ICS 632

Assignment #2

Hualiang Li

Folder Structure:

-/assignment2
- -/exercise1
  - exercise1.c
  - run_all.py
  - exercise1.slurm
  - -/data
    - perf_output_data
- -/exercise2
  - exercise2.c
  - naïve.py
  - naïve.slurm
  - exercise2_fast.c
  - fast.py
  - naïve.slurm
  - combine.py
  - combine.slurm
  - -/data
    - perf_output_data

How to Run:

To run exercise1 on Cray:

sbatch exercise1.slurm

To run exercise2 naïve version on Cray:

sbatch naïve.slurm

To run exercise2 tiled version on Cray:

sbatch fast.slurm

To run both version and plot 2 curve into one figure:

sbatch combine.slurm

Exercise 1

Question 1:

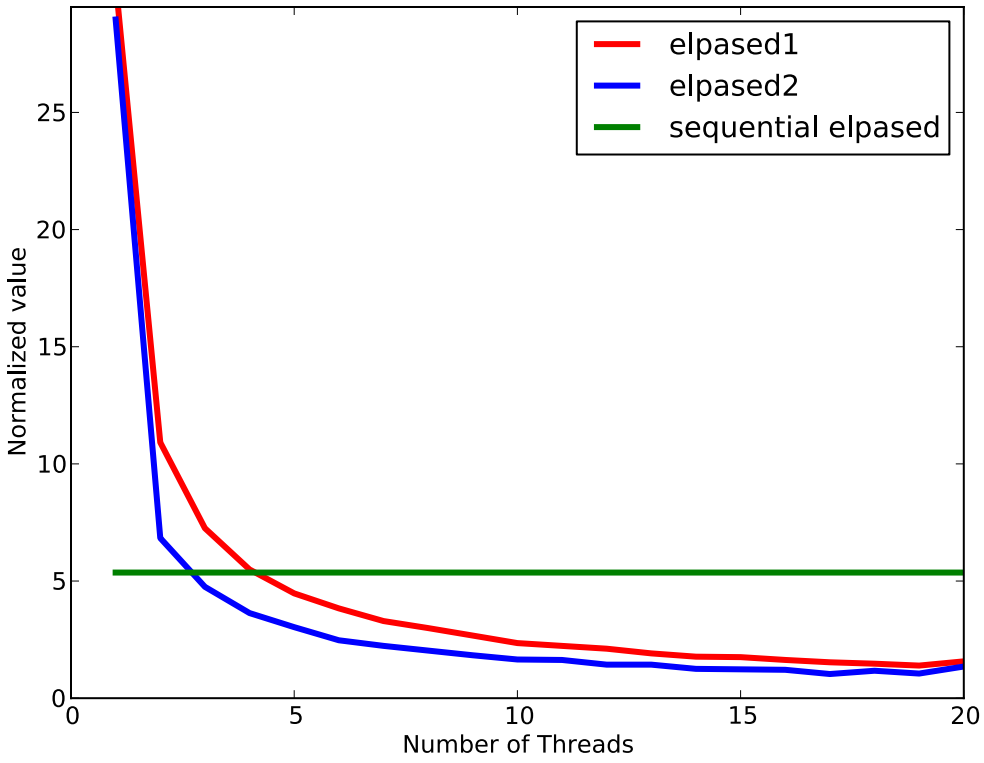
3 different parallel versions are implemented as in the figure below. The third version, which parallel the most inner for loop might cause race condition if we just use simple pragma omp parallel for. This is because each thread has its own i;j, which means there could be situation that multiple data written into same memory address.

```
30 #ifdef p1
31 #pragma omp parallel for
32     for(i=0; i<N; i++){
33         for(j=0; j<N; j++){
34             for(k=0; k<N; k++){
35                 C[i][j] += A[i][k] * B[k][j];
36             }
37         }
38     }
39 #endif
40
41 #ifdef p2
42     for(i=0; i<N; i++){
43 #pragma omp parallel for
44         for(j=0; j<N; j++){
45             for(k=0; k<N; k++){
46                 C[i][j] += A[i][k] * B[k][j];
47             }
48         }
49     }
50 #endif
51
52 #ifdef p3
53     for(i=0; i<N; i++){
54         for(j=0; j<N; j++){
55 #pragma omp parallel for
56             for(k=0; k<N; k++){
57 #pragma omp critical
58                 C[i][j] += A[i][k] * B[k][j];
59             }
60         }
61     }
62 #endif
63
64 #ifdef s
```

Question 2:

To make the program running about 5 seconds, size N is defined to be 2500. The elapsed time vs. number of threads is plotted in following figure. Notice that, the

third parallel version run too long time, which is not shown in the plot. This is because to avoid race condition, I have to declare the critical section, which has a huge overhead.



Question 3:

From the plot above, we can see that the second parallel version is the fastest, with about 17 threads, which runs about 1.03 seconds. The speedup is about 5.3/1.03 = 5. The parallel efficiency is about 5/17 = 30%. This is low. It is because the overhead of parallel computing is high. Especially when the threads number is getting higher, the more communication and more synchronization is required.

Exercise 2:

Question 1:

Using simple parallel for pragama will not lead to correct execution because of the data dependency issue. The pixel value is depend on four neighbor values, which means, if the neighbor value is read before it is updated, the calculation is not valid. Therefore, we need to find a for loop that in this loop, the data calculation in the same loop is independent from each other.

