BEYZA ÇAVUŞOĞLU

Part 1) MANDELBROT

CODE CHANGES:

I parallelized both loop by using collapse(2). I got rid of the pixelCount since it was creating extra pixels in the visual because of the race conditions happening due to the loop parallelizations. Instead of such global value, I used an index value described below in Figure 3, line 181.

```
void mandelbrot::loop() {
  finished = false;
 // pixelCount = 0;

  #pragma omp parallel for collapse(2)
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      calculatePixel(x, y);
      //++pixelCount;
    }
  }

  finished = true;
}
```

In the calculatePixel, there is a necessity to prevent the race condition of setColor() by different tasks (line 164). #pragma omp critical used fort hat purpose which identifies a section of code that must be executed by a single thread at a time.

```
148 void mandelbrot::calculatePixel(int pix_X, int pix_Y) {
149   LD x0 = picToMand_x(double(pix_X));
150   LD y0 = picToMand_y(double(pix_Y));
151
152   LD x = 0;
153   LD y = 0;
154   LD xtemp = 0;
155   LD iter = 0;
156
157   while (((x * x + y * y) <= 4) && iter <= max_iteration) {
158     xtemp = x * x - y * y + x0;
159     y = 2 * x * y + y0;
160     x = xtemp;
161     iter = iter + 1;
162   }
163   #pragma omp critical
164   setColor(pix_X, pix_Y, iter);
165 }
```

As i explained before due to the race conditions and extra created wrong pixels in the image, to get rid of the global pixelcount variable in loop() function, I defined a new index which uses x and y values of threads to determine the position and color by the index in vertexPixels array.

```
void mandelbrot::setColor(int x, int y, int iterations) {

  int color = 0;
  if (iterations >= max_iteration) {
    color = 0;
  } else {
    color = 1;
  }

  // You can play with the following values to get different coloring patterns
  int red = 256 - (iterations % 256);
  int green = 256 - (iterations % 256);
  int blue = 256 - (iterations % 256);

  int index = y * width + x; // got rid of the pixelcount global variable

  vertexPixels[index].position = sf::Vector2f(x, y);
  vertexPixels[index].color =
      color == 1 ? sf::Color(red, green, blue % 256, 255)
                 : sf::Color(0, 0, 0, 255);
}
```

I have also added extra points to test as zoom_in_test_extra1 and zoom_in_test_extra2.

```
#ifdef KUACC
  RUN_TEST(img, zoom_in_test_mixed);
  RUN_TEST(img, zoom_in_test_all_white);
  RUN_TEST(img, zoom_in_test_all_black);
  RUN_TEST(img, zoom_in_test_extra1);
  RUN_TEST(img, zoom_in_test_extra2);

```

```
std::vector<std::pair<int, int>> zoom_in_test_extra1 = {
    {369, 211},
    {910, 355},
    {624, 457},
    {417, 751},
};

std::vector<std::pair<int, int>> zoom_in_test_extra2 = {
    {319, 314},
    {929, 455},
    {664, 657},
    {213, 738},
};
```

a) Scheduling Tests:

I calculated SpeedUp values as : Serial Execution Time / Parallel Execution Time.

SCHEDULING	
Version (32 thread)	Total Time (ms)
Parallel Static - Mixed	4649
Parallel Dynamic 10 - Mixed	4469
Parallel Dynamic 100 - Mixed	4567
Parallel Dynamic 20 - Mixed	7222
Parallel Static - All White	2255
Parallel Dynamic 10 - All White	2302
Parallel Dynamic 100 - All White	1907
Parallel Dynamic 20 - All White	4419
Parallel Static - All Black	7012
Parallel Dynamic 10 - All Black	6715
Parallel Dynamic 100 - All Black	6914
Parallel Dynamic 20 - All Black	13429
SPEED-UPS	
Speedup-Dynamic10- Mixed	1.040277467
Speedup- Dynamic10-All Black	1.044229337
Speedup-Dynamic10-All White	0.979582971
Speedup-Dynamic100- Mixed	1.017954894
Speedup- Dynamic100-All Black	1.014174139
Speedup-Dynamic100-All White	1.182485579
Speedup-Dynamic20- Mixed	0.643727499
Speedup- Dynamic20-All Black	0.510296447
Speedup-Dynamic20-All White	0.522153548

Even though there is not a huge difference between Dynamic 10 and 100, if we have to choose a better one, Dynamic 100 is better in speedup. Dynamic 20 is even worse than Static Scheduling baseline. The explanation can be as follows: in Dynamic 10 with a smaller chunk size (10), the scheduler assigns smaller portions of work to each thread. This can result in more frequent task scheduling overhead, leading to potential inefficiency. Also, in dynamic 10, frequent task assignments can lead to increased synchronization overhead, as threads may need to coordinate more frequently when acquiring new tasks.

