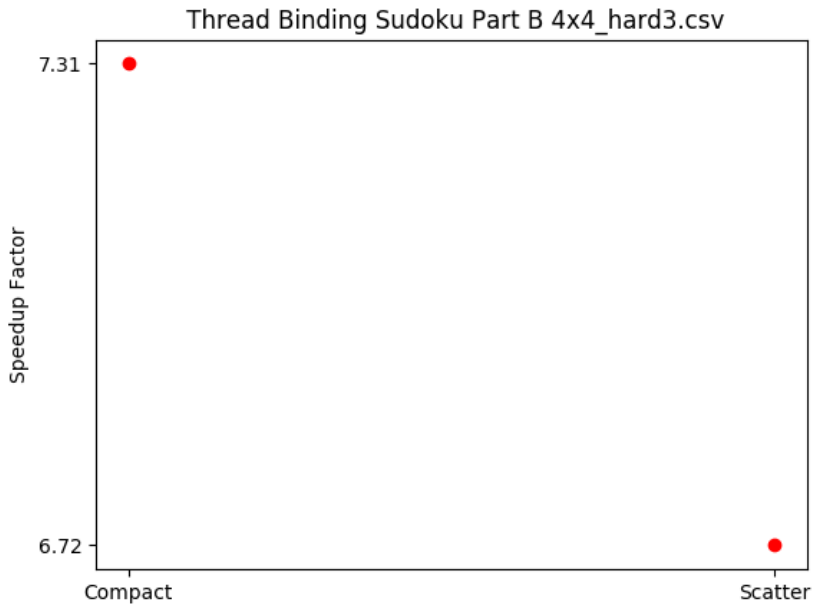


Figure 7: Results for the blurring on strawberry image.

Which Mapping Gives Better Performance, Why?

Compact gives better performance for both images because when neighbouring threads are accessing the same(Temporal Locality) or nearby data(Spatial Locality); the data which is brought into the cache by one thread can be used by the other, avoiding a costly memory access. If the tasks were longer scatter could perform better.

2.2.3 Part C

Different mapping strategies; Compact and Scatter

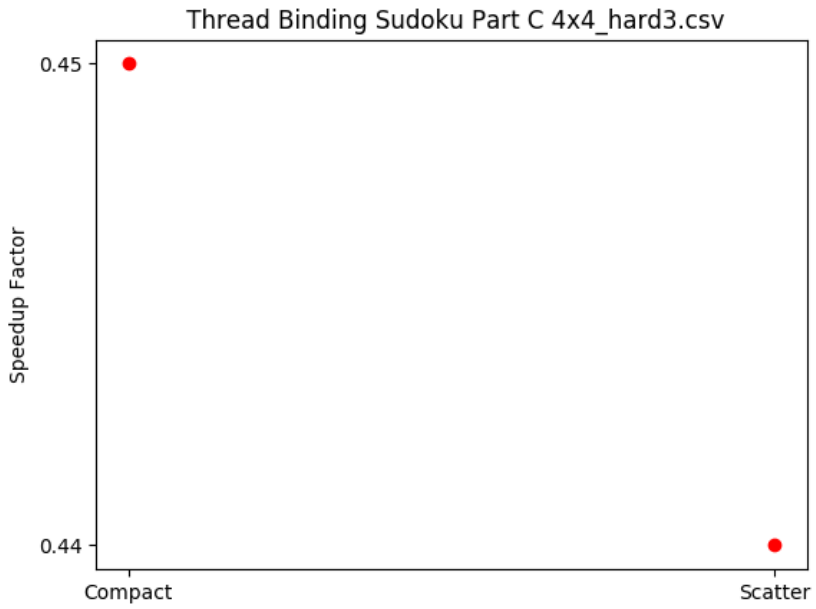


Figure 8: Results for the blurring on strawberry image.

Which Mapping Gives Better Performance, Why?

Compact gives better performance for both images because when neighbouring threads are accessing the same(Temporal Locality) or nearby data(Spatial Locality); the data which is brought into the cache by one thread can be used by the other, avoiding a costly memory access. If the tasks were longer scatter could perform better.

2.3 Tests on Sudoku Problems of Different Grids

Part-B

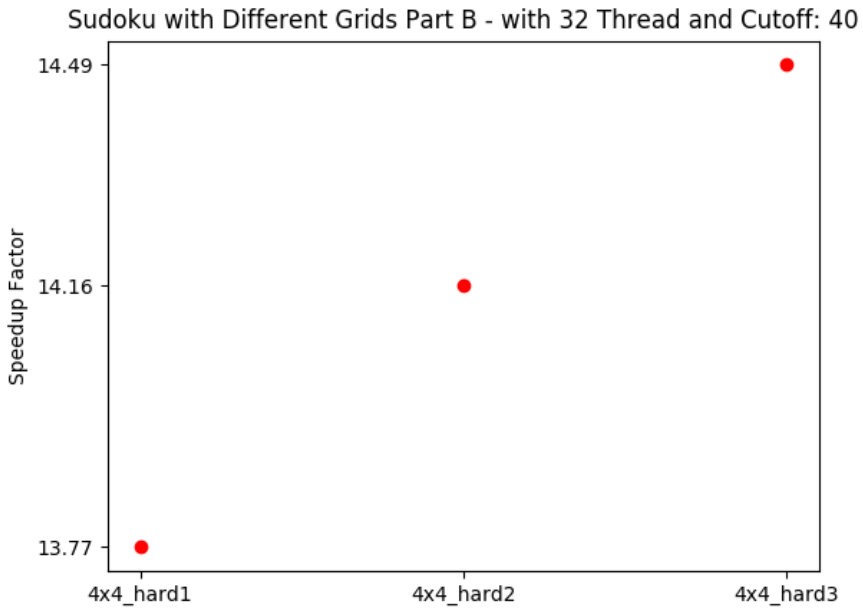


Figure 9: Results for the 32 Thread Parallel Sudoku solver in Part B with different sizes and difficulties.

When the difficulties of sudoku problem increases parallized algorithm in Part B performs better over serial serial version. Effectiveness of parallization increases beacuse when the task difficulty increased the execution time ratio of parallelizable partion over non-parallelizable partion increases.

3 Formulas Used

a. *Speedup*

$$Speedup = \frac{\mathrm{T1}}{\mathrm{Tp}}$$

b. *Amdahl's Law*

$$Tp \geq W serial + \frac{\mathrm{Wparallel}}{\mathrm{P}}$$