ECE565 HW4 Report

1. Histogram

1.1 Strategies description:

locks:

```
  omp_lock_t lock_array [256];
  for(int i = 0; i < 256; ++i){
    omp_init_lock(&lock_array[i]);
  }

  t_start = omp_get_wtime();
  /* obtain histogram from image, repeated 100 times */
  for (m=0; m<100; m++) {
#pragma omp parallel for collapse(2) default(shared) private(i, j)
    for (i=0; i<image->row; i++) {
      for (j=0; j<image->col; j++) {
        omp_set_lock(&lock_array[image->content[i][j]]);
        histo[image->content[i][j]]++;
        omp_unset_lock(&lock_array[image->content[i][j]]);
      }
    }
  }
```

atomic:

```
  t_start = omp_get_wtime();
  /* obtain histogram from image, repeated 100 times */
  for (m=0; m<100; m++) {
#pragma omp parallel for collapse(2) default(shared) private(i, j)
    for (i=0; i<image->row; i++) {
      for (j=0; j<image->col; j++) {
        #pragma omp atomic update
        histo[image->content[i][j]]++;
      }
    }
  }
```

creative:

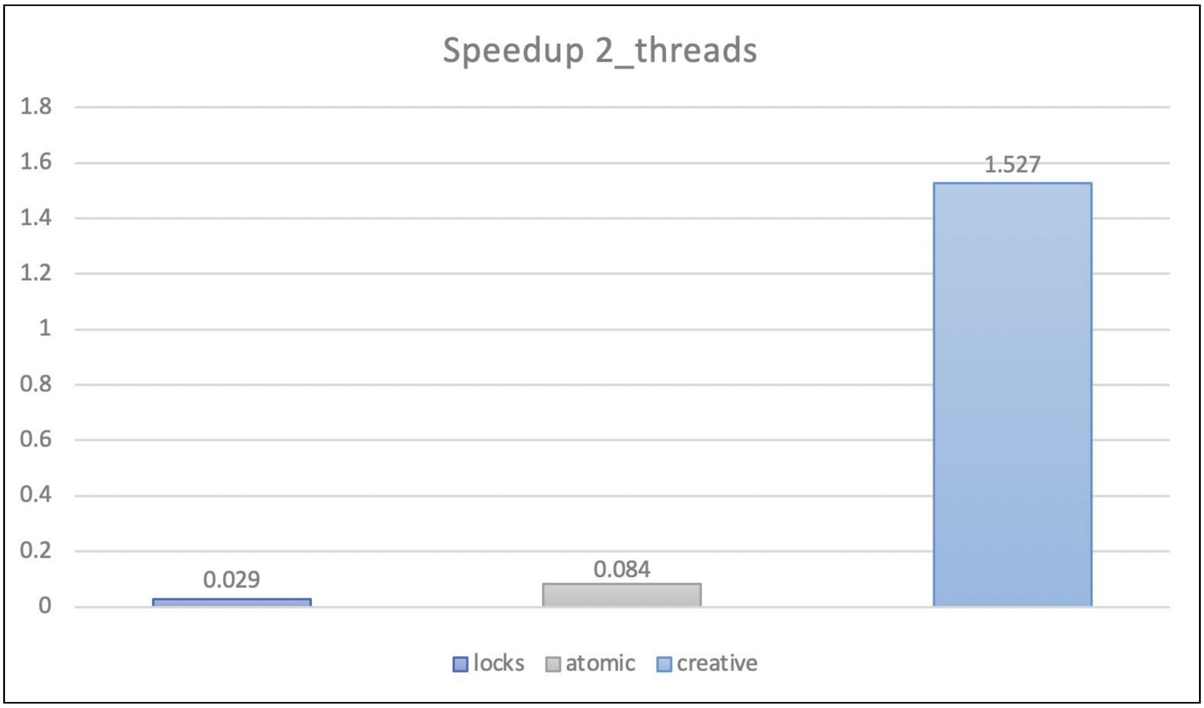
```
  t_start = omp_get_wtime();
  /* obtain histogram from image, repeated 100 times */
  for (m=0; m<100; m++) {
#pragma omp parallel for collapse(2) default(shared) private(i, j) reduction(+:histo[:256])
    for (i=0; i<image->row; i++) {
      for (j=0; j<image->col; j++) {
        histo[image->content[i][j]]++;
      }
    }
  }
```

