Project 1

Zihao Wu, Kaiwen Wang
zw154, kw284

1. Histogram

> **Principles**
>
> - Use for construct for parallelism
> - Since the calculation time for each iteration is reasonably predictable, static scheduling is preferable than dynamic one which has higher overhead than static
> - Chunk size and number of threads is essentially a tradeoff between time overhead (fewer chunks, fewer overhead) and load balancing effects (more chunks, tasks are distributed more evenly); Don't forget creating threads and mapping tasks to threads have time overhead

0. Original code

The runtime of original code is as below:

- Small Image: 0.06s.
- Large Image: 6.02s

1.Array of locks

```
//e.g. for j iteration
for (m = 0; m < 100; m++) {
    for (i = 0; i < image->row; i++) {
#pragma omp parallel for private(j)
        for (j = 0; j < image->col; j++) {
          unsigned char ct = image->content[i][j];
          omp_set_lock(&locks[ct]);
          histo[ct]++;
      omp_unset_lock(&locks[ct]);
        }
    }
}
```

Default: # of threads = 8, run the program for 10 times and take average

```
#!/bin/bash
make clean
```

```
make
# set program and picture
PRG=./histo_locks
PIC=uiuc.pgm

sum=0
for i in {1..10}
do
    result=$($PRG $PIC | grep time | awk '{print $3}')
    echo round$i: $result seconds
    sum=$(echo $sum + $result | bc)
done
echo total amount of time for 10 times: $sum seconds
```

Benchmark result:

Type	Average time for small image	Average time for large image
for i	2.43	230.71
for j	3.41	342.26
Collapse	1.89	183.65

Observation1: It turns out that the array-of-locks version is about 35 times slower than the original code. The array-of-locks version is highly parallel, but the result is way more slower than we expected.

Analysis1: The reason we assume is that acquire & release of 256 locks will lead to a large overhead adding to the system that finally caused this result.

Observation2: Average time performance collapse > for i > for j

Analysis2: The j loop is slowest since the directive #pragma omp parallel is inside for i loop, resulting in (image->row) times threads creation which introduces higher overhead than the other two. The collapse wins over the for i loop as both (image->row) and (image->col) are not divisible by number of threads; therefore, benefits from using vectorization by compiler might be cancelled and extra loops to handle incomplete chunks are required.

Test on number of threads using collapse and smaller picture

# of threads	1	2	4	8
Average time	1.43	1.49	1.73	1.92

Observation1: performance get worse with increasing number of threads

Analysis1: Since most time is spent on acquire & release lock, parallelism in the increment operation has not much impact on performance. As number of threads increase, overhead on create & destroy treads and task mapping increases.

2.Atomic

```
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]]++;
        }
    }
}
```

Type	Average time for small image	Average time for large image
for i	0.74	71.43
for j	1.16	114.32
Collapse	0.87	86.60

Observation1: Overall the atomic version is about 10 times slower than the original code. The difference between for-i, for-j and collapse version is the same as above.

Analysis1: The atomic command makes it serialized when multiple threads want to execute the target line. So it is almost identical with the original serialized code. But, there will be overhead caused by executing atomic command. The cost of atomic is much less than locks.

Observation2: Collapse has worse performance than for i

Analysis2: since cost of atomic is much less than locks, increment operation plays a more important role in performance. To access (image->content[i][j]), the compiler needs use divide and mod operations to obtain i and j, and thus, collapse worsens performance.

3.Creative

```
/* obtain histogram from image, repeated 100 times */
for (m=0; m<100; m++) {
#pragma omp parallel
  {
    int i, j, local_histo[256];//local under "parallel" directive
    for (i = 0; i < 256; ++i) {
        local_histo[i] = 0;
    }
```

```
#pragma omp for nowait
    for (i = 0; i < image->row; i++) {
        //#pragma omp for private(j)
        for (j = 0; j < image->col; j++) {
            local_histo[image->content[i][j]]++;
        }
    }
#pragma omp critical
  {
      for (i = 0; i < 256; ++i) {
          histo[i] += local_histo[i];
      }
  }
    }
}
```

Description: The ideas comes from website. It is pretty similar to create reduction variables. To create private variables for every thread so that there will be no sychronization problem between all the threads. Every thread count the pixels and write the data into its own private histo array to achieve the highest level of parallelism. We use the `nowait` command to make all threads continue on their mission after the for loop. And we use a `critical` command to make sure there will not be a race condition when updating the global histo array.

