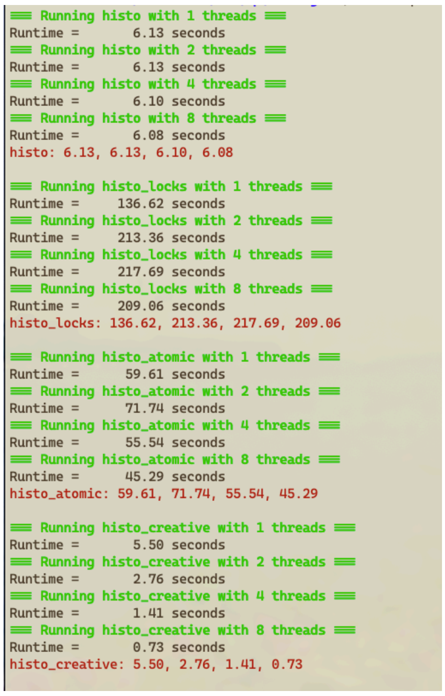
ECE 565 Homework 4

- Kewei Xia (kx30)
- Junqi Sun (js895)

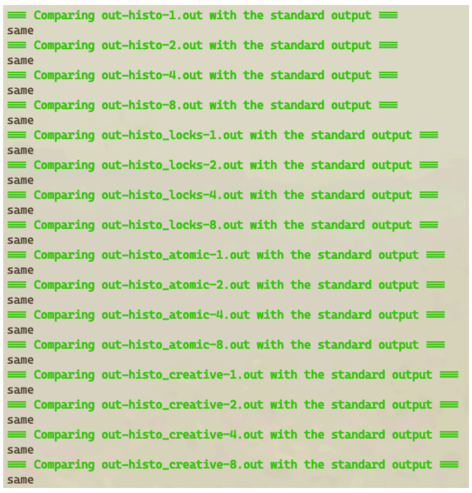
Q1 Histogram

Runtime & validation

I wrote a small script to run all executable with 1, 2, 4, 8 threads and compare the output with the validation one. Here is the result:



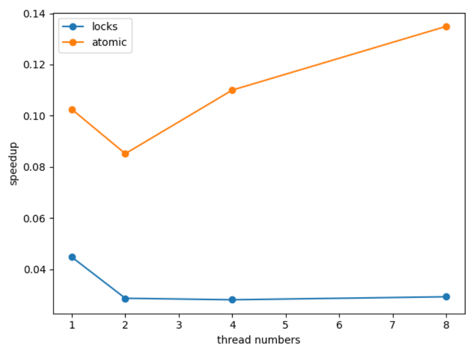
```
=== Running histo with 1 threads ===
Runtime =       6.13 seconds
=== Running histo with 2 threads ===
Runtime =       6.13 seconds
=== Running histo with 4 threads ===
Runtime =       6.10 seconds
=== Running histo with 8 threads ===
Runtime =       6.08 seconds
histo: 6.13, 6.13, 6.10, 6.08

=== Running histo_locks with 1 threads ===
Runtime =     136.62 seconds
=== Running histo_locks with 2 threads ===
Runtime =     213.36 seconds
=== Running histo_locks with 4 threads ===
Runtime =     217.69 seconds
=== Running histo_locks with 8 threads ===
Runtime =     209.06 seconds
histo_locks: 136.62, 213.36, 217.69, 209.06

=== Running histo_atomic with 1 threads ===
Runtime =      59.61 seconds
=== Running histo_atomic with 2 threads ===
Runtime =      71.74 seconds
=== Running histo_atomic with 4 threads ===
Runtime =      55.54 seconds
=== Running histo_atomic with 8 threads ===
Runtime =      45.29 seconds
histo_atomic: 59.61, 71.74, 55.54, 45.29

=== Running histo_creative with 1 threads ===
Runtime =       5.50 seconds
=== Running histo_creative with 2 threads ===
Runtime =       2.76 seconds
=== Running histo_creative with 4 threads ===
Runtime =       1.41 seconds
=== Running histo_creative with 8 threads ===
Runtime =       0.73 seconds
histo_creative: 5.50, 2.76, 1.41, 0.73
```

