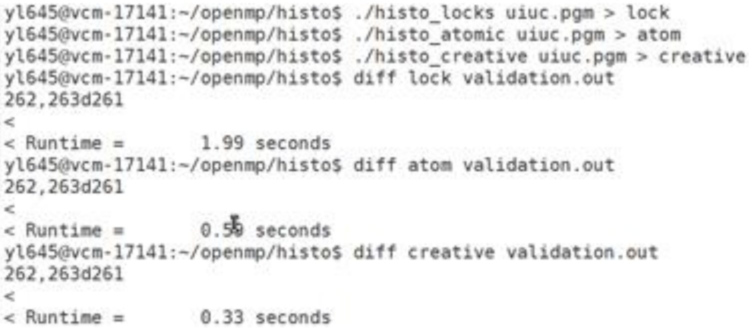
Homework4 Report

Yuanrui Zhang (yz545) & Yuchuan Li (yl645)

Problem 1: Histogram

Correctness:

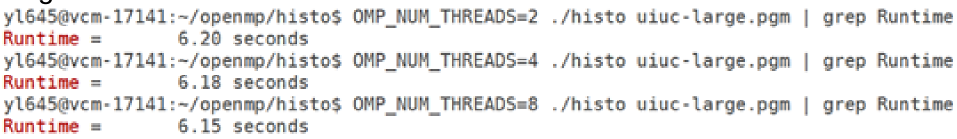


```
yl645@vcm-17141:~/openmp/histo$ ./histo_locks uiuc.pgm > lock
yl645@vcm-17141:~/openmp/histo$ ./histo_atomic uiuc.pgm > atom
yl645@vcm-17141:~/openmp/histo$ ./histo_creative uiuc.pgm > creative
yl645@vcm-17141:~/openmp/histo$ diff lock validation.out
262,263d261
<
< Runtime =        1.99 seconds
yl645@vcm-17141:~/openmp/histo$ diff atom validation.out
262,263d261
<
< Runtime =        0.59 seconds
yl645@vcm-17141:~/openmp/histo$ diff creative validation.out
262,263d261
<
< Runtime =        0.33 seconds
```

The only difference between my results and the validation.out file is the runtime part, not the result of histogram part, so these three versions are all correct.

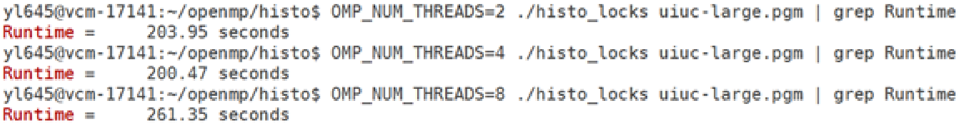
Performance:

Original Version:

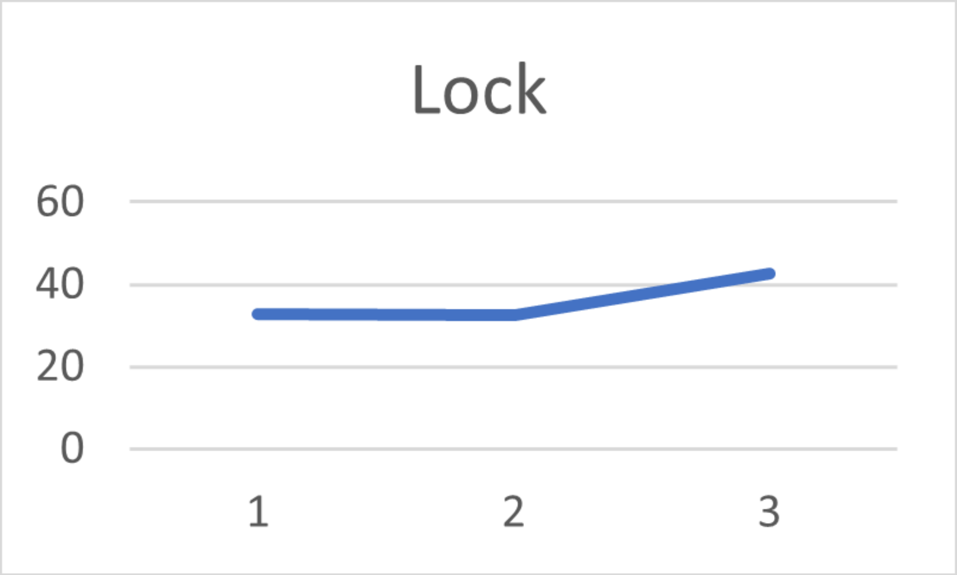


```
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=2 ./histo uiuc-large.pgm | grep Runtime
Runtime =       6.20 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=4 ./histo uiuc-large.pgm | grep Runtime
Runtime =       6.18 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=8 ./histo uiuc-large.pgm | grep Runtime
Runtime =       6.15 seconds
```