But in Dynamic 100, a larger chunk size (100) is used and this allows each thread to process a more significant amount of work before needing to request more tasks. This can reduce the scheduling overhead and improve overall efficiency. Moreover, with larger chunks, threads have less frequent synchronization, reducing overhead associated with task management.

b) Scalability Tests:

THREADS	Thread Counts						
Version	**1**	**2**	**4**	**8**	**16**	**32**	
Serial - Mixed	36484						Time in ms
Parallel - Mixed	26785	26018	16305	10413	5848	4649	
Serial - All Black	184175						
Parallel - All Black	143375	76244	40930	22611	12822	7012	
Serial - All White	649						
Parallel - All White	196	673	1163	1351	1852	2255	
SPEED-UPS							
Speedup-Mixed	1.362105656	1.40226	2.2376	3.5037	6.23871	7.84771	
Speedup-All Black	1.284568439	2.4156	4.49976	8.14537	14.364	26.2657	
Speedup-All White	3.31122449	0.96434	0.55804	0.48038	0.35043	0.2878	

In the mixed and all-black test, increasing the thread count increased the speed up (mixed – 7.8x and all-black 26.26x). However, for all black test 1 thread and serial worked better than the thread count increase. The reason for these result may differ. The Mandelbrot set has regions with varying levels of complexity and randomness. Some regions may be computationally more intensive, while others may be less so. The distribution of points in the test cases could influence how well parallelization can be utilized. In the "all-black" test case, where all points are at the center of the image, there might be symmetry or repetitive patterns that can be exploited by parallelization. If the computation exhibits regularity or repetitive structures around the central point, multiple threads can work on different parts of the image concurrently, leading to effective parallelization.

Part 2) PARALLEL SUDOKU SOLVER

Part 2A) Task Parallelism

We were asked to parallelize the sudoku solver with task parallelism. For this goal, I have changed the code as below:

#pragma omp critical is used to protect the printing of the solution matrix to avoid data corruption when multiple threads attempt to print simultaneously.

```
void printMatrix(int matrix[MAX_SIZE][MAX_SIZE], int box_sz)
{
    #pragma omp critical
    {
    printf("solution matrix\n");
    int row, col;
    for (row = 0; row < box_sz; row++)
    {
       for (col = 0; col < box_sz; col++)
            printf("%d ", matrix[row][col]);
        printf("\n");
    }
    }
}
```

Then, since task will parallelize the solving part, I defined in main function the lines 147-153 to do task parallelism. #pragma omp single initiate the parallel execution of the solveSudoku function, so only one thread starts the solving process.

```
130 int main(int argc, char const *argv[])
131 {
132 
133     if (argc < 3){
134         printf("Please specify matrix size and the CSV file name as inputs.\n");
135         exit(0);
136     }
137 
138     int box_sz     = atoi(argv[1]);
139     int grid_sz = sqrt(box_sz);
140     char filename[256]; strcpy(filename, argv[2]);
141 
142     int matrix[MAX_SIZE][MAX_SIZE];
143 
144     readCSV(box_sz, filename, matrix);
145 
146         double time1 = omp_get_wtime();
147     #pragma omp parallel
148     {
149         #pragma omp single
150         {
151         solveSudoku(0, 0, matrix, box_sz, grid_sz);
152         }
153     }
154     printf("Elapsed time: %0.6lf\n", omp_get_wtime() - time1);
155     return 0;
156 }
```