# of threads = 8

Type	Average time for small image	Average time for large image
for i	0.034	0.74
for j	0.108	1.63
Collapse	0.042	1.04

Observation1: The performance is better the original code

Analysis1: Why synchronization here is better than the lock and atomic version? using local histo[] transforms synchronization on each histo[] element to synchronization on each thread; therefore, the number of synchronziation has been reduced greatly.

Observation2: performance for i > collapse > for j

Analysis2: similar to analysis 2 for atomic version

Test on number of threads using for i and small picture, run 100 times and take average

# of threads	1	2	4	8	16	32	128
Average time	0.055	0.031	0.023	0.037	0.025	0.037	0.089

Observation: There is an equilibrium point in performance against number of threads

Analysis: As synchronization is removed from each element in histo[], performance improvement should scale with number of threads. However, in the critical region, each thread needs to handle a loop in size of 256. More threads means more computation in critical region.

2. Amgmk

0. Profiling

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 56.33      1.56     1.56     1000     1.56     1.56  hypre_BoomerAMGSeqRelax
 40.08      2.67     1.11     1000     1.11     1.11  hypre_CSRMatrixMatvec
  2.89      2.75     0.08     1000     0.08     0.08  hypre_SeqVectorAxpy
  0.72      2.77     0.02        2    10.00    10.00  GenerateSeqLaplacian
  0.00      2.77     0.00       26     0.00     0.00  hypre_CAlloc
  0.00      2.77     0.00       26     0.00     0.00  hypre_Free
  0.00      2.77     0.00        8     0.00     0.00  hypre_SeqVectorCreate
  0.00      2.77     0.00        7     0.00     0.00  hypre_SeqVectorDestroy
  0.00      2.77     0.00        5     0.00     0.00  hypre_SeqVectorSetConstantValues
  0.00      2.77     0.00        2     0.00     0.00  hypre_CSRMatrixCreate
  0.00      2.77     0.00        2     0.00     0.00  hypre_CSRMatrixDestroy
  0.00      2.77     0.00        2     0.00     0.00  hypre_SeqVectorInitialize
```

Profiling Result of Original Code

From the profiling result, we noticed that the three functions that cost most of the runtime is `hypre_BoomerAMGSeqRelax`, `hypre_CSRMatrixMatvec` and `hypre_SeqVectorAxpy`, which are the functions of three computation method we need to optimize.

1. Matvec Optimization (hypre_CSRMatrixMatvec)

```
1  // original code from line 105 to line 111
2  if (alpha == 0.0) {
3  #pragma omp parallel for
4      for (i = 0; i < num_rows*num_vectors; i++)
5          y_data[i] *= beta;
6      return ierr;
7  }
```

```
// original code from line 119 to line 131
if (temp != 1.0) {
    if (temp == 0.0) {
#pragma omp parallel for
        for (i = 0; i < num_rows*num_vectors; i++)
            y_data[i] = 0.0;
    }
    else {
#pragma omp parallel for
        for (i = 0; i < num_rows*num_vectors; i++)
            y_data[i] *= temp;
    }
}
```

```
// original code from line 139 to line 191
if (num_rownnz < xpar*(num_rows)) {
    if ( num_vectors==1 ) {
#pragma omp parallel for private(jj, m, tempx)
        for (i = 0; i < num_rownnz; i++) {
            m = A_rownnz[i];
            tempx = y_data[m];
            for (jj = A_i[m]; jj < A_i[m+1]; jj++)
                tempx +=  A_data[jj] * x_data[A_j[jj]];
            y_data[m] = tempx;
        }
    }
    else {
#pragma omp parallel for private(j, tempx, jj)
        for (i = 0; i < num_rownnz; i++) {
            for (j = 0; j < num_vectors; ++j) {
                tempx = y_data[ j*vecstride_y + m*idxstride_y ];
                for (jj = A_i[m]; jj < A_i[m+1]; jj++)
                    tempx += A_data[jj] * x_data[j*vecstride_x +
A_j[jj]*idxstride_x];
                y_data[j*vecstride_y + m*idxstride_y] = tempx;
            }
        }
    }
}
else {
    if ( num_vectors==1 ) {
#pragma omp parallel for private(temp, jj)
        for (i = 0; i < num_rows; i++) {
            temp = y_data[i];
            for (jj = A_i[i]; jj < A_i[i+1]; jj++)
```