Lock Version:

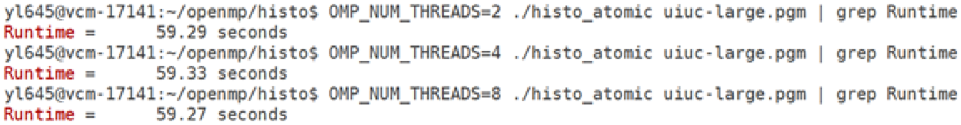


```
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=2 ./histo_locks uiuc-large.pgm | grep Runtime
Runtime =     203.95 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=4 ./histo_locks uiuc-large.pgm | grep Runtime
Runtime =     200.47 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=8 ./histo_locks uiuc-large.pgm | grep Runtime
Runtime =     261.35 seconds
```

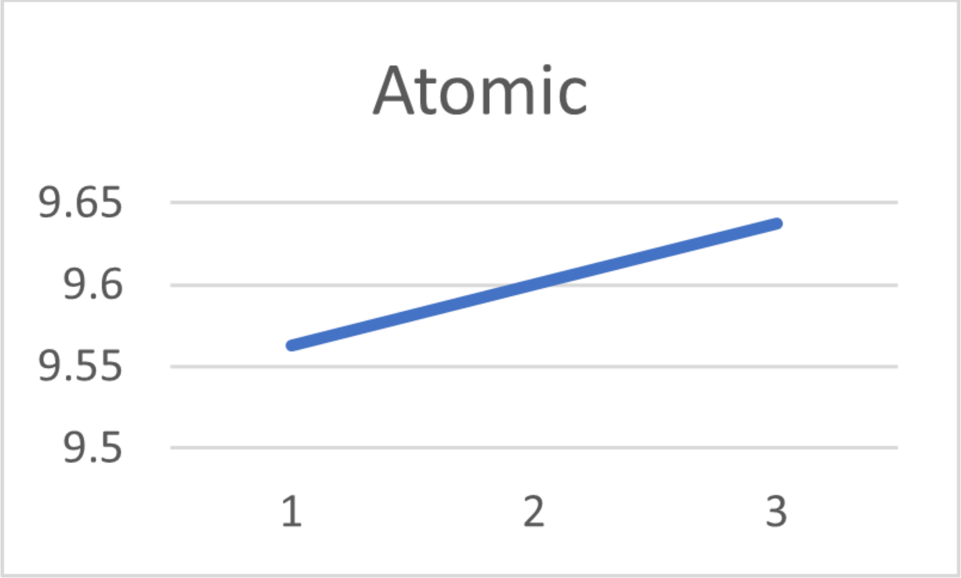


*1 2 3 represents 2, 4, 8 processes

Atomic Version:

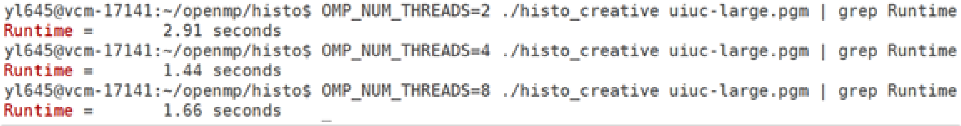


```
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=2 ./histo_atomic uiuc-large.pgm | grep Runtime
Runtime =      59.29 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=4 ./histo_atomic uiuc-large.pgm | grep Runtime
Runtime =      59.33 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=8 ./histo_atomic uiuc-large.pgm | grep Runtime
Runtime =      59.27 seconds
```