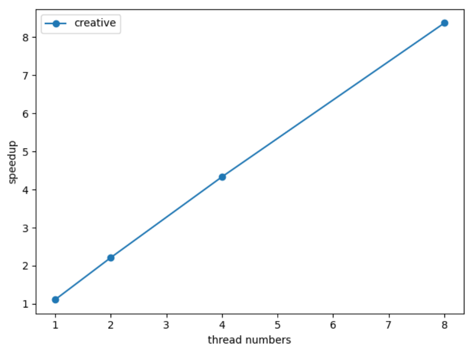


```
=== Comparing out-histo-1.out with the standard output ===
same
=== Comparing out-histo-2.out with the standard output ===
same
=== Comparing out-histo-4.out with the standard output ===
same
=== Comparing out-histo-8.out with the standard output ===
same
=== Comparing out-histo_locks-1.out with the standard output ===
same
=== Comparing out-histo_locks-2.out with the standard output ===
same
=== Comparing out-histo_locks-4.out with the standard output ===
same
=== Comparing out-histo_locks-8.out with the standard output ===
same
=== Comparing out-histo_atomic-1.out with the standard output ===
same
=== Comparing out-histo_atomic-2.out with the standard output ===
same
=== Comparing out-histo_atomic-4.out with the standard output ===
same
=== Comparing out-histo_atomic-8.out with the standard output ===
same
=== Comparing out-histo_creative-1.out with the standard output ===
same
=== Comparing out-histo_creative-2.out with the standard output ===
same
=== Comparing out-histo_creative-4.out with the standard output ===
same
=== Comparing out-histo_creative-8.out with the standard output ===
same
```

Speedup figure

From the result we can see that both the atomic and lock version is even slower than the sequential version. So I put those two in a separate figure with the creative way. Here is the speedup curve.





Result analysis

Paste the core code snippet here for reference.

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

From the figure and the runtime result, we can clearly see that the creative way has the best performance (the largest speedup). For both atomic and lock version, they run even slower than the sequential version, this is because the *synchronization is expensive*, it introduces a lot overhead to our program. For the original program, we only do one read and one update operations, which can be done in like 5 instructions, but to add the synchronization, we may need to add a lot more instructions especially for the lock version. This will degrade our programe performance.

Another reason of the atomic and locks version is worse than the sequential one is that the **cost of data movement**. As we can see in the code snippt, both the `histo` and `image` shoule be shared be all threads i.e. need to be move around different cores. Take `histo` as an example, it's a fixed length array contains 256 `Long` elements, which is relatively small to be fit in the L1 cache (at worst L2 cache) for a single thread, so for the sequential version, most of the access will hit in L1 cache. However, in multi-threading, a lot of the access will miss (the data need to be moved from another thread), and this may become even worse if the number of threads increase.

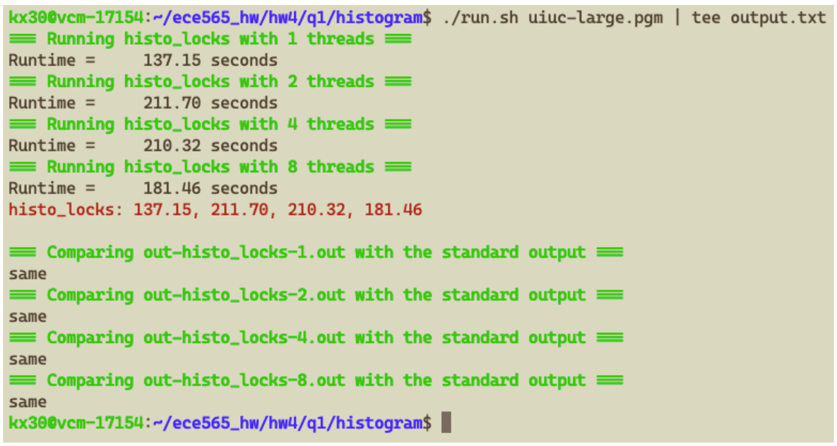
We can see from the figure, the locks version is worse than the atomic version, there are a few reasons causing this: 1) **data movement**, in lock version, we not only need to move `histo` and `image` around cores, we also need to move all locks around cores; that's also why with the number of threads increase, the performance of atomic version slightly increase but the locks version is alomost the same; 2) **overhead of lock**, for atomic, the hardware support some atomic operations, which will be faster; for lock, we need at least two more operations (lock & unlock) which might be corresponding to several instructions; 3) **lock contention**, we don't know anything about the input image, so this code `histo[image->content[i][j]]++;` may cause a lot lock contentions (e.g. nerighbour pixels have the same value).

From the above analysis, we know that we need the locks or atomic, because all threads will update the `histo` variable, also most of the overhead is introduced by the cost of data movement. To fix this, I propose my creative method that make the `histo` variable as a reduction variable, so each thread will keep its own copy, and there is no more race conditions, and no data movement of the `histo` variable. As we can see from the running script result and the figure, this method produce the correct output also has the largest speedup. I also noticed that the speedup of this version is linear to the thread number until 8 (i.e. 8x speedup for 8 threads), so I think if we can have a machine with more cores, we may get a even higher speedup.

