Mingxiang Dai (md382), Jiawei Sun (js896)

ECE 565 - Fall 2020 Assignment # 4

Problem 1: Histogram

1. Original

```
  for (m = 0; m < 100; m++) {
    for (i = 0; i < image->row; i++) {
      for (j = 0; j < image->col; j++) {
        histo[image->content[i][j]]++;
      }
    }
  }
```

Note: In the original code, there isn't an initialization for array, I add one.

2. Array of locks

1. Set an array of locks

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

2. Lock each element

```
 for (m = 0; m < 100; m++) {
#pragma omp parallel for private(i, j) collapse(2)
    for (i = 0; i < image->row; i++) {
      for (j = 0; j < image->col; j++) {
        omp_set_lock(&locks[image->content[i][j]]);
        histo[image->content[i][j]]++;
        omp_unset_lock(&locks[image->content[i][j]]);
      }
    }
  }
```

3. Atomic operation

```
  for (m = 0; m < 100; m++) {
#pragma omp parallel for private(i, j) collapse(2)
    for (i = 0; i < image->row; i++) {
      for (j = 0; j < image->col; j++) {
#pragma omp atomic update
        histo[image->content[i][j]]++;
      }
    }
  }
```

4. Our creative solution

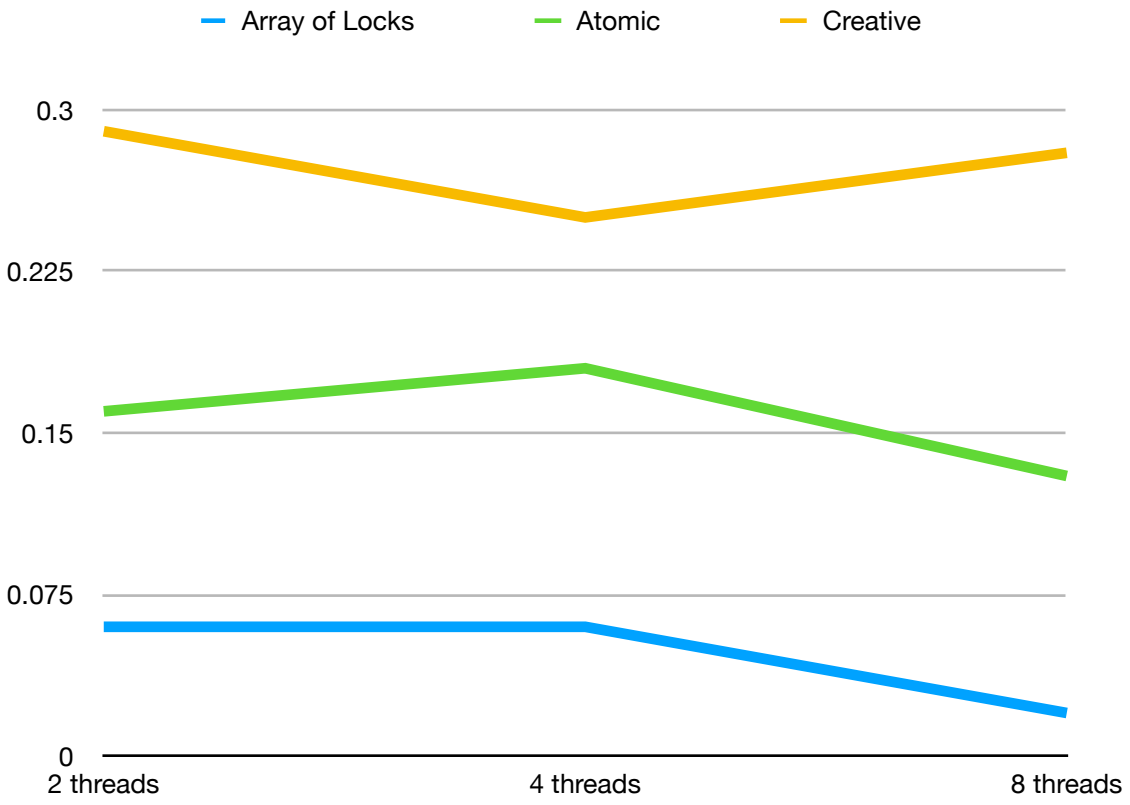
```
for (m = 0; m < 100; m++) {
    for (i = 0; i < image->row; i++) {
#pragma omp parallel for
      for (j = 0; j < image->col; j++) {
        results_of_each_thread[image->content[i][j]]
[omp_get_thread_num()]++;
      }
    }
  }

#pragma omp parallel for private(j)
  for (i = 0; i < 256; i++) {
    for (j = 0; j < threads_count; j++) {
      histo[i] += results_of_each_thread[i][j];
    }
  }
```

5. Results

Record

Number of Thread	Sequential	Array of Locks		Atomic		Creative	
	runtime	run time	speedup	run time	speedup	run time	speedup
2	8.88	156.22	0.06	54.28	0.16	30.34	0.29
4	8.88	160.06	0.06	50.07	0.18	35.82	0.25
8	8.88	364.72	0.02	68.27	0.13	31.96	0.28