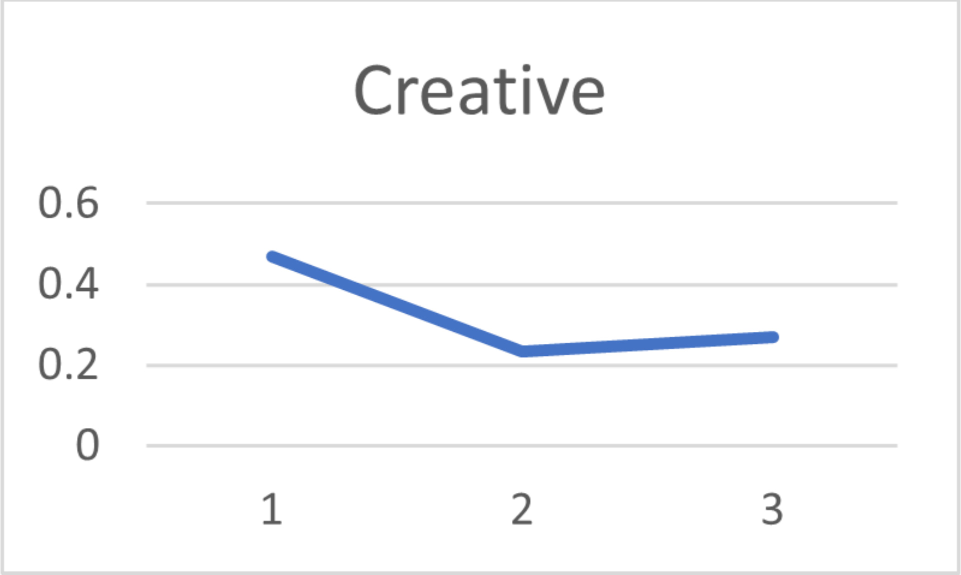


*1 2 3 represents 2, 4, 8 processes

Creative Version:



```
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=2 ./histo_creative uiuc-large.pgm | grep Runtime
Runtime =       2.91 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=4 ./histo_creative uiuc-large.pgm | grep Runtime
Runtime =       1.44 seconds
yl645@vcm-17141:~/openmp/histo$ OMP_NUM_THREADS=8 ./histo_creative uiuc-large.pgm | grep Runtime
Runtime =       1.66 seconds
```



*1 2 3 represents 2, 4, 8 processes

Creative Version Demonstration:

The histo[x] is being updated all the time for the same x, so there will be race condition when executing parallelly. We can use the reduction for the array histo[] to update the value of histo[x] concurrently and sum them up in the end to get correct values for the whole array. To be more specific, this method will create a private version per process and update the private version concurrently.

Analysis:
Speed comparison: creative < original < atomic < lock
The lock version is significantly slower than other versions, this might be due to the cost of communication and the overhead of using locks. Even though I used an array of locks, which means that the granularity of locks is smaller, it still took much longer time than others.
There is not much speed-up for the lock version and atomic version, even worse. But the creative version has a boost when the number of processes increase.

