COL380 HW2

Shashwat Shivam
2016CS10328

1. Answers:-

1.1. The parallelization in the code is safe as the schedule is static and the number of iterations in both loops is fixed. This causes the same distribution of iterations to the threads for both the loops as the same type of scheduling is carried out.

When the number of iterations and the scheduling type is changed to dynamic, the parallelization is no more safe as we cannot guarantee that the thread which does the operations on some index in the first loop work on the same index in the second loop. The data dependence in the two loops with different threads without synchronization is therefore not parallelization safe.

1.2.

```
for ( int i =1; i<n ; i++) {
#pragma omp parallel for schedule(static) shared(a,j) private(j)
        for ( int j =0; j<n ; j++) {
                a [ i ∗n+j ] += a [ ( i −1)∗n+j ] ;
        }
}
```

The above is a parallel version of the code. This parallelization is efficient as we divide the task to update a row into parts which helps us take some speedup from the spatial locality.
Had we parallelized along each column, due to false sharing, the cache would have to be invalidated and re-fetched for each write, which would cause slow down of the program.

A faster implementation would be to have parallelization along column but assign each thread contiguous columns equal to the size of the cache line. This would prevent the false sharing and recreation of threads.

```
#pragma omp parallel for schedule(static,CACHE_LINE/sizeof(int))
shared(a,j) private(j)
for ( int i =1; i<n ; i++) {
        for ( int j =0; j<n ; j++) {
                a [ i ∗n+j ] += a [ ( i −1)∗n+j ] ;
        }
}
```

1.3.

```
// located inside a parallel region in function foo
#pragma omp atomic
        a++; // Modify a exclusively
// located inside a parallel region in function bar
```

```
#pragma omp critical
{
        b++; // Modify b exclusively
        a += b ; // Modify a exclusively
}
```

The corrected version of the code is written above.

In the function foo, the operation ++ is an atomic operation so it does not need a whole critical section, instead, the atomic construct can be used. Atomic is more efficient than a critical section.

In the function bar, to see consistent behaviour in function after updation of b, we should update a with the value of b without interference by other threads to prevent inconsistent values during the function call.

2. The outer loop shares the value of X and tmp between iterations. The inner loop shares the value of i from the outer loop and X. The outer loop is ideally suited for parallelization as this will prevent the creation of threads on each iteration of the outer loop.

   There is no read-write data dependency for outer loop as each loop needs to read X which remains constant. The inner loop shares the variable count.

   A parallel version of the program is as follows:-

```
void seq sort ( std::vector <unsigned int> &X) {
        unsigned int i , j , count , N=X.size( ) ;
        std::vector <unsigned int> tmp (N) ;
        #pragma    omp    parallel    for    schedule(static,CACHE_LINE/sizeof(int))
shared(X,tmp) private(i,j,count)
        for( i =0; i<N; i++){
                count =0;
                for ( j =0; j<N; j++)
                        if (X[ j ]<X[ i ] | | X[ j]==X[ i ] && j<i )
                        count++;
                        tmp [ count ]=X[ i ] ;
                }
        std::copy ( tmp.begin( ) , tmp.end( ) , X.begin( ) ) ;
}
```

   To prevent false sharing of tmp between loops, the chunk size of the number of iterations for each thread is given the number of elements of tmp on 1 cache line.

Speedup:- Outer loop will be parallelized. If we assume p number of threads. Loop takes n^2 / p order of instructions and copy command takes order n instructions. The speedup = T_s/T_p = p(n+1)/(n+p)

Efficiency: - = (n+1)/(n+p)

3.
```
global void reduce0 ( int ∗ g_idata , int ∗ g_odata ) {
        extern shared int sdata [ ] ;
        // each thread loads one element from global to shared mem
        unsigned int tid = threadIdx.x ;
        unsigned int i = blockIdx.x∗blockDim.x + threadIdx.x ;
        sdata[tid] = g_idata [ i ] ;
        // assuming warp dimension to be 32 do reduction on blocks of 32
        for ( unsigned int s =1; s < 32 ; s ∗= 2 ) {
                if ( tid % ( 2∗ s ) == 0 ) {
                        sdata [ tid ] += sdata [ tid + s ] ;
                }
        }
        // do reduction in shared mem
        for ( unsigned int s =32; s < blockDim.x ; s ∗= 2 ) {
                __syncthreads( ) ;
                if ( tid % ( 2∗ s ) == 0 ) {
                        sdata [ tid ] += sdata [ tid + s ] ;
                }
        }
        // write result for this block to global mem
        if ( tid == 0 ) g_odata[blockIdx.x] = sdata[0] ;
}
```

We add another loop in the code to operate until the warp dimension. This helps in increasing efficiency of code, because if we consider a threads in the same warp, they proceed in lockstep manner and therefore in operations between them no synchronization is required. The number of times __synchthreads() is called is greatly reduced. This saves a lot of overhead. We also move the __syncthreads() inside the loop to the top, because the last iteration does not need the threads to synchronize as we need only the value of one thread to write to output.

The two ways to avoid data race among threads of a block are by using the following synchronization primitives:-

- __threadfence_block() - This blocks the threads until all global and shared data accesses done before this instruction is accessible by all the threads of the block.
- __threadfence() - This blocks the threads until all the shared data accesses done before this instruction is accessible by all the threads of the device.

The function qualifiers available in CUDA are:-

- __global__ :- These functions are called from the host and executed on the device.
- __device__ :- These functions are called from a device and executed on the device itself.
- __host__ :- These functions are called from the host and executed on the host itself.

4. Future construct helps us to pass variables to other thread whose value will be required or requested in future. The thread does not need to busy wait for the other thread to complete its calculation and the original thread can get the value of the variable on request.
   Promises construct are implemented using future objects. A promise is passed to the other thread whose future object is kept by the original thread. The thread having the promise object can set the value of the variable when it's calculations are complete. After this, the future object can get the value of the associated variable.

   Asynchronous function calls are function calls which are run on other threads. These calls can be run lazily (deferred) depending on load or launched instantaneously. The variables between these threads can be passed using future and promise constructs. Async calls can also accept callbacks for calling when completed.

   Program:-

```
void fetcher_thread(promise<int> promise,int index)
{
        int val = fetch_index(index);
        promise.set_value(val);
}

int main()
{
        auto promise_obj = promise<int>();
        auto future_obj = promise.get_future();

        int index1 = 0, index2 = 1;

        async(std::launch::async,fetcher_thread,promise_obj,index2);

        int val1 = fetch_index(index1);
        int val2 = future_obj.get();
}
```

   In the above program, if it takes 2 seconds to fetch each index, by parallelizing the code we can fetch both indices in just 2 seconds instead of 4.

We call another function to fetch index asynchronously and pass it a promise object whose value can be fetched at any time from its respective future object.

These functions simplify thread creation and also remove the role of thread management and locks.