```
                temp += A_data[jj] * x_data[A_j[jj]];
            y_data[i] = temp;
        }
    }
    else {
#pragma omp parallel for private(j, temp, jj)
        for (i = 0; i < num_rows; i++) {
            for ( j=0; j<num_vectors; ++j ) {
                temp = y_data[ j*vecstride_y + i*idxstride_y ];
                for (jj = A_i[i]; jj < A_i[i+1]; jj++) {
                    temp += A_data[jj] * x_data[j*vecstride_x +
A_j[jj]*idxstride_x];
                }
                y_data[j*vecstride_y + i*idxstride_y] = temp;
            }
        }
    }
}
```

```
// original code from line 198 to line 202
if (alpha != 1.0) {
#pragma omp parallel for
    for (i = 0; i < num_rows*num_vectors; i++)
        y_data[i] *= alpha;
}
```

The optimized code is as above, with the correspoinding line number in the original code.

The change we made is basically adding parallel regions and for construct in OpenMP. In the third code block, i.e., the longest one, we used loop unswitching transofrmation to extract the if-else judgement out of the outer for loop to eliminate conditional branches from each loop iteration.

2. Relaxs Optimization (hypre_BoomerAMGSeqRelax)

```
// original code from line 76 to line 92
#pragma omp parallel for private(i,ii,jj,res)
for (i = 0; i < n - 1; i++)  /* interior points first */ {
    if ( A_diag_data[A_diag_i[i]] != 0.0) {
        res = f_data[i];
        for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++) {
            ii = A_diag_j[jj];
            res -= A_diag_data[jj] * u_data[ii];
        }
        u_data[i] = res / A_diag_data[A_diag_i[i]];
    }
}
```

We analyzed the data scoping in the parallel region and privatized the conflicting variables. Initially, we consider to set res as reduction variable as "res -= A_diag_data[jj] * u_data[ii];" satisfies requirement for reduction. However, res accesses f_data[i] in for i loop, already been set as private; it is no worth to move parallel region from for i to for jj in order to set res as reduction, since n is a big number, and creating n times team of threads cost a lot.

3. Axpy Optimization (hypre_CSRMatrixMatvec)

```
// original code from line 384 to line 385
#pragma omp parallel for private(i)
for (i = 0; i < size; i++) {
    y_data[i] += alpha * x_data[i];
}
```

What we did here is basically the same as above two optimizations.

4. Result after optimization

Runtime before Optimization

Original Code	MATVEC	Relax	Axpy	Total
Runtime(s)	1.159965	1.586160	0.078409	2.843816

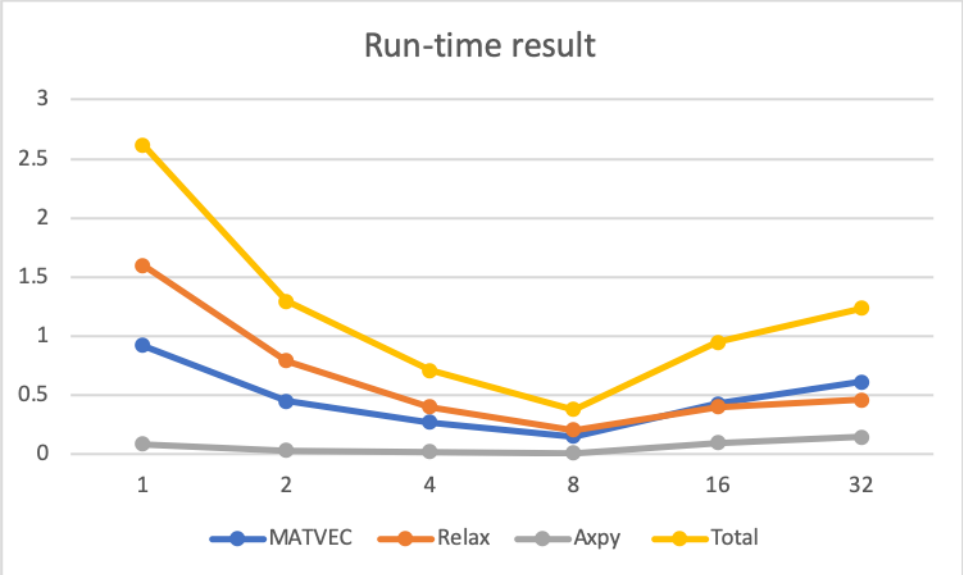
Runtime after Optimization

