

Project Report

Histogram

1 process

1.1 Code of four approaches inside for 'm' loop

Origin:

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

Locks:

```
// initialize array of locks  
omp_lock_t lock[256];  
for (i=0; i<256; i++) {  
    omp_init_lock(&lock[i]);  
}  
# 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++) {  
        // lock access to single array element  
        omp_set_lock(&lock[image->content[i][j]]);  
        histo[image->content[i][j]]++;  
        omp_unset_lock(&lock[image->content[i][j]]);  
    }  
}  
// destroy locks  
for (i=0; i<256; i++) {  
    omp_destroy_lock(&lock[i]);  
}
```

Atomic:

```
# 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: use reduction

```
# pragma omp parallel for collapse(2) default(shared) private(i, j) reduction(+:  
histo[0:256])  
for (i=0; i<image->row; i++) {  
    for (j=0; j<image->col; j++) {  
        histo[image->content[i][j]]++;  
    }  
}
```

2 results

run `bash ./run.sh` to get all performance results for the large input image uiuc-large.pgm.

2.1 validation

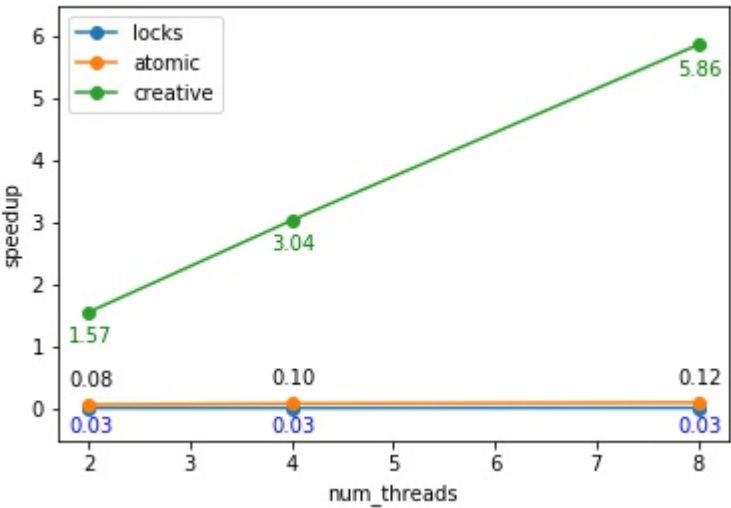
`diff validation-large.out ./reault/*.out` is used to check the correctness. There are no differences between validation file validation-large.out and output files except for the runtime.

2.2 execution time record

run `bash ./show_result.sh` to see results for this report. Results are shown in table below.

num_threads	sequential	locks	atomic	creative
2	5.63s	201.29s	68.76s	3.59s
4	5.62s	202.74s	56.39s	1.85s
6	5.63s	186.70s	48.09s	0.96s

speedup = sequential execution time / parallel execution time. Speedup curves are shown in picture below.



2.3 performance observation

The creative version which we utilized provided efficient code for sum by specifying "reduction(+: array)" performs best and gives the largest speedup. For the required methods, the sequential method gives the shortest time and best performance. Speedup increases with number of threads for atomic and creative methods while doesn't change much for sequential and locks methods. That's because when using locks, no matter how much number of threads are used, every operation of adding 1 to the element should wait for the completion of addition of other threads, causing constantly high value of running time. There are also extra instructions to communicate between threads and determine whether the lock is unset, which make the performance worse. Regarding atomic method, threads still need to synchronize to avoid race condition which means parallelism is lost. However, the compiler uses special hardware instructions to reduce overhead of entering and exiting code when lock and unlock. Therefore, it shows better performance than locks method. Reduction for sum allocates a copy of element per thread. The local element won't be influenced by other threads thus avoid writing to the same location simultaneously. Since each thread is operating separately, we don't need lock and atomic to wait for finishing instructions from other threads and fully utilized the parallel execution time. Finally this method combines per-thread copies into original element to ensure the correct results.

AMG

1 Analysis of Flat Profile

To link the timber library and enable gprof, added the following content to the makefile.

```
11 CFLAGS = -O3 -fopenmp -c
12
13 LDFLAGS = -O3 -fopenmp -lm -L. -ltimer -pg
```

Looking into the flat profile, we can clearly see the top three time consuming portion, which is the 3 computation task mentioned in the homework specification. Each of them is a function in their individual C program, which can be parallelized separately using OpenMP for better result. Amongst these three potential candidates, hypre_CSRMatrixMatvec and hypre_BoomerAMGSeqRelax shows better potential of improvement by occupying 41.44% and 56.08% of the total runtime respectively.

```
amgmk > E analysis.txt
1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4 % cumulative self self total
5 time seconds seconds calls Ts/call Ts/call name
6 56.08 1.57 1.57 hypre_BoomerAMGSeqRelax
7 41.44 2.73 1.16 hypre_CSRMatrixMatvec
8 2.50 2.80 0.07 hypre_SeqVectorApxy
```

2 Changes Made to the Code

run `bash ./run.sh` to get all performance results for the large input image uiuc-large.pgm.

2.1 To hypre_BoomerAMGSeqRelax

Made changes to file `scr_matvec.c`. Added parallel task for the for loop on line 76 in file relax.c, i, jj, ii are declared private and res reduction (is like private in this case, but faster).

