COL380 Homework-2

Pradyumna Meena

2016CS10375

Q 1

A. Static scheduling binds the same iterations to a thread across two for loops i.e. one thread will execute the same space of iterations in the given code if the total iteration space is of same size. However when we change the iteration space and the scheduling to dynamic then due to presence of nowait construct in the first for loop it might happen that in the second for loop the value being read is the old one and not the updated one. This was not an issue in static scheduling because the same thread would be working over a set of elements in both of the for loops and it will start with the second thread only when it is done with the first loop. However false sharing remains a common issue in both of the implementations. It might happen in the first loop depending on the array length and the cache line size. It can be prevented by using appropriate offset depending on the cache line size. Hence the parallelization implemented isn't safe for the given code.

B. Change the loop to the following

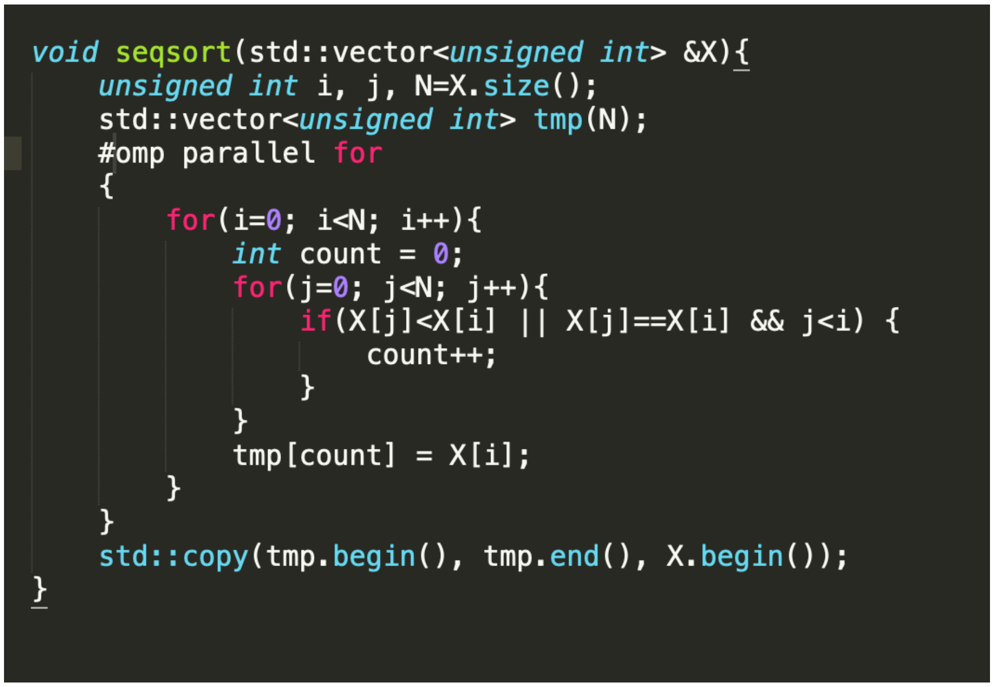
```
for (int i = 1; i < n; i++){
     for (int j = 0; j < n ; j++){
          a[i*n + j] += a[(i-1)*n + j];
     }
}
```

Now parallelize the outer loop. The advantage it offers over the previous implementation is that in the previous one in every iteration of inner loop the element to be added was n indices away but each iteration also requires a element which is n indices away i.e. for i_1 iteration we need element with index i_1*n + j and in the just next iteration we need element with index i_1*n + j + n. The above implementation however only advances by one index in each iteration of inner loop. This means less refetching if n is quite large. Above implementation essentially exploits the contiguity property of array that if i^{th} element is to be fetched it might as well fetch nearby elements. This will help in increasing efficiency by a considerable amount.

C. In openmp all unnamed critical sections are considered identical. Hence if one thread is in some critical section then no other thread can enter any other critical section since all the critical sections would be associated with the same lock. Hence one modification would be to assign different name to each of the critical section so that two threads can simultaneously work on two critical sections. Moreover since both instructions in the critical section are essentially atomic instructions. Using critical causes much higher overheads than the atomic construct because acquiring and releasing locks. Therefore instead of critical construct used we should use atomic in the given code to reduce the overheads since it has no locks involved.

Q 2

The algorithm determines the position of any element in the sorted vector by counting the number of elements which are smaller than it or those which are equal to it as well as are present before it. This clearly has data dependency across various indices of the vector. For assigning value corresponding to any index we need to look at all the other elements of the vector. The outer loop consists of the memory update part which could give rise to false sharing.
If we parallelize the inner loop then the memory update also should be included in the parallel region because no data sharing construct enables us to pass the value of private variable to pass outside the parallel region except for lastprivate (count variable for temp vector). However if we use lastprivate then in the count variable only the value corresponding to the thread which was last to update the count variable would be reflected and hence we would end up updating only one index in the temp vector. If we use private or firstprivate then the correct value would be lost outside the parallel region and hence of no use. That means we need to include the memory update inside the parallel region if we want to parallelize the inner loop. Another approach would be to parallelize the outer loop and keeping the count variable private to each thread. Both of the approaches (inner loop and firstprivate, outer loop and private) have the same amount of overheads and performance.



```
void seqsort(std::vector<unsigned int> &X){
    unsigned int i, j, N=X.size();
    std::vector<unsigned int> tmp(N);
    #omp parallel for
    {
        for(i=0; i<N; i++){
            int 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());
}
```