```bash
#!/bin/bash

make clean
make

for i in 1 2 4 8 16 32
do
    echo number of threads = $i
    echo top-down: MATVEC Relax Axpy Total
    OMP_NUM_THREADS=$i ./AMGMk | grep -oP "Wall time.*" | awk '{print $4}'
    echo
done
```

Threads	1	2	4	8	16	32
MATVEC	0.917717	0.450806	0.267296	0.150263	0.428789	0.608891
Relax	1.597241	0.792244	0.401375	0.203188	0.398143	0.459196
Axpy	0.084093	0.030416	0.018087	0.008932	0.09525	0.147828
Total	2.617274	1.291687	0.711396	0.380328	0.945692	1.233809



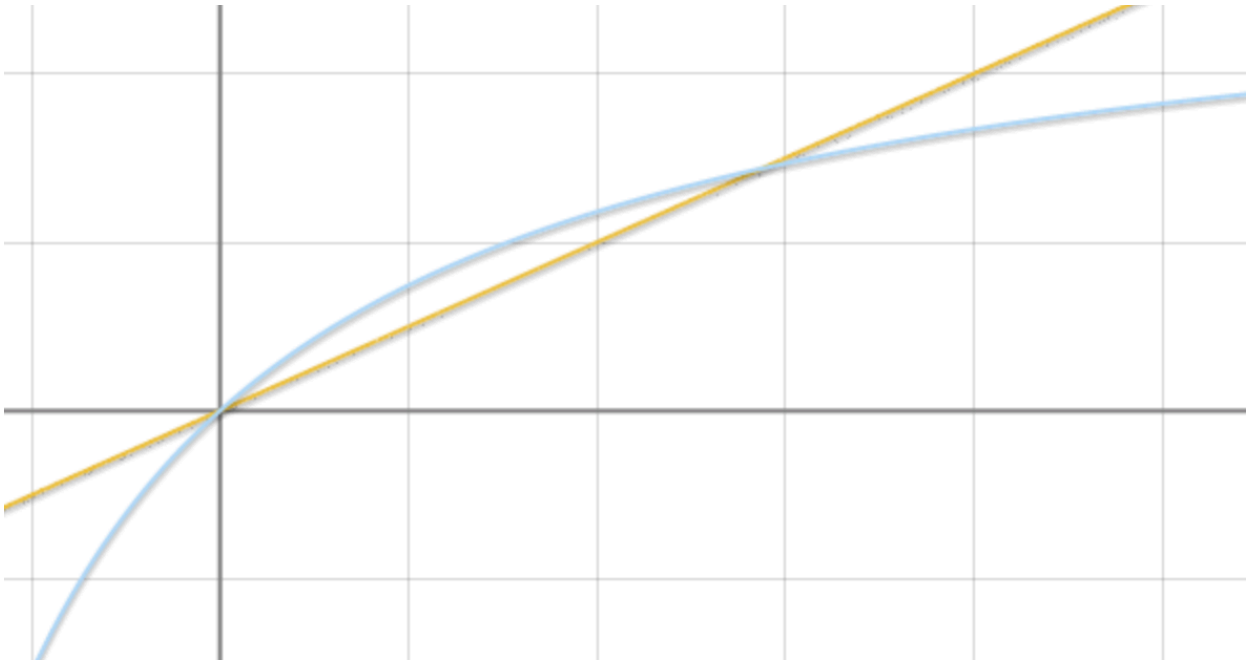
It is obvious that all optimizations have a better performance than original code.

- The three functions cost 99.3% of the total runtime.
- From Amdahl's Law, we can know that the speedup formula for the perfect performance is as below:

$$N = number\ of\ threads$$

$$speedup = \frac{1}{0.007 + \frac{0.993}{N}}$$

- The actual performance is influenced by thread creation and task distribution, which scales linearly with number of threads, unlike the theoretical logarithm speedup. The speedup will meet a limit when number of threads increases infinitely, however, the overhead cost by thread creation and task distribution will eventually exceed speedup.



Conclusion

From the result we noticed that when thread number is 8, we can have the best performance. For particular code, there's always a critical point where the program will reach its best balance between parallelism and thread overhead.

In the best performance situation, i.e., the 8-threads version, the total runtime is about 7.5 times faster than the original code.