In the solveSudoku function, line 52 is defined fort ask parallel sudoku solving. But for every thread, in order to prevent the race conditions, a private version of “matrix” is defined and the original matrix is copied to this privateMatrix. Also, num col row values are private to each thread and I used firstprivate so they would have same initialization. I put line 67 so all threads can be synched in the final to return.

```
45     else
46     {
47         int num;
48         for (num = 1; num <= box_sz; num++)
49         {
50             if (canBeFilled(matrix, row, col, num, box_sz, grid_sz))
51             {
52             #pragma omp task firstprivate(num, col, row, matrix)
53                 {
54                     int privateMatrix[MAX_SIZE][MAX_SIZE];
55                     memcpy(privateMatrix, matrix, sizeof(int) * MAX_SIZE * MAX_SIZE);
56 
57 
58                         privateMatrix[row][col] = num;
59 
60                         if (solveSudoku(row, col + 1, privateMatrix, box_sz, grid_sz))
61                             printMatrix(privateMatrix, box_sz);
62 
63                         privateMatrix[row][col] = EMPTY;
64                 }
65             }
66         }
67     #pragma omp taskwait
68     }
69     return 0;
70 }
```

Part 2B) Task Parallelism with Cutoff

Since the performance was not good at all in the Part 2A, I used a cutoff parameter to limit the number of parallel tasks. I played around and chosen 20 as the cutoff for a better performance.

```
10   #define MAX_DEPTH_CUTOFF 20
```

```
33   int solveSudoku(int row, int col, int matrix[MAX_SIZE][MAX_SIZE], int box_sz, int grid_sz, int depth_cutoff)
34   {
```

I put in line 60 an if-conditional to check the cutoff to not create more than limit number of tasks.

```
52   else
53   {
54       int num;
55       for (num = 1; num <= box_sz; num++)
56       {
57           if (canBeFilled(matrix, row, col, num, box_sz, grid_sz))
58           {
59           #pragma omp task firstprivate(num, col, row, matrix, depth_cutoff) if (depth_cutoff < MAX_DEPTH_CUTOFF)
60               {
61                   depth_cutoff = depth_cutoff + 1;
62
63                   int privateMatrix[MAX_SIZE][MAX_SIZE];
64                   memcpy(privateMatrix, matrix, sizeof(int) * MAX_SIZE * MAX_SIZE);
65
66
67                       privateMatrix[row][col] = num;
68
69                       if (solveSudoku(row, col + 1, privateMatrix, box_sz, grid_sz, depth_cutoff))
70                           printMatrix(privateMatrix, box_sz);
71
72                       privateMatrix[row][col] = EMPTY;
73               }
74           }
75       }
76   #pragma omp taskwait
77   }
78   return 0;}
```



```
161   #pragma omp parallel
162       {
163   #pragma omp single
164           {
165               solveSudoku(0, 0, matrix, box_sz, grid_sz, depth_cutoff);
166           }
167       }
```

Part 2C) Task Parallelism with Early Termination

I added a flag “solution_found” to stop the algorithm searching for all the sudoku solutions when one of the tasks found a solution. In line 48 and 72, critical section was necessary so that only one thread at that time can set the value of solution_found = 1 to prevent race condition.

```c
int solveSudoku(int row, int col, int matrix[MAX_SIZE][MAX_SIZE], int box_sz, int grid_sz, int *solution_found)
{
    if (col > (box_sz - 1))
    {
        col = 0;
        row++;
    }
    if (row > (box_sz - 1))
    {
        return 1;
    }
    if (matrix[row][col] != EMPTY)
    {
        if (solveSudoku(row, col + 1, matrix, box_sz, grid_sz, solution_found))
        {
            printMatrix(matrix, box_sz);
            #pragma omp critical
            {
                *solution_found = 1;
            }
        }
    }
```

```c
    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, matrix)
            {
             if (!(*solution_found))
                {
                    int privateMatrix[MAX_SIZE][MAX_SIZE];
                    memcpy(privateMatrix, matrix, sizeof(int) * MAX_SIZE * MAX_SIZE);

                    privateMatrix[row][col] = num;

                    if (solveSudoku(row, col + 1, privateMatrix, box_sz, grid_sz, solution_found))
                    {
                        printMatrix(privateMatrix, box_sz);
                        #pragma omp critical
                        {
                        *solution_found = 1;
                        }
                    }
                    privateMatrix[row][col] = EMPTY;
                }
            }
            }
        }
    #pragma omp taskwait
```

```c
    int solution_found = 0;
#pragma omp parallel
    {
#pragma omp single
        {
            solveSudoku(0, 0, matrix, box_sz, grid_sz, &solution_found);
        }
    }
```