Q 3

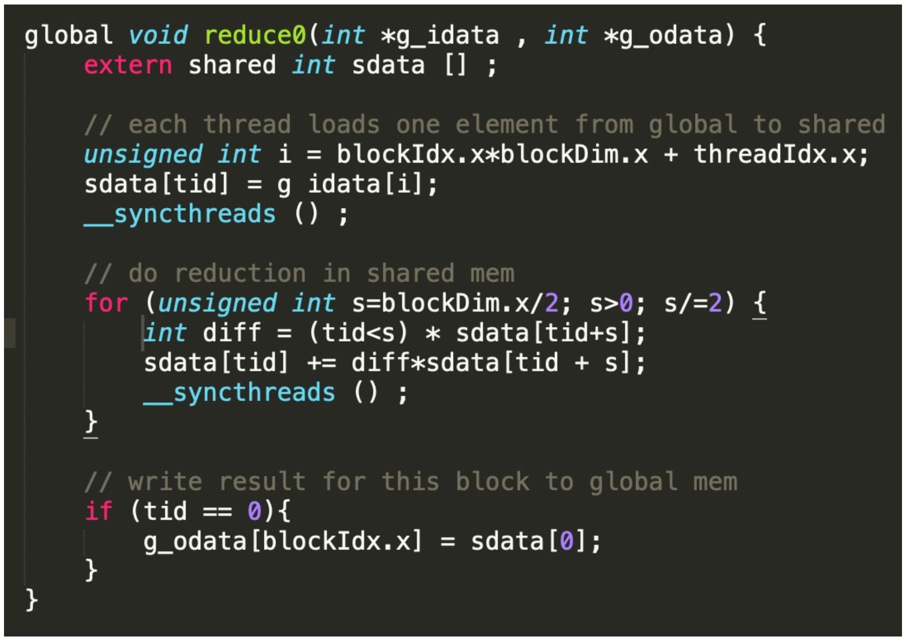
Branch divergence is arising from the if condition in the for loop because because not every thread would be executing the complete code. Few of them might directly go to the end of the conditional block.

Memory bank conflicts arise when two threads try to access same bank. NVIDIA gpus have 16 banks which are interleaved with granularity of 32 bits or 4 bytes. Whenever two threads are trying to access the same bank their requests are essentially serialized and hence the parallelization efficiency is lost. The banks are distributed in the following way

Bank -> 1 | 2 | 3 | 4 ……
Byte -> 0 1 2 3 | 4 5 6 7 | 8 9 10 11 | 12 13 14 15…..
64 65 66 67| 68 69 70 71 | 72 73 74 75 | 76 77 78 79…..

Barrier Overheads occur due to recursive calling of syncthreads in every loop iteration and since it waits for all of the threads to arrive at that point causes delay in execution.

Diverging branches are avoided by using diff variable which captures the if condition in itself and hence all threads end up doing the same amount of work. Memory conflicts are reduced by changing the initial value of loop variable and the manner in which it is updated.



```
global void reduce0(int *g_idata , int *g_odata) {
    extern shared int sdata [] ;

    // each thread loads one element from global to shared
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads () ;

    // do reduction in shared mem
    for (unsigned int s=blockDim.x/2; s>0; s/=2) {
        int diff = (tid<s) * sdata[tid+s];
        sdata[tid] += diff*sdata[tid + s];
        __syncthreads () ;
    }

    // write result for this block to global mem
    if (tid == 0){
        g_odata[blockIdx.x] = sdata[0];
    }
}
```

To avoid data races among threads of the block it must be ensured that they all move in a lock-step manner. Hence we must take care that all thread do same amount of work and also access same memory region at a given time.
Below are the function qualifiers used in CUDA.C

1. **__device__** : used to declare a function which would be executed on the device and is also called from the device itself
2. **__global__** : used to declare a function which would be executed on the device and is called by the host
3. **__host__** : used to declare a function which runs on the host and is also called by the host itself

Q 4

Normally when a function is called the caller halts or waits until the called function completes its execution and then it proceeds. This was the case with synchronous functions. However in

case of asynchronous functions the caller does not waits for the called function. Instead it proceeds its execution with the belief that it will be signalled after the called function has completed its execution.
Promise is a type of object which holds a value that it promises to give when an object of type future (possibly in another thread) tries to access it some point later in the execution. Future is type of object whose value would be defined in the future. This offers another way to have synchronization amongst the threads. It would be more clear in the following examples.

```
#include <iostream>
#include <thread>
#include <future>
#include <functional>
using namespace std;

void print_int (future<int>& future_param) {
  int x = future_param.get();
  cout << "value supplied: " << x << '\n';
}

int main ()
{
 // creating promise object
  promise<int> promise_obj;

 // promising to give the value
  future<int> future_obj = promise_obj.get_future();

 // passing thread to some other thread
  thread th1 (print_int,ref(future_obj));

 // fulfill promise by supplying value
  promise_obj.set_value (380);

 // (synchronizes with getting the future)
  th1.join();
  return 0;
}
```

The promise object and future object made a pact that promise object promises to supply value to the future object in the future. We see that a thread is created without supplying any value. Afterwards the promise keeps it's end of the bargain and value is supplied to the object and is used in the thread th1.

```
#include <iostream>
#include <thread>
#include <future>
#include <functional>
using namespace std;

int factorial(future<int> f) {
     int N = f.get();
     f.get();
     int res = 1;
     for(int i = 2;i <=N;i++){
          res*=i;
     }
     cout << "Result computed in child thread is: " << res << endl;
     return res;
}

int main() {
     promise<int> p;
     future<int> f = p.get_future();
     future<int> future_fact = std::async(std::launch::async,
factorial, std::ref(f));
     p.set_value(6);
     cout << "My child sended: " << future_fact.get() << endl;
     return 0;
}
```

Now if say we don't supply value that is the promise is broken then the execution will throw an error saying broken promise. The future f is the future of p i.e. it would be signalled (according to the promise made by p) when p will have a value. Once the future f is set then future future_fact would be signalled and the factorial function would be called.