Problem 2: amgmk

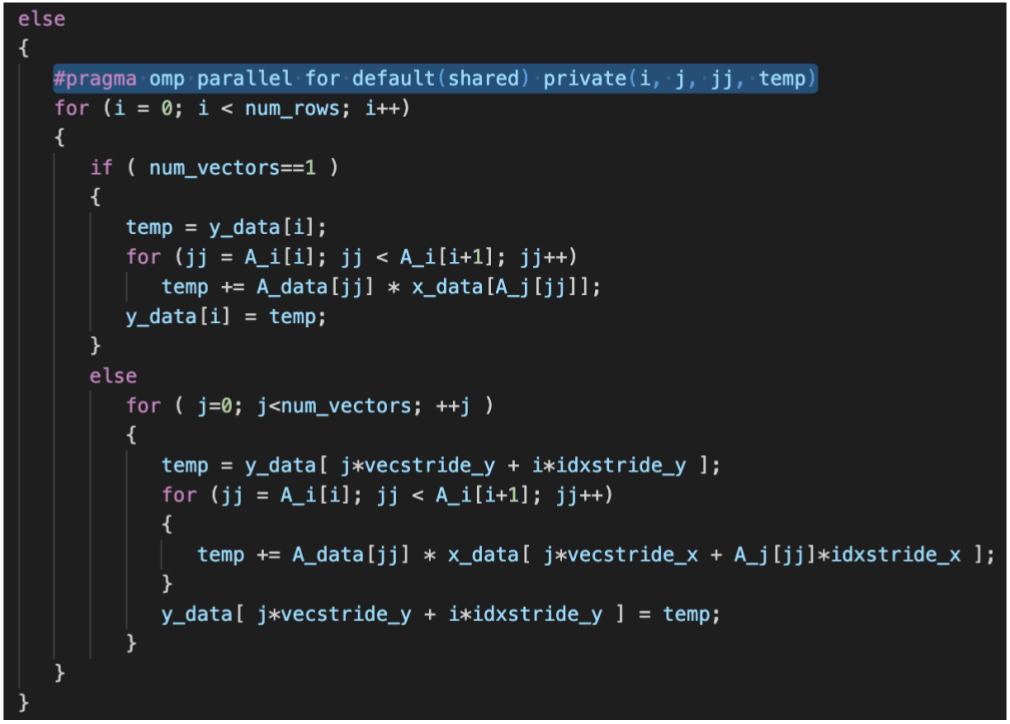
1 Code changes

(1) MATVEC

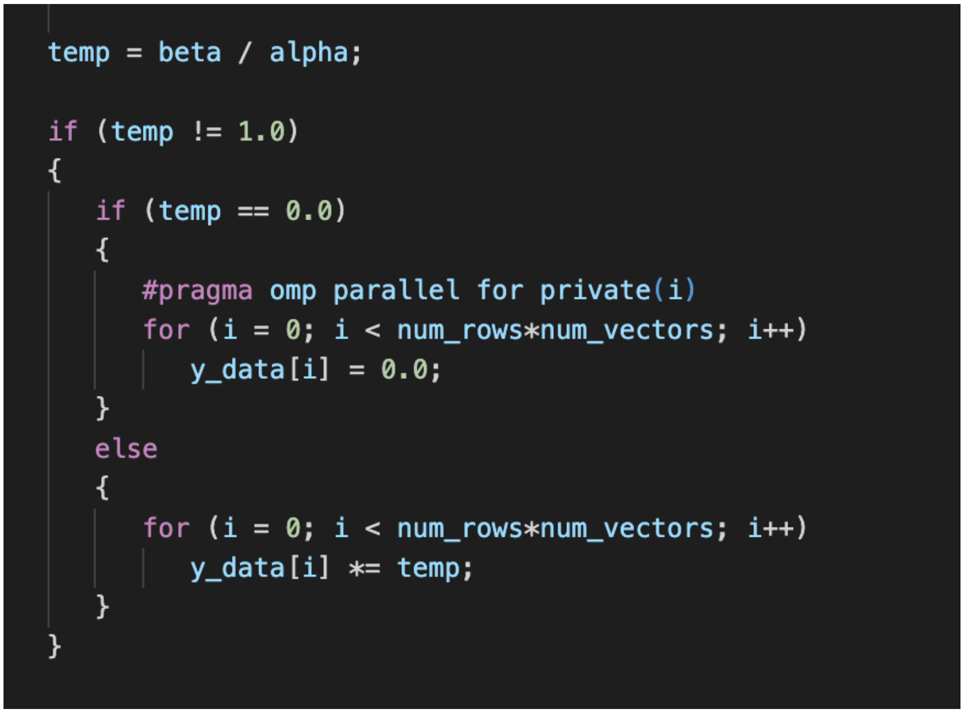
a. Source file: csr_matvec.c

b. Line number: 172

c. Code snippet:



```
else
{
   #pragma omp parallel for default(shared) private(i, j, jj, temp)
   for (i = 0; i < num_rows; i++)
   {
      if ( num_vectors==1 )
      {
         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
         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;
         }
   }
}
```