Question 2:

The general idea to parallel computing this is as following. Say we have a matrix

like:

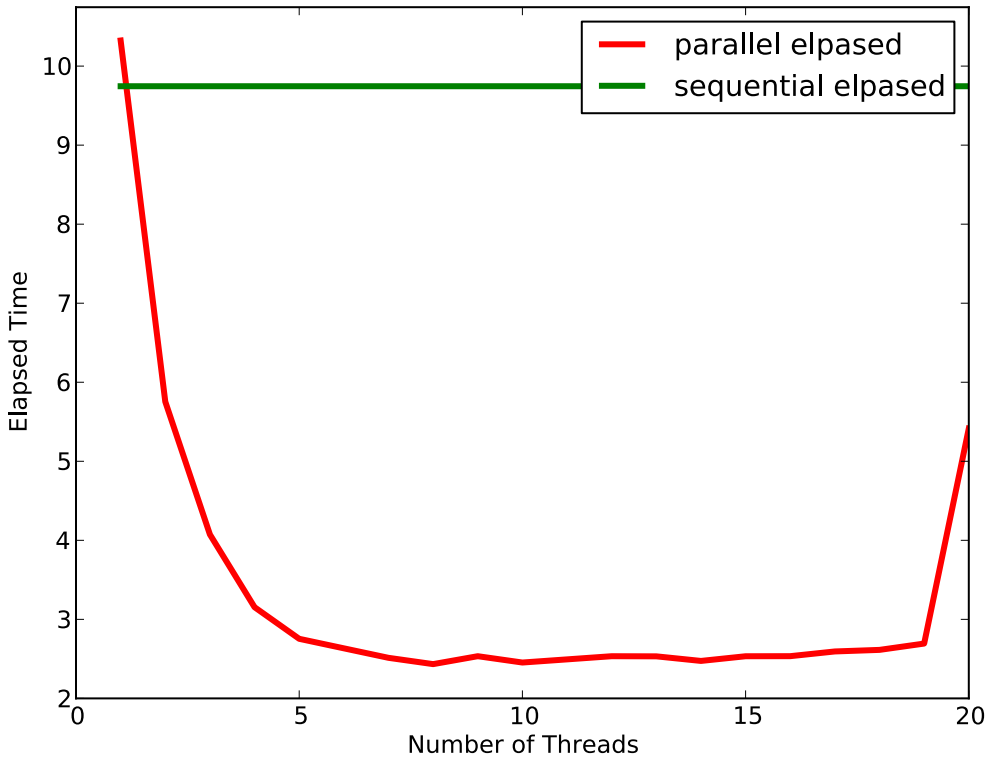
1, 2, 3
4, 5, 6
7, 8, 9

The order I traverse the matrix would be [1], [2, 4], [3, 5, 7], [6, 8], [9]. The elements in the same [] can be computed parallel.

```
#ifdef p
    omp_set_num_threads(num_threads);
    // Loop for num_iterations iterations
    int s;
    for (iter = 0; iter < num_iterations; iter++) {
        for(s = 2; s < 2*N+1; s++) {
            int z = s < N+2 ? 1: s-N;
#pragma omp parallel for
            for (i = z; i < s-z+1; i++) {
                A[i][s-i] = (3*A[i-1][s-i] + A[i+1][s-i] + 3*A[i][s-i-1] + A
[i][s-i+1])/4;
            }
        }
    }
#endif
```

Question 3:

The iterations number is defined to be 16 such that the sequential program will run approximately 10 seconds. The plot for elapsed time vs. number of threads is shown below. I observed that the more threads, the faster program would run. But the acceleration is getting smaller. Moreover, when there is only one thread, the parallel version took more time than the original version. Also, the run time is almost same when threads number is greater than 6. This is because with multiple threads, the overhead for threads communication, synchronization is big. Besides, the cache miss rate is higher compared with sequential version. This can be observed by using “perf” tool. Another observation I have is when the threads number is 20, which is exactly the core number of the compute node, the performance is getting worse. I am guessing this is because the system needs a master thread, so one of the cores has to switch between two threads to run program, this slow down the run time. The best speedup is around 4. The parallel efficiency is low with large amount of threads. We can see there is no gain with more than 6 threads. We can still improve the performance by increasing the cache hit, which is introduced in question 4.



Question 4:

I choose the tile size to be 400. Below is my implementation:

```
#ifdef p
    omp_set_num_threads(num_threads);
    // Loop for num_iterations iterations
    int s;
    int maxSum = 2*(N/T)-1;
    int ii, jj, iii, jjj;
    for (iter = 0; iter < num_iterations; iter++) {
        for(s = 0; s < maxSum; s++) {
            int z = s < (N/T) ? 0 : s-(N/T)+1;
#pragma omp parallel for
            for (i = z; i < s-z+1; i++) {
                for(ii=0; ii < T; ii++){
                    for(jj=0; jj< T; jj++){
                        iii = i*T+1+ii;
                        jjj = (s-i)*T+1+jj;
                        A[iii][jjj] = (3*A[iii-1][jjj] + A[iii+1][jjj] + 3*A[iii][jjj-1] + A[iii][jjj+1])/4;
                    }
                }
            }
        }
    }
#endif
```

The basic idea for this is divide the matrix into multiple tiles. Say we have tiles like:

1, 2, 3
4, 5, 6
7, 8, 9

The order I traversal of the tiles would be [1], [2, 4], [3, 5, 7], [6, 8], [9]. The tiles in the same [] can be computed parallel. In each tile, the pixel is computed sequentially.

Question 5:

Below is the plot of two versions of parallel implementation on elapsed time vs. number of threads. It can be seen that the tiled parallel version is faster than the naïve parallel version. This benefits from the increasing cache hits, which could be examined by using "perf" tool. The best speedup is about 6. Notice that the naïve parallel version has speedup of 4. This is a pretty good improvement.