1.2 Performance and speedup curve:

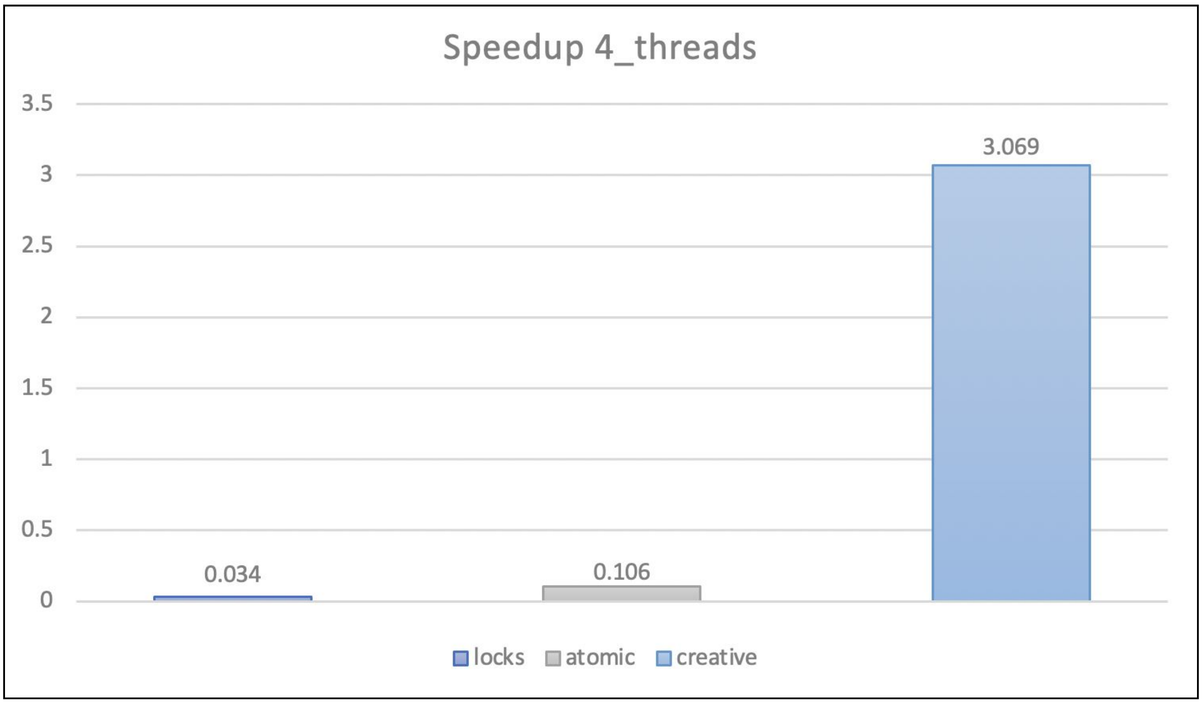
performance:

Thread	locks	atomic	creative	sequential
2	209.10s	73.74s	4.06s	6.20s
4	182.52s	58.61s	2.03s	6.23s
8	185.81s	49.61s	1.22s	6.27s

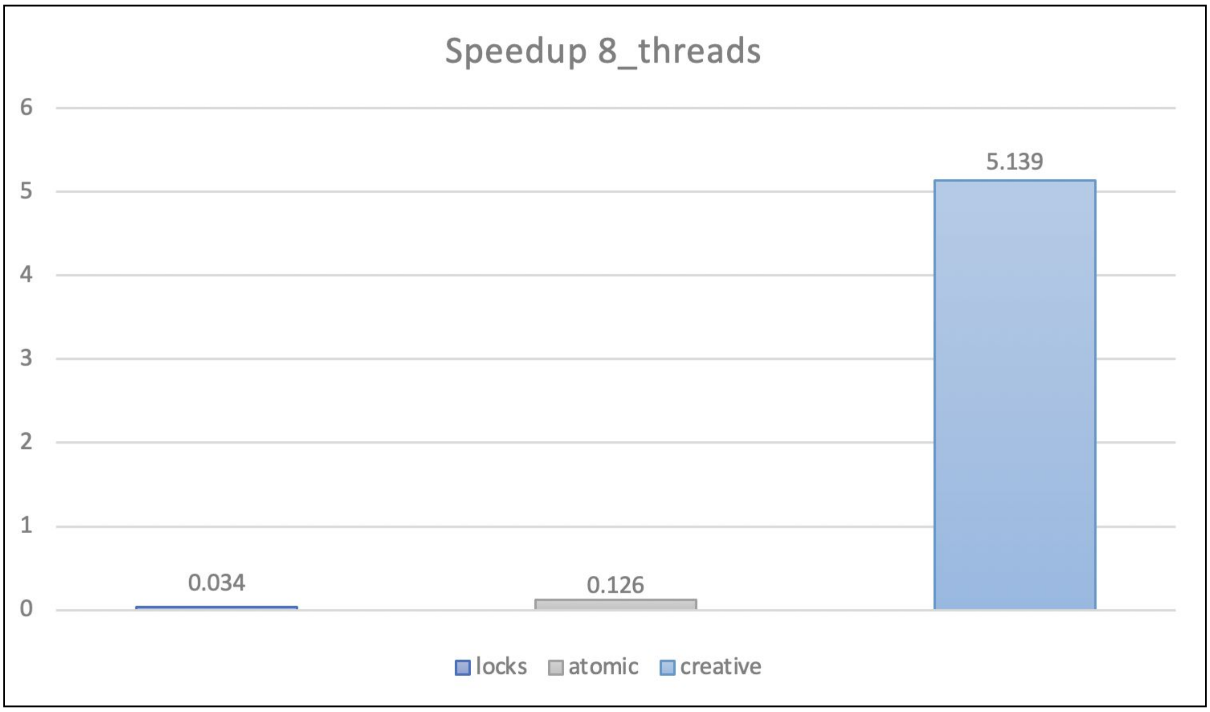
2 threads speed up:



4 threads speed up:



8 threads speed up:



1.3 Observation & Analysis

We can see from the 1.2 that our creative solution has the best performance. However, the execution time of both locks and atomic is much longer than the sequential code.

For lock solution, I think it’s because if multiple threads are accessing the array, they’ll have to wait for others to finish first, so it will waste a lot of time. Also each thread has to get a lock, set it before operation and unset it afterwards, the OS needs to do context switching between threads, which can cause a lot of overhead. We can see that with more threads, the performance doesn’t get improved. It’s because even if we have more threads, they still all have to wait for each other to finish and unlock the array, so the number of threads won’t impact performance that much.

For the atomic solution, it is achieved by hardware instead of OS and it’s much faster than locks. For different threads, instead of blocking each other, they are busy waiting, so they can execute without needing the OS to do context switching, so it avoids a lot of overhead. However, it still cannot achieve fully parallel because of busy waiting, so it’s slower than sequential code.

For the creative solution, I’m using the reduction directive. This will allow all the different threads to have their own copy of the reduction variable and then combine these local results at the end of the loop. This solution doesn't have many overheads and different threads can work in parallel without affecting each other. So the performance of this solution is much better than the sequential code. With more threads, the performance gets improved significantly.

2. AMG

1) A clear description of what changes you make to the code (e.g. describe your OpenMP directives and any code changes you make). You can refer to source files, line numbers, etc. and show code snippets.

Step1: Use Gprof to check the flat profile and find which part of the code is the most important

Add the following to the makefile LDFLAGS to enable gprof and also link the timer library

```
CFLAGS   = -O3 -fopenmp -c

LDFLAGS  = -O3 -fopenmp -lm -L. -ltimer -pg
```

The following is the flat profile of gprof of the **baseline sequential code :**

amgmk > analysis.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 56.13      1.56     1.56                             hypre_BoomerAMGSeqRelax
 40.30      2.68     1.12                             hypre_CSRMatrixMatvec
  2.88      2.76     0.08                             hypre_SeqVectorAxpy
  0.72      2.78     0.02                             GenerateSeqLaplacian
```

We can see that the most important function is **hypre_BoomerAMGSeqRelax** which is 56.13% of the total running time. And the second important one is **hypre_CSRMatrixMatvec**, we might want to do the parallel stuff mainly in these two.

Step2: Change the code to make it faster

1. Change the code inside **relax.c** to make **hypre_BoomerAMGSeqRelax** faster :

```c
70   #pragma omp parallel for default(shared) private(i, jj, ii) reduction(- \
71                                                                         : res)
72     for (i = 0; i < n; i++) /* interior points first */
73     {
74 
75       /*-----------------------------------------------------------
76        * If diagonal is nonzero, relax point i; otherwise, skip it.
77        *-----------------------------------------------------------*/
78       if (A_diag_data[A_diag_i[i]] != 0.0)
79       {
80         res = f_data[i];
81         for (jj = A_diag_i[i] + 1; jj < A_diag_i[i + 1]; jj++)
82         {
83           ii = A_diag_j[jj];
84           res -= A_diag_data[jj] * u_data[ii];
85         }
86         u_data[i] = res / A_diag_data[A_diag_i[i]];
87       }
88     }
89 
90     return (relax_error);
91   }
92
```

On line 70 here I add the parallel for and set i jj ii as private as they have write conflict and set res as reduction as it is updating itself every loop

2.Change the code inside **csr_matvec.c** to make **hypre_CSRMatrixMatvec** faster :