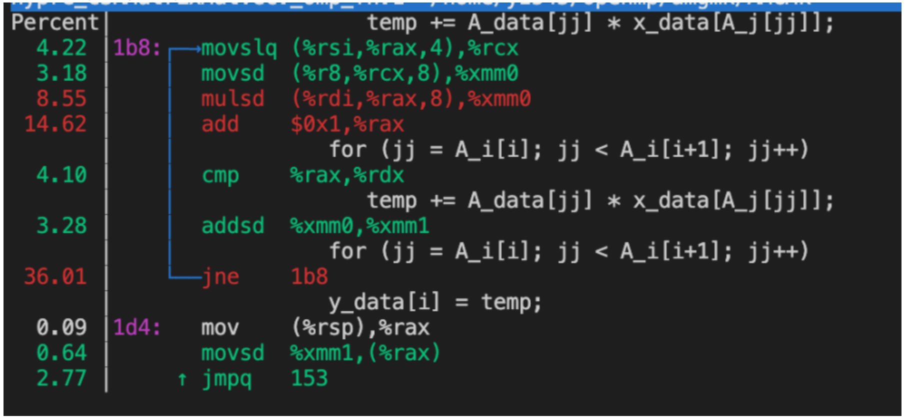


```c
temp = beta / alpha;

if (temp != 1.0)
{
    if (temp == 0.0)
    {
        #pragma omp parallel for private(i)
        for (i = 0; i < num_rows*num_vectors; i++)
            y_data[i] = 0.0;
    }
    else
    {
        for (i = 0; i < num_rows*num_vectors; i++)
            y_data[i] *= temp;
    }
}
```

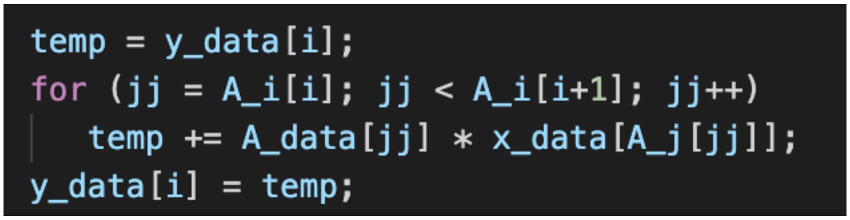
d. OpenMP directive description
As for the first optimization, I set up an omp parallel and for directive with default as shared variable. And private variables are i, j, jj and temp because these variables have read/write conflict. I didn't change the code other than that.
Similarly, I tried the same thing at the second spot, but slightly different in setting up private variables.

e. How do I know to add parallel code.
I used *perf report:*



```
Percent│                 temp += A_data[jj] * x_data[A_j[jj]];
  4.22 │1b8:┌─→movslq (%rsi,%rax,4),%rcx
  3.18 │    │  movsd  (%r8,%rcx,8),%xmm0
  8.55 │    │  mulsd  (%rdi,%rax,8),%xmm0
 14.62 │    │  add    $0x1,%rax
       │    │              for (jj = A_i[i]; jj < A_i[i+1]; jj++)
  4.10 │    │  cmp    %rax,%rdx
       │    │                 temp += A_data[jj] * x_data[A_j[jj]];
  3.28 │    │  addsd  %xmm0,%xmm1
       │    │              for (jj = A_i[i]; jj < A_i[i+1]; jj++)
 36.01 │    └──jne    1b8
       │                   y_data[i] = temp;
  0.09 │1d4:   mov    (%rsp),%rax
  0.64 │       movsd  %xmm1,(%rax)
  2.77 │     ↑ jmpq   153
```

