ECE 565: Performance Optimization & Parallelism
Homework 4

Prathikshaa Rangarajan (pr109) **Rijish Ganguly (rg239)**

November 6, 2019

1 Histogram

1.1 Using Locks

```
/* obtain histogram from image, repeated 100 times */
for (m=0; m<100; m++) {
#pragma omp parallel for collapse(2) private(i, j)
  for (i=0; i<image->row; i++) {
    for (j=0; j<image->col; j++) {
      omp_set_lock(&lock_set[image->content[i][j]]);
      histo[image->content[i][j]]++;
      omp_unset_lock(&lock_set[image->content[i][j]]);
    }
  }
}
```

Figure 1: Parallelized code using Open MP and locks

1.2 Using Atomics

```
t_start = omp_get_wtime();

/* obtain histogram from image, repeated 100 times */
for (m=0; m<100; m++) {
#pragma omp parallel for collapse(2) 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]]++;
    }
  }
}
```

Figure 2: Parallelized code using Open MP and atomic update operation

1.3 Creative Method

```
 /* obtain histogram from image, repeated 100 times */
 for (m=0; m<100; m++) {
   for (i=0; i<image->row; i++) {
    #pragma omp parallel for
     for (j=0; j < image->col; j++) {
       histo_arr[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; j++) {
     histo[i] += histo_arr[i][j];
   }
 }
```

Figure 3: Parallelized code using Open MP and 2-D array

This method uses an idea similar to reduction where we use a two dimensional array corresponding where we make copies of the original histogram array per thread and perform the updates per thread. The final histogram array is the resulting sum of the thread-wise histogram arrays.

1.4 Results

Threads	Atomic	Locks	Creative	Sequential
2	73.89s	205.56s	42.96s	5.59s
4	50.07s	204.5s	48.88s	5.59s
8	36.2s	172.26s	48.65s	5.59s