```c
103   #pragma omp parallel for default(shared) private(i)
104         for (i = 0; i < num_rows * num_vectors; i++)
105            y_data[i] *= beta;
106 
107         return ierr;
108      }
109 
```

On line 103 I add the line to set the parallel for and set i as private

```
119         {
120 #pragma omp parallel for default(shared) private(i)
121             for (i = 0; i < num_rows * num_vectors; i++)
122                 y_data[i] = 0.0;
123         }
124         else
125         {
126 #pragma omp parallel for default(shared) private(i)
127             for (i = 0; i < num_rows * num_vectors; i++)
128                 y_data[i] *= temp;
129         }
130     }
131
```

On line 120 and 126 I add the line to set the parallel for and set i as private

```
140 #pragma omp parallel default(shared) private(i, m, jj, j) reduction(+ \
141                                                                     : tempx)
142         for (i = 0; i < num_rownnz; i++)
143         {
144             m = A_rownnz[i];
145
146             /*
147              * for (jj = A_i[m]; jj < A_i[m+1]; jj++)
148              * {
149              *         j = A_j[jj];
150              *  y_data[m] += A_data[jj] * x_data[j];
151              * } */
152             if (num_vectors == 1)
153             {
154                 tempx = y_data[m];
155                 for (jj = A_i[m]; jj < A_i[m + 1]; jj++)
156                     tempx += A_data[jj] * x_data[A_j[jj]];
157                 y_data[m] = tempx;
158             }
159             else
160                 for (j = 0; j < num_vectors; ++j)
161                 {
162                     tempx = y_data[j * vecstride_y + m * idxstride_y];
163                     for (jj = A_i[m]; jj < A_i[m + 1]; jj++)
164                         tempx += A_data[jj] * x_data[j * vecstride_x + A_j[jj] * idxstride_x]
165                     y_data[j * vecstride_y + m * idxstride_y] = tempx;
166                 }
```

On line 140 I add the line to set the parallel for and set i,m,jj,j as private, tempx as reduction

```
171 #pragma omp parallel for default(shared) private(i, jj, j) reduction(+ \
172                                                               : temp)
173         for (i = 0; i < num_rows; i++)
174         {
175             if (num_vectors == 1)
176             {
177                 temp = y_data[i];
178                 for (jj = A_i[i]; jj < A_i[i + 1]; jj++)
179                     temp += A_data[jj] * x_data[A_j[jj]];
180                 y_data[i] = temp;
181             }
182             else
183                 for (j = 0; j < num_vectors; ++j)
184                 {
185                     temp = y_data[j * vecstride_y + i * idxstride_y];
186                     for (jj = A_i[i]; jj < A_i[i + 1]; jj++)
187                     {
188                         temp += A_data[jj] * x_data[j * vecstride_x + A_j[jj] * idxstride_x];
189                     }
190                     y_data[j * vecstride_y + i * idxstride_y] = temp;
191                 }
192         }
193     }
```

On line 140 I add the line to set the parallel for and set i,jj,j as private, temp as reduction

```
199         if (alpha != 1.0)
200         {
201     #pragma omp parallel for default(shared) private(i)
202             for (i = 0; i < num_rows * num_vectors; i++)
203                 y_data[i] *= alpha;
204         }
205 
206         return ierr;
207     }
208 
```

On line 201 I add the line to set the parallel for and set i as private

3.Change the code inside **vector.c** to make **hypre_SeqVectorAxpy** faster :

```
369 #pragma omp parallel for default(shared) private(i)
370     for (i = 0; i < size; i++)
371         y_data[i] += alpha * x_data[i];
372 
373     return ierr;
374 }
```

On line 369 I add the line to set the parallel for and set i as private

4.Change the code inside **laplace.c** to make **GenerateSeqLaplacian** faster :

```c
68   #pragma omp parallel for default(shared) private(i)
69       for (i = 0; i < grid_size; i++)
70       {
71           x_data[i] = 0.0;
72           sol_data[i] = 0.0;
73           rhs_data[i] = 1.0;
74       }
75
```

On line 68 I add the line to set the parallel for and set i as private