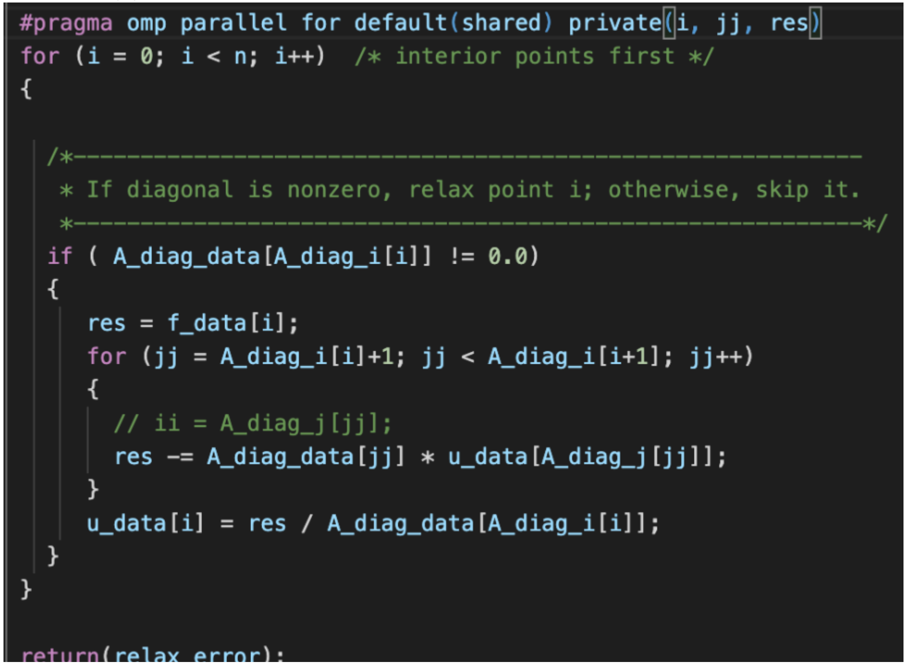
And from this report, I got to know that the most time-consuming part of code is:



```
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;
```

Thus, I built the OpenMP directive on top of this code.

(2) Relax
   a. Source file: relax.c
   b. Line number: 76
   c. Code snippet:



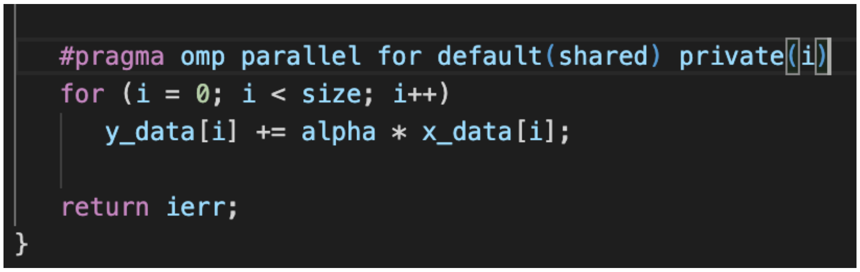
```
#pragma omp parallel for default(shared) private(i, jj, res)
for (i = 0; i < n; i++)  /* interior points first */
{

  /*-----------------------------------------------------------
   * If diagonal is nonzero, relax point i; otherwise, skip it.
   *-----------------------------------------------------------*/
  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[A_diag_j[jj]];
     }
     u_data[i] = res / A_diag_data[A_diag_i[i]];
  }
}

return(relax_error);
```

   d. OpenMP directive description
      I set up an omp parallel and for directive with default as shared variable. And private variables are i, jj and res because these variables have read/write conflict. I also removed variable ii in order to decrease a private storage overhead.

(3) Axpy
   a. Source file: vector.c
   b. Line number: 383
   c. Code snippet:



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

    return ierr;
}
```

d. OpenMP directive description
   I set up an omp parallel and for directive with default as shared variable. And private variable is only i. I didn't change the code other than that.

2 Performance Summary

(1) Sequential.
   Run *./AMGMk* 5 times, get averagely:
   Total Wall time = 2.86 seconds.

(2) Parallelize with 1 thread
   Run *OMP_NUM_THREADS=1 ./AMGMk* 5 times, get averagely,
   Total Wall time = 2.87 seconds

(3) Parallelize with 2 thread
   Run *OMP_NUM_THREADS=2 ./AMGMk* 5 times, get averagely,
   Total Wall time = 1.44 seconds

(4) Parallelize with 4 thread
   Run *OMP_NUM_THREADS=4 ./AMGMk* 5 times, get averagely,
   Total Wall time = 0.74 seconds

(5) Parallelize with 8 thread
   Run *OMP_NUM_THREADS=8 ./AMGMk* 5 times, get averagely,
   Total Wall time = 0.42 seconds

In conclusion, as the number of threads goes up, the performance increases, but not strictly proportionally.