More improvements we can make

From the analysis of last section, we know that the data movement is expensive, and we also prove this by making the `histo` variable firstprivate to gain a linear speedup. We can do the same thing to `image` variable (note this is also shared by all thread of reading), but it's a bit tricky here, as we said before, the `histo` is relatively small, so it's not that expensive to keep one copy each thread. For the `image`, it's specify by the user, so when it's small, this approach will not need too many extra space, but when it's large, we will take too many extra space. Also, if the `image` is large, even for single core, it may live in L2 or even L3 cache, which means that the difference between accessing it on single thread and across different threads is small.

I did a small experiment to make the `image` to firstprivate for the locks version, here is the result, it's a bit better than before.



```
kx30@vcm-17154:~/ece565_hw/hw4/q1/histogram$ ./run.sh uiuc-large.pgm | tee output.txt
== Running histo_locks with 1 threads ==
Runtime =      137.15 seconds
== Running histo_locks with 2 threads ==
Runtime =      211.70 seconds
== Running histo_locks with 4 threads ==
Runtime =      210.32 seconds
== Running histo_locks with 8 threads ==
Runtime =      181.46 seconds
histo_locks: 137.15, 211.70, 210.32, 181.46

== Comparing out-histo_locks-1.out with the standard output ==
same
== Comparing out-histo_locks-2.out with the standard output ==
same
== Comparing out-histo_locks-4.out with the standard output ==
same
== Comparing out-histo_locks-8.out with the standard output ==
same
kx30@vcm-17154:~/ece565_hw/hw4/q1/histogram$
```

Q2

Instruction

When there is exe file `AMGMk`, just run: `OMP_NUM_THREADS=num_threads ./AMGMk`, The num_threads should be range from 1, 2, 4, 8.

If there is no exe file, first `make`, then execute it as above.

Changes of Code

First I added some elements to Makefile:

```
CC        = gcc
LDR       = gcc

CFLAGS    = -O3 -fopenmp -c

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

LIBS      =
LIB_DIRS =

PROG      = AMGMk

OBJS      = main.o \
            csr_matrix.o   csr_matvec.o   \
            laplace.o relax.o \
            hypre_error.o hypre_memory.o \
            vector.o
```

According to gprof, I located the functions to optimize:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 56.08      1.57     1.57                             hypre_BoomerAMGSeqRelax
 40.37      2.70     1.13                             hypre_CSRMatrixMatvec
  2.86      2.78     0.08                             hypre_SeqVectorAxpy
  0.71      2.80     0.02                             GenerateSeqLaplacian
```

So we should focus on optimizing the four functions above, especially the first two.

`csr_matvec.c`

In file `csr_matvec.c`, line 144 and line 167, I added `#pragma omp parallel for default(shared) private(i,j,jj,tempx)` (for line 167 is `temp`) to optimize the nested for loops for function `hypre_CSRMatrixMatvec()` function. I also added similar sentences in line 288, 324, 439 to optimize other function in this file.

At the same time, I also use Loop Unswitching to further optimize the loop. e.g.

```
if (num_vectors == 1) {
#pragma omp parallel for default(shared) private(i, j, jj, temp)  // 2
  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 default(shared) private(i, j, jj, temp)  // 2
 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;
  }
 }
}
```

`relax.c`

In file `relax.c`, I added `#pragma omp parallel for default(shared) private(i, jj, ii, res)` in line 69 to optimize function `hypre_BoomerAMGSeqRelax()`.

```
#pragma omp parallel for default(shared) private(i, jj, ii, res)
for (i = 0; i < n; i++) /* interior points first */
{
```

`vector.c`

In file `vector.c`, I added `#pragma omp parallel for default(shared) private(i)` in line 334 to optimize function `hypre_SeqVectorAxpy()`.

```
#pragma omp parallel for default(shared) private(i) firstprivate(size, alpha)
for (i = 0; i < size; i++) {
    y_data[i] += alpha * x_data[i];
}
```

other

In file `laplace.c`, there exists loop carried dependency so it can’t be paralleled.

Performance

Sequential: 2.7954 seconds

Optimized Parallel(since we have 8-cores machine, max threads here should be 8):

# threads	Total wall time/s
1	2.5919
2	1.3262
4	0.7203
8	0.4312