```c
161  #pragma omp parallel for default(shared) private(i, j)
162      for (i = 0; i < grid_size; i++)
163          for (j = A_i[i]; j < A_i[i + 1]; j++)
164              sol_data[i] += A_data[j];
165
```

On line 161 I add the line to set the parallel for and set i, j as private

2) You should summarize your baseline sequential vs. optimized parallel performance across 1, 2, 4, and 8 threads.

Baseline sequential code performance

```
yz558@vcm-17140:~/ECE565/ECE565_Parallel/hw4/amgmk$ ./AMGMk

//------------
//
//  AMGmk Benchmark
//
//------------

max_num_threads = 8


 testIter   = 1000


//------------
//
//   MATVEC
//
//------------

Wall time = 1.138619 seconds.

//------------
//
//   Relax
//
//------------

Wall time = 1.561802 seconds.

//------------
//
//   Axpy
//
//------------

Wall time = 0.077469 seconds.

Total Wall time = 2.802154 seconds.
```

Optimized parallel performance across 1 thread

```
yz558@vcm-17140:~/ECE565/ECE565_Parallel/hw4/amgmk$ OMP_NUM_THREADS=1 ./AMGMk

//------------
//
//  AMGmk Benchmark
//
//------------

max_num_threads = 1


 testIter   = 1000


//------------
//
//   MATVEC
//
//------------

Wall time = 1.091131 seconds.

//------------
//
//   Relax
//
//------------

Wall time = 1.573901 seconds.

//------------
//
//   Axpy
//
//------------

Wall time = 0.080909 seconds.

Total Wall time = 2.764992 seconds.
```

Optimized parallel performance across 2 threads

```
yz558@vcm-17140:~/ECE565/ECE565_Parallel/hw4/amgmk$ OMP_NUM_THREADS=2 ./AMGMk

//------------
//
//  AMGmk Benchmark
//
//------------

max_num_threads = 2


 testIter   = 1000


//------------
//
//   MATVEC
//
//------------

Wall time = 0.534737 seconds.

//------------
//
//   Relax
//
//------------

Wall time = 0.799485 seconds.

//------------
//
//   Axpy
//
//------------

Wall time = 0.038395 seconds.

Total Wall time = 1.391713 seconds.
```

Optimized parallel performance across 4 threads

```
yz558@vcm-17140:~/ECE565/ECE565_Parallel/hw4/amgmk$ OMP_NUM_THREADS=4 ./AMGMk

//------------
//
//  AMGmk Benchmark
//
//------------

max_num_threads = 4


 testIter   = 1000


//------------
//
//   MATVEC
//
//------------

Wall time = 0.313480 seconds.

//------------
//
//   Relax
//
//------------

Wall time = 0.402439 seconds.

//------------
//
//   Axpy
//
//------------

Wall time = 0.014167 seconds.

Total Wall time = 0.749260 seconds.
```

Optimized parallel performance across 8 threads

```
yz558@vcm-17140:~/ECE565/ECE565_Parallel/hw4/amgmk$ OMP_NUM_THREADS=8 ./AMGMk

//------------
//
//  AMGmk Benchmark
//
//------------

max_num_threads = 8


 testIter   = 1000


//------------
//
//   MATVEC
//
//------------

Wall time = 0.151678 seconds.

//------------
//
//   Relax
//
//------------

Wall time = 0.207808 seconds.

//------------
//
//   Axpy
//
//------------

Wall time = 0.009652 seconds.

Total Wall time = 0.384048 seconds.
```

Time Summary:

Number of threads/Sequential	Matvec Time (Seconds)	Relax Time (Seconds)	Axpy Time (Seconds)	Total Wall time(Seconds)
Sequential	1.138619	1.561802	0.077469	2.802154
1	1.091131	1.573901	0.080909	2.764992
2	0.534737	0.799485	0.038395	1.391713
4	0.313480	0.402439	0.014167	0.749260
8	0.151678	0.207808	0.009652	0.384048

How many times of speed-up compared to sequential:

Number of threads/Sequential	Matvec Time (Seconds)	Relax Time (Seconds)	Axpy Time (Seconds)	Total Wall time(Seconds)
Sequential	1x	1x	1x	2.802154
1	1.04x	0.99x	0.95x	1.01x
2	2.13x	1.95x	2.23x	2.01x
4	3.64x	3.89x	5.5x	3.74x
8	7.49x	7.51x	8.02x	7.30x

According to the above results, the performance improves as the number of threads increases. And the total wall time has a proportional reduction and the number of threads increases.