ECE 565 Hw4 report

Koushik Annareddy Sreenath

Problem 1 Histogram

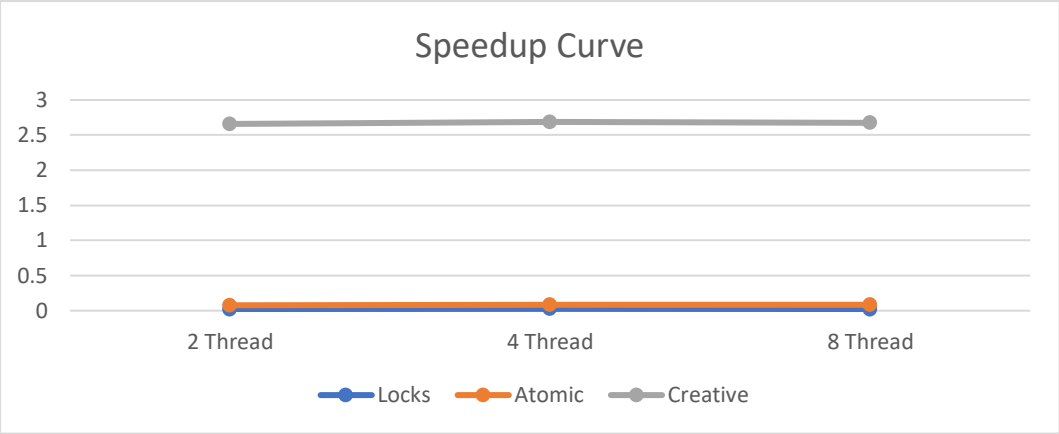
	2 threads	4 threads	8 threads
Sequential (Default)	4.92	4.92	4.90
Locks	203.68	183.68	199.27
Atomic	63.71	57.93	57.71
Creative	1.85	1.83	1.83

1. **Sequential (Default):** As expected, the sequential version's execution time remains constant regardless of the number of threads specified, since it does not utilize any parallelism. This serves as our baseline for comparison with parallel implementations.
2. **Locks:** The locks implementation incurs significant overhead due to the lock and unlock mechanisms. Even though multiple threads are used, they are forced to wait for the lock to be released before proceeding, which diminishes the benefits of parallelism. As evidenced by the execution times being far higher than the sequential baseline.
3. **Atomic:** The atomic implementation leverages hardware optimization for atomic operations, which reduces the overhead compared to locks. This implementation performs better than locks, as atomic operations efficiently ensure that histogram updates are not interrupted, thereby preventing race conditions. Despite this, the atomic version still has overhead associated with managing critical sections and is not as fast as the sequential version.
4. **Creative solution:** The creative solution outperforms all other implementations, including the sequential baseline because there is no critical regions. By allocating a private histogram for each thread (private_histos), I eliminate the need for synchronization mechanisms during the histogram update phase. Each thread updates its own private histogram, which is then combined into the final histogram after a synchronization barrier. The final combination is also done in parallel, which contributes to the efficiency of this method. The trade-off is the additional memory required for the private histograms.

   The correctness of this method is ensured by the fact that each thread is working independently on its own copy of the histogram and only updates the global histogram after all threads have completed their computation, thus eliminating race conditions.

Below is the graph plotting the Speedup Curve for the above

	Locks	Atomic	Creative
2 Thread	0.024	0.077	2.659
4 Thread	0.026	0.084	2.688
8 Thread	0.024	0.084	2.677



Problem2: AMG

When we did code profiling we observed that 'hypre_CSRMatrixMatvec()' function consumed 52.70% of the overall execution time. Therfore, following Amdahl's Law I aimed to maximize performance gains by targeting the most time-intensive parts of the code (i.e. hypre_CSRMatrixMatvec()). I did this by following two strategies

1) Loop Unstitching: Initially, the function contained several if statements within a for loop, which were evaluated at every iteration. To reduce the overhead, I refactored the code by moving the if conditions outside of the loop, thus executing the conditional checks only once before entering the loop

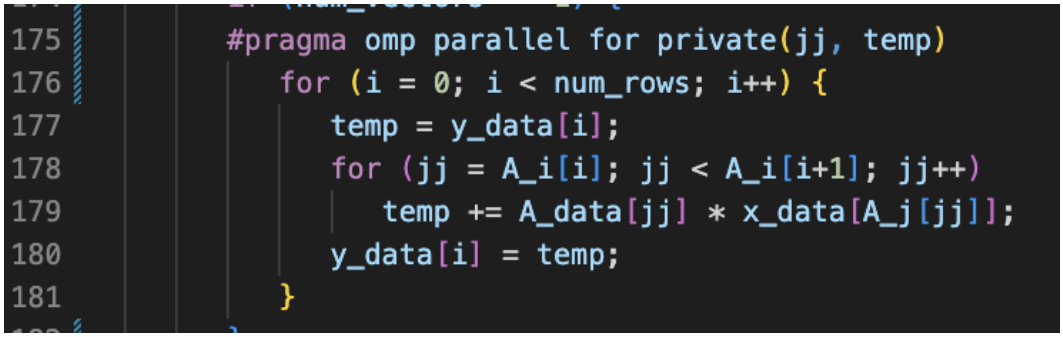
   Before:

```
171     for (i = 0; i < num_rows; i++)
172     {
173        if ( num_vectors==1 )
174        {
175           temp = y_data[i];
176           for (jj = A_i[i]; jj < A_i[i+1]; jj++)
177              temp += A_data[jj] * x_data[A_j[jj]];
178           y_data[i] = temp;
179        }
180        else
181           for ( j=0; j<num_vectors; ++j )
182           {
183              temp = y_data[ j*vecstride_y + i*idxstride_y ];
184              for (jj = A_i[i]; jj < A_i[i+1]; jj++)
185              {
186                 temp += A_data[jj] * x_data[ j*vecstride_x + A_j[jj]*idxstride_x ];
187              }
188              y_data[ j*vecstride_y + i*idxstride_y ] = temp;
189           }
190     }
```

   After:

```
174     if (num_vectors == 1) {
175     #pragma omp parallel for private(jj, temp)
176        for (i = 0; i < num_rows; i++) {
177           temp = y_data[i];
178           for (jj = A_i[i]; jj < A_i[i+1]; jj++)
179              temp += A_data[jj] * x_data[A_j[jj]];
180           y_data[i] = temp;
181        }
182     }
183     else {
184        #pragma omp parallel for private(jj, temp)
185        for (i = 0; i < num_rows; i++) {
186           for (j = 0; j < num_vectors; ++j) {
187              temp = y_data[ j*vecstride_y + i*idxstride_y ];
188              for (jj = A_i[i]; jj < A_i[i + 1]; jj++) {
189                 temp += A_data[jj] * x_data[ j*vecstride_x + A_j[jj]*idxstride_x ];
190              }
191              y_data[ j*vecstride_y + i*idxstride_y ] = temp;
192           }
193        }
194     }
```

2) OpenMP Parallelization: I used OpenMP directive used was #pragma omp parallel for private(jj, temp), applied at lines 175 and 185. This directive instructs the compiler to parallelize the loop while ensuring that each thread has its own private copies of the variables jj and temp, preventing any race conditions from occurring.



The final results due to the optimisation are.

	1 thread	2 thread	4 thread	8 thread
Base version	2.488368	2.457374	2.540370	2.780921
Optimised version	2.203745	1.897081	1.831993	1.699576

As expected, the base version's execution time remains constant regardless of the number of threads specified, since it does not utilize any parallelism. Whereas Optimised version has parallel regions hence it takes advantages of multiple threads and the performance improves when we increase the no of threads.