Figure 4: Performance measurement for sequential and parallelized code

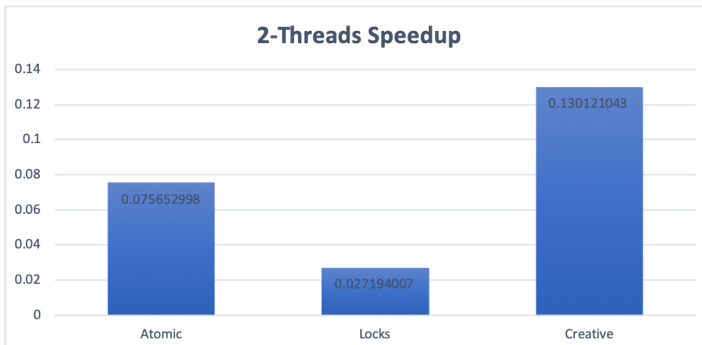

Figure 5: Speedup for different versions of parallelized code using two threads

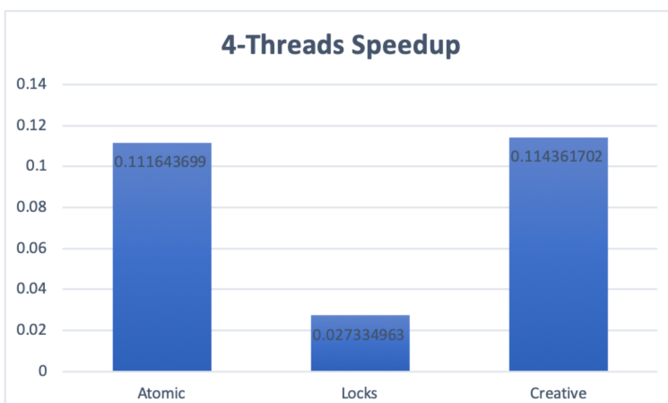

Figure 6: Speedup for different versions of parallelized code using four threads

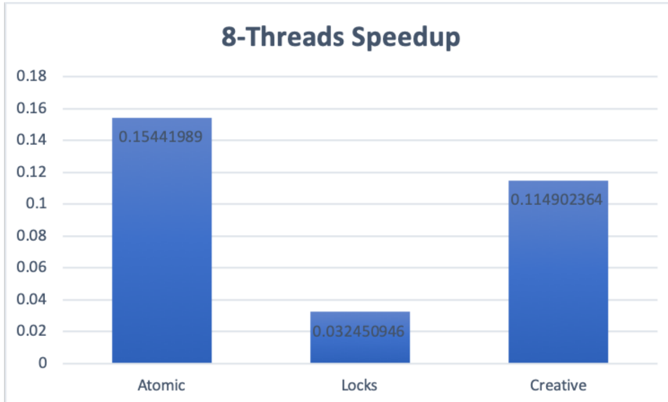

Figure 7: Speedup for different versions of parallelized code using eight threads

1.5 Analysis

Among all the parallelized code version, our histo_creative had the best performance. For both the locks and atomic update version, the program takes a considerable amount of time to execute, as the threads stall when the other threads are in the critical section which penalizes the performance significantly. For the lock version some time is lost in synchronization overhead as each thread must acquire a lock before entering the critical section and release it afterwards. As the atomic directive reduces some overhead because no lock needs to be acquired, performance is much better. With increase in number of threads, there is a trade-off between resource contention and increased communication latency.

The histo_creative achieved accurate results in a comparatively shorter time. We implemented the program using a 2D array which practically added a second dimension to the histo array which was determined by the thread number. After the execution of the main loop, the elements over all the threads representing a given original histo array element was summed and assigned to the array. For getting the number of thread we used the omp_get_thread_num() function.

2 AMG

2.1 Code Profiling

```
core@vcm-11541:~/hw4/amgmk$ head analysis.txt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 57.10      1.57     1.57     1000     1.57     1.57  hypre_BoomerAMGSeqRelax
 39.64      2.66     1.09     1000     1.09     1.09  hypre_CSRMatrixMatvec
  2.91      2.74     0.08     1000     0.08     0.08  hypre_SeqVectorAxpy
  0.36      2.75     0.01        2     5.00     5.00  GenerateSeqLaplacian
  0.00      2.75     0.00       26     0.00     0.00  hypre_CAlloc
```