```
#pragma omp parallel for default(shared) private(i, jj, ii) reduction (-: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[ii];
        }
        u_data[i] = res / A_diag_data[A_diag_i[i]];
    }
}
```

2.2 To hypre_CSRMatrixMatvec

Made several changes to file `csr_matvec.c`. On line 107, added parallel task with default shared and private i.

```
if (alpha == 0.0)
{
    #pragma omp parallel for default(shared) private(i)
    for (i = 0; i < num_rows*num_vectors; i++)
        y_data[i] *= beta;

    return ierr;
}
```

On line 124 and 129, added parallel task with default shared and private i.

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

On line 143, added parallel task with default shared, private i, jj, m, j and reduction temp.

```

if (num_rownnz < xpar*(num_rows))
{
    #pragma omp parallel for default(shared) private(i, jj, m, j)
    reduction(+:temp)
    for (i = 0; i < num_rownnz; i++)
    {
        m = A_rownnz[i];

        /*
        * for (jj = A_i[m]; jj < A_i[m+1]; jj++)
        * {
        *     j = A_j[jj];
        *     y_data[m] += A_data[jj] * x_data[j];
        * } */
        if ( num_vectors==1 )
        {
            temp = y_data[m];
            for (jj = A_i[m]; jj < A_i[m+1]; jj++)
                temp += A_data[jj] * x_data[A_j[jj]];
            y_data[m] = temp;
        }
        else
            for ( j=0; j<num_vectors; ++j )
            {
                temp = y_data[ j*vecstride_y + m*idxstride_y ];
                for (jj = A_i[m]; jj < A_i[m+1]; jj++)
                    temp += A_data[jj] * x_data[ j*vecstride_x +
A_j[jj]*idxstride_x ];
                y_data[ j*vecstride_y + m*idxstride_y] = temp;
            }
    }
}

```

On line 174, added parallel task with default shared, private i, jj, j and reduction temp.

```

else
{
    #pragma omp parallel for default(shared) private(i, jj, j) reduction(+: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;
        }
    }
}

```

On line 204, added parallel task with default shared and private i.

```

if (alpha != 1.0)
{
    #pragma omp parallel for default(shared) private(i)
    for (i = 0; i < num_rows*num_vectors; i++)
        y_data[i] *= alpha;
}

```

2.3 To hypre_SeqVectorApxy

Made changes to file `vector.c`. On line 383, added parallel task with default shared and private i.

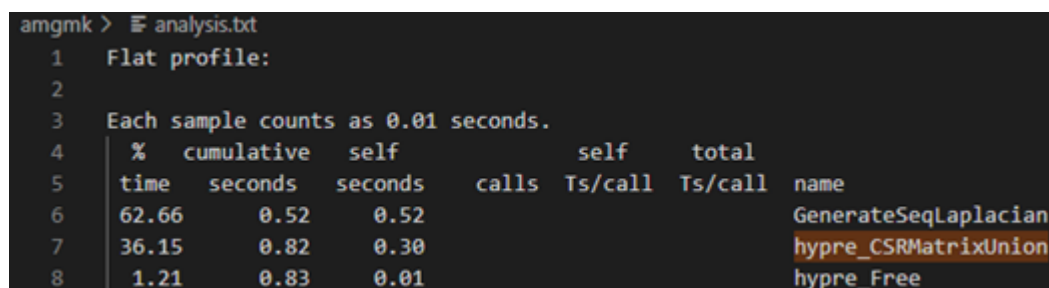
```

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

```

2.4 Further Performance Optimization

After making the previous changes, the new flat profile indicate that there are other functions that could be further optimized.



```

amgmk > analysis.txt
1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4 % cumulative self self total
5 time seconds seconds calls Ts/call Ts/call name
6 62.66 0.52 0.52 GenerateSeqLaplacian
7 36.15 0.82 0.30 hypre_CSRMatrixUnion
8 1.21 0.83 0.01 hypre_Free

```

Hence the following changes were made to file `laplace.c`:

On line 73, added parallel task with default shared and private i.

```
#pragma omp parallel for default(shared) private(i)
for (i=0; i < grid_size; i++)
{
    x_data[i] = 0.0;
    sol_data[i] = 0.0;
    rhs_data[i] = 1.0;
}
```

On line 166, added parallel task with default shared and private i, j.

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

3 Performance Comparison

3.1 Runtime With Different Thread Count

Number of Threads	MATVEC (seconds)	Relax (Seconds)	Axpy (Seconds)	Total Wall Time (Seconds)
Sequential	1.00	1.00	1.00	1.00
1	1.00	0.90	0.90	0.94
2	1.95	1.82	2.31	1.87
4	3.69	3.48	4.93	3.52
8	6.65	7.04	7.40	6.04

3.2 Speed Up Achieved With Different Thread Count

Number of Threads	MATVEC (speedup)	Relax (speedup)	Axpy (speedup)	Total Wall Time (speedup)
Sequential	1.077	1.444	0.074	2.616
1	1.080	1.588	0.082	2.774
2	0.552	0.793	0.032	1.398
4	0.292	0.414	0.015	0.742
8	0.162	0.205	0.010	0.433

We can see that the speed up is proportional to the number of threads, with higher number of threads have better performance.