Problem 2: AMG

1. Modification

By compiling the code with -pg option, we are able to locate which functions takes up the longest time when running the program. And here is the result:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 57.95      1.57     1.57     1000     1.57     1.57  hypre_BoomerAMGSeqRelax
 38.02      2.60     1.03     1000     1.03     1.03  hypre_CSRMatrixMatvec
  2.95      2.68     0.08     1000     0.08     0.08  hypre_SeqVectorAxpy
  1.11      2.71     0.03        2    15.00    15.00  GenerateSeqLaplacian
  0.00      2.71     0.00       26     0.00     0.00  hypre_CAlloc
  0.00      2.71     0.00       26     0.00     0.00  hypre_Free
  0.00      2.71     0.00        8     0.00     0.00  hypre_SeqVectorCreate
  0.00      2.71     0.00        7     0.00     0.00  hypre_SeqVectorDestroy
  0.00      2.71     0.00        5     0.00     0.00  hypre_SeqVectorSetConstantValues
  0.00      2.71     0.00        2     0.00     0.00  hypre_CSRMatrixCreate
  0.00      2.71     0.00        2     0.00     0.00  hypre_CSRMatrixDestroy
  0.00      2.71     0.00        2     0.00     0.00  hypre_SeqVectorInitialize
```

As it's shown, function *hypre_BoomerAMGSeqRelax, hypre_CSRMatrixMatvec and hypre_SeqVectorAxpy* took up the most time of execution and they are relevant to the 3 computations labeled "MATVEC", "Relax" and "Axpy" that we are concerned about. These 3 functions are in *relax.c*, *csr_matvec.c* and *vector.c* repectively.

(1) change1
in relax.c line 71

```
71    #pragma omp parallel for default(shared) private(i, jj)
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    }
```

(2) change2
In csr_matvec.c line 103

```
101     if (alpha == 0.0)
102     {
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     }
```

(3) change3
In csr_matvec.c line 120 and line 126

```
116     if (temp != 1.0)
117     {
118        if (temp == 0.0)
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     }
```

(4) change4
In csr_matvec.c line 140

```
140        #pragma omp parallel for default(shared) private(i)
141        for (i = 0; i < num_rownnz; i++)
```

(5) change5
In csr_matvec.c line 171

```
171        #pragma omp parallel for default(shared) private(i)
172        for (i = 0; i < num_rows; i++)
```

(6) change6
In csr_matvec.c line 200

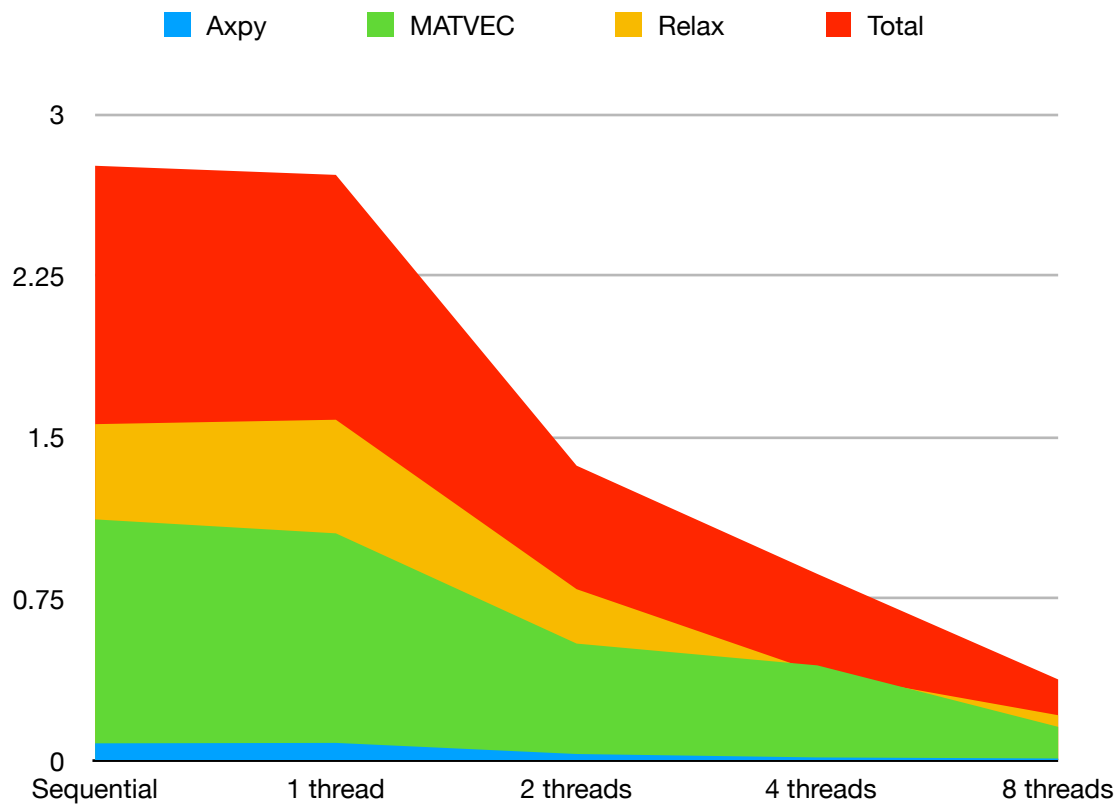
```
200     #pragma omp parallel for default(shared) private(i)
201     for (i = 0; i < num_rows * num_vectors; i++)
```

(7) change
In vector.c line 370

```
370 #pragma omp parallel for default(shared) private(i)
371   for (i = 0; i < size; i++)
```

2. Results

N of Threads	MATVEC	Relax	Axpy	Total
Sequential	1.119289	1.561335	0.079936	2.76056
1	1.054471	1.581843	0.081934	2.718248
2	0.542841	0.795361	0.030380	1.368582
4	0.441696	0.409798	0.014477	0.865971
8	0.156503	0.210473	0.009567	0.376543



Speedup:
Technically, speed of sequential and 1 thread should be the same.
2 threads: 2.76056/1.368582 = 2.02
4 threads: 2.76056/0.865971 = 3.19
8 threads: 2.76056/0.376543 = 7.33