Figure 8: Code profiling using gprof

This shows that the maximum execution time is in the `hypre_BoomerAMGSeqRelax()` and `hypre_CSRMatricMatvec()` functions.

```
core@vcm-11541:~/hw4/amgmk$ grep -iRn "hypre_BoomerAMGSeqRelax"
seq_mv.h:176:int hypre_BoomerAMGSeqRelax( hypre_CSRMatrix *A , hypre_Vector *x, hypre_Vector *y) ;
main.c:195:      hypre_BoomerAMGSeqRelax(A, sol, x);
relax.c:39: * hypre_BoomerAMGSeqRelax
relax.c:42:int  hypre_BoomerAMGSeqRelax( hypre_CSRMatrix *A,
Binary file main.o matches
Binary file relax.o matches
Binary file AMGMk matches
analysis.txt:6: 57.10      1.57     1.57     1000     1.57     1.57  hypre_BoomerAMGSeqRelax
analysis.txt:60:                1.57    0.00    1000/1000        hypre_BoomerAMGSeqRelax [2]
analysis.txt:69:[2]     57.1    1.57    0.00    1000         hypre_BoomerAMGSeqRelax [2]
analysis.txt:237:   [2] hypre_BoomerAMGSeqRelax [4] hypre_CSRMatrixMatvec  [11] hypre_SeqVectorDestro
y
core@vcm-11541:~/hw4/amgmk$ grep -iRn "hypre_CSRMatrixMatvec"
seq_mv.h:151:int hypre_CSRMatrixMatvec ( double alpha , hypre_CSRMatrix *A , hypre_Vector *x , double
 beta , hypre_Vector *y );
seq_mv.h:152:int hypre_CSRMatrixMatvecT ( double alpha , hypre_CSRMatrix *A , hypre_Vector *x , doubl
e beta , hypre_Vector *y );
seq_mv.h:153:int hypre_CSRMatrixMatvec_FF( double alpha , hypre_CSRMatrix *A , hypre_Vector *x , doub
le beta , hypre_Vector *y , int *CF_marker_x , int *CF_marker_y , int fpt );
main.c:141:      hypre_CSRMatrixMatvec(1,A,x,0,y);
csr_matvec.c:39: * hypre_CSRMatrixMatvec
csr_matvec.c:43:hypre_CSRMatrixMatvec( double           alpha,
csr_matvec.c:208: * hypre_CSRMatrixMatvecT
csr_matvec.c:212: *   From Van Henson's modification of hypre_CSRMatrixMatvec.
csr_matvec.c:216:hypre_CSRMatrixMatvecT( double           alpha,
csr_matvec.c:395: * hypre_CSRMatrixMatvec_FF
csr_matvec.c:399:hypre_CSRMatrixMatvec_FF( double           alpha,
Binary file main.o matches
Binary file csr_matvec.o matches
Binary file AMGMk matches
analysis.txt:7: 39.64      2.66     1.09     1000     1.09     1.09  hypre_CSRMatrixMatvec
analysis.txt:73:                1.09    0.00    1000/1000        hypre_CSRMatrixMatvec [4]
analysis.txt:82:[4]     39.6    1.09    0.00    1000         hypre_CSRMatrixMatvec [4]
analysis.txt:237:   [2] hypre_BoomerAMGSeqRelax [4] hypre_CSRMatrixMatvec  [11] hypre_SeqVectorDestro
y
core@vcm-11541:~/hw4/amgmk$ grep -iRn "hypre_SeqVectorAxpy"
seq_mv.h:168:int hypre_SeqVectorAxpy ( double alpha , hypre_Vector *x , hypre_Vector *y );
vector.c:365: * hypre_SeqVectorAxpy
vector.c:369:hypre_SeqVectorAxpy( double        alpha,
main.c:244:      hypre_SeqVectorAxpy(alpha,x,y);
Binary file main.o matches
Binary file vector.o matches
Binary file AMGMk matches
analysis.txt:8:  2.91      2.74     0.08     1000     0.08     0.08  hypre_SeqVectorAxpy
analysis.txt:85:[5]      2.9    0.08    0.00    1000         hypre_SeqVectorAxpy [5]
analysis.txt:89:                0.08    0.00    1000/1000        hypre_SeqVectorAxpy [5]
analysis.txt:239:  [13] hypre_CSRMatrixCreate   [5] hypre_SeqVectorAxpy    [12] hypre_SeqVectorSetCon
stantValues
```

Figure 9: Critical Source Code identified based on Profiling

Based on the profiling information seen in Fig.8 and Fig.9, we know that the functions of interest are in `relax.c, line 42`, `csr_matvec.c, line 43` and `vector.c, line 369`, where the functions of interest are defined.

2.2 Code Modifications using OpenMP

Based on print and timing based analysis, several sections of the code were modified. The code in `relax.c` was best optimized just by parallelizing the outer loop. Directives such as collapse do not work because of the conditional statements within the code.

2.2.1 relax.c

Parallelize outer loop:

```
76#pragma omp parallel for default(shared) private(i, jj)
77   for (i = 0; i < n; i++)      /* interior points first */
78   {
79
80     /*-----------------------------------------------------------
81      * If diagonal is nonzero, relax point i; otherwise, skip it.
82      *-----------------------------------------------------------*/
83     if ( A_diag_data[A_diag_i[i]] != 0.0)
84     {
85       res = f_data[i];
86       for (jj = A_diag_i[i]+1; jj < A_diag_i[i+1]; jj++)
87       {
88         ii = A_diag_j[jj];
89         res -= A_diag_data[jj] * u_data[ii];
90       }
91       u_data[i] = res / A_diag_data[A_diag_i[i]];
92     }
93   }
94
95   return(relax_error);
96}
```