TESTS:

in seconds

4x4_Easy.csv	Thread Counts					
Version	1	2	4	8	16	32
Original Serial	8.698	7.802	10.098	8.275	8.476	9.47
Part A	10.951	9.807	9.405	16.734	24.28	36.662
Part B	8.957	7.274	3.74	3.444	2.717	3.351
Part C	5.471	4.826	1.006	2.624	5.42	1.906
Serial Single Solution	0.136	0.147	0.137	0.149	0.117	0.109
	SPEED-UPS					
Speedup-Part A	0.794265	0.795554196	1.073684211	0.494502211	0.349093904	0.258305603
Speedup-Part B	1.029777	0.932325045	0.37037037	0.416193353	0.320552147	0.353854277
Speedup-Part C	0.024858	0.030460008	0.136182903	0.056783537	0.021586716	0.057187828

4x4_Hard1.csv	Thread Counts					
Version	1	2	4	8	16	32
Original Serial	72.932	81.36	73.069	73.042	69.677	84.281
Part A	105.385	117.879	66.895	60.312	174.43	294.933
Part B	81.21	75.876	49.66	39.018	35.286	36.251
Part C	24.732	26.143	2.865	8.161	29.252	54.598
Serial Single Solution	0.67	0.652	0.583	0.766	0.714	0.712
	SPEED-UPS					
Speedup-Part A	0.692053	0.690199272	1.092293893	1.211069107	0.399455369	0.285763207
Speedup-Part B	0.898067	1.072275818	1.471385421	1.872007791	1.974635833	2.324928967
Speedup-Part C	0.02709	0.024939754	0.203490401	0.093861046	0.024408587	0.013040771

4x4_Hard2.csv	Thread Counts					
Version	1	2	4	8	16	32
Original Serial	152.203	133.028	130.146	129.547	125.099	125.863
Part A	153.466	218.207	109.855	125.251	357.344	535.408
Part B	150.739	141.283	86.379	73.709	79.026	79.951
Part C	8.063	1.153	2.115	8.851	22.22	29.754
Serial Single Solution	1.173	1.128	1.254	1.133	1.19	1.148
	SPEED-UPS					
Speedup-Part A	0.99177	0.609641304	1.184707114	1.034299127	0.350080035	0.235078669
Speedup-Part B	1.009712	0.941571173	1.506685653	1.757546568	1.583010655	1.574251729
Speedup-Part C	0.145479	0.978317433	0.592907801	0.128008135	0.053555356	0.038583048

4x4_Hard3.csv	Thread Counts					
Version	1	2	4	8	16	32
Original Serial	142.245	142.185	139.447	141.234	165.161	168.251
Part A	168.78	233.208	117.826	190.622	330.499	524.037
Part B	192.481	156.727	93.7	86.326	74.24	89.715
Part C	9.844	1.365	1.97	5.476	28.052	10.794
Serial Single Solution	1.195	1.365	1.273	1.371	1.252	1.274
SPEED-UPS						
Speedup-Part A	0.842784	0.609691777	1.183499397	0.740911332	0.499732223	0.321067024
Speedup-Part B	0.739008	0.907214456	1.488228388	1.636054028	2.224690194	1.875394304
Speedup-Part C	0.121394	1	0.646192893	0.25036523	0.044631399	0.118028534

When we compare, in the overall the best speed-up is obtained with Part b where we have a cutoff. This makes sence since without a cutoff parameter, the parallel implementation might generate a large number of tasks, leading to an excessive overhead in task creation, management, and synchronization. For 1 single solution found and returned case, the serial version works faster than the parallel version (part c). Implementing early termination in a parallel version can introduce additional complexities. Coordinating the termination of multiple threads when one of them finds a solution may lead to additional overhead, potentially negating the advantages of parallelization. Also, increasing the thread count does not always mean a better speedup as it can be seen in 4x4_hard3 Speedup-Part B, the speedup decreased for 32 threads.