Figure 10: Change 1 in relax.c, `line 76`

```
104   /*-----------------------------------------------------------------------
105    * Do (alpha == 0.0) computation - RDF: USE MACHINE EPS
106    *-----------------------------------------------------------------------*/
107    // fprintf(stderr, "before alpha is 0\n");
108
109    if (alpha == 0.0)
110    {
111#pragma omp parallel for default(shared) private(i)
112       for (i = 0; i < num_rows*num_vectors; i++)
113          y_data[i] *= beta;
114       fprintf(stderr, "alpha is 0\n"); // hw4 - unreachable
115
116       return ierr;
117    }
118
```

Figure 11: Change 1 in csr_matvec.c, `line 111`

```
119    /*-----------------------------------------------------------------------
120     * y = (beta/alpha)*y
121     *-----------------------------------------------------------------------*/
122    // fprintf(stderr, "before y = beta/alpha \n");
123
124    temp = beta / alpha;
125
126    if (temp != 1.0)
127    {
128      // fprintf(stderr, "y = beta / alpha \n"); // hw4 reachable
129
130       if (temp == 0.0)
131       {
132#pragma omp parallel for default(shared) private(i)
133          for (i = 0; i < num_rows*num_vectors; i++)
134             y_data[i] = 0.0;
135       }
136       else
137       {
138#pragma omp parallel for default(shared) private(i)
139          for (i = 0; i < num_rows*num_vectors; i++)
140             y_data[i] *= temp;
141       }
142    }
143
```

Figure 12: Change 2 in csr_matvec.c, `lines 132, 138`

```
144    /*-----------------------------------------------------------------
145     * y += A*x
146     *-----------------------------------------------------------------*/
147
148/* use rownnz pointer to do the A*x multiplication  when num_rownnz is smaller than num_rows */
149
150    if (num_rownnz < xpar*(num_rows))
151    {
152#pragma omp parallel for default(shared) private(i)
153       for (i = 0; i < num_rownnz; i++)
154       {
155          m = A_rownnz[i];
156
157          /*
158           * for (jj = A_i[m]; jj < A_i[m+1]; jj++)
159           * {
160           *         j = A_j[jj];
161           *  y_data[m] += A_data[jj] * x_data[j];
162           * } */
163          if ( num_vectors==1 )
164          {
165             tempx = y_data[m];
166             for (jj = A_i[m]; jj < A_i[m+1]; jj++)
167                tempx +=  A_data[jj] * x_data[A_j[jj]];
168             y_data[m] = tempx;
169          }
170          else
171             for ( j=0; j<num_vectors; ++j )
172             {
173                tempx = y_data[ j*vecstride_y + m*idxstride_y ];
174                for (jj = A_i[m]; jj < A_i[m+1]; jj++)
175                   tempx +=  A_data[jj] * x_data[ j*vecstride_x + A_j[jj]*idxstride_x ];
176                y_data[ j*vecstride_y + m*idxstride_y] = tempx;
177             }
178       }
179
180    }
181    else
182    {
183#pragma omp parallel for default(shared) private(i)
184       for (i = 0; i < num_rows; i++)
185       {
```

Figure 13: Change 3 in csr_matvec.c, `line 152`

```
core@vcm-11541:~/hw4/amgmk$ gprof AMGMk gmon.out |head
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 59.01      0.59     0.59        2   295.06   295.06  GenerateSeqLaplacian
 33.01      0.92     0.33                             hypre_CSRMatrixUnion
  8.00      1.00     0.08     1000     0.08     0.08  hypre_SeqVectorAxpy
  0.00      1.00     0.00     1000     0.00     0.00  hypre_BoomerAMGSeqRelax
  0.00      1.00     0.00     1000     0.00     0.00  hypre_CSRMatrixMatvec
core@vcm-11541:~/hw4/amgmk$
```

Figure 14: Results after modifying both relax.c and csr_matvec.c

As seen above, AXPY related functions still do not show up as a bottleneck and are irrelevant to current execution.

2.3 Results

No. of threads	MATVEC	Relax	Axpy	Total
Baseline Sequential	1.197	1.573	0.079	2.870
1	1.140	1.161	0.079	2.853
2	0.584	0.829	0.073	1.536
4	0.361	0.414	0.079	0.873
8	0.236	0.228	0.078	0.564

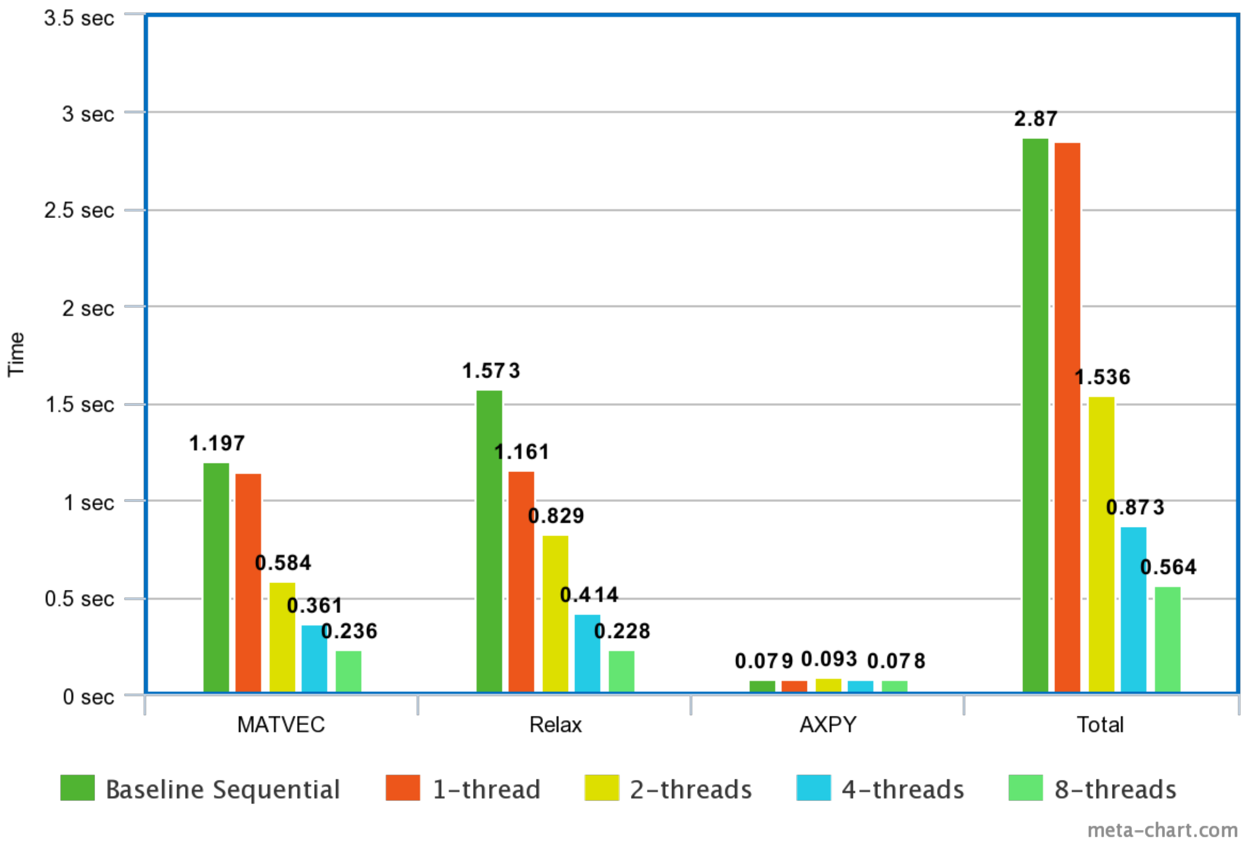

Figure 15: Timing Results of baseline sequential and multithreaded performance

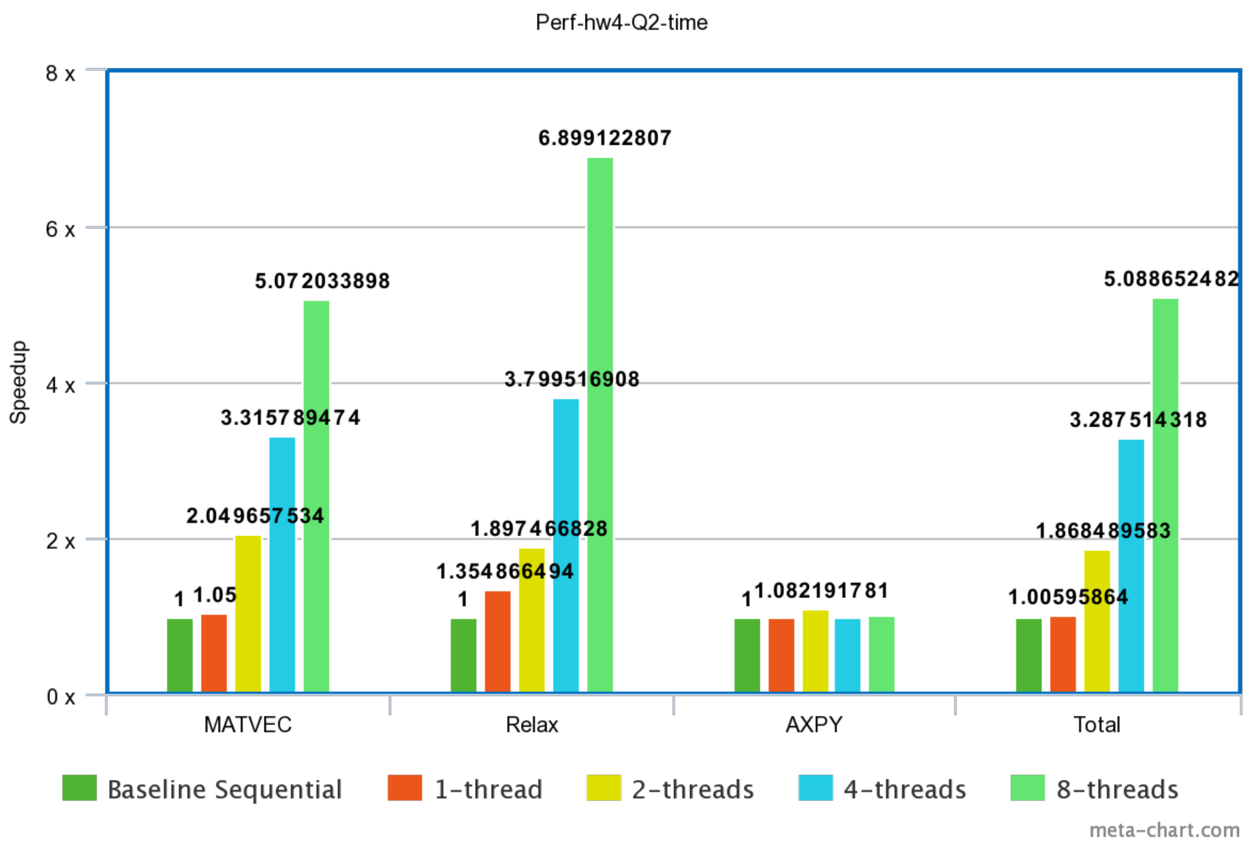

Figure 16: Speedup against baseline sequential for different threads

As inferred from the profiling and timing information, AXPY is not a bottleneck or critical to the overall performance to the code. Even after applying the above parallelism techniques, the next bottlenecks are functions not related to AXPY in any manner. Therefore, it has been excluded from the analysis.

As seen in fig.16, the total execution sees up to 5x improvement. MATVEC has been improved by almost 5x and Relax has improved by almost 7x when going from baseline to 8-threads.