OpenMP: First And Last Private

Previously, you saw that private variables are completely separate from any variables by the same name in the surrounding scope. However, there are two cases where you may want some storage association between a private variable and a global counterpart.

First of all, private variables are created with an undefined value. You can force their initialization with

```
  int t=2;
#pragma omp parallel firstprivate(t)
  {
    t += f( omp_get_thread_num() );
    g(t);
  }
```

The variable t behaves like a private variable, except that it is initialized to the outside value.

Secondly, you may want a private value to be preserved to the environment outside the parallel region. This really only makes sense in one case, where you preserve a private variable from the last iteration of a parallel loop, or the last section in a section construct. This is done with

```
#pragma omp parallel for \
        lastprivate(tmp)
for (i=0; i<N; i+) {
  tmp = ......
  x[i] = .... tmp ....
}
..... tmp ....
```

Q1

Debug the code downloaded from [P7Q1] that computes the [Mandelbrot set]. Use firstprivate to initialize the private variable. Refer to this [slide] for detailed explanation.

Instructions:

1) Identify private variables in the parallel region.
2) Pass correct argument in testpoint function
3) Identify atomic function in the testpoint function

Output:

```
Area of Mandlebrot set =    1.51211812 +/-    0.00151212
Correct answer should be around 1.510659
```

Consider simple list traversal

Given what we've covered about OpenMP, how would you process this loop in Parallel?

```
p=head;
while(p){
      process(p);
      p=p->next;
}
```

Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compiler time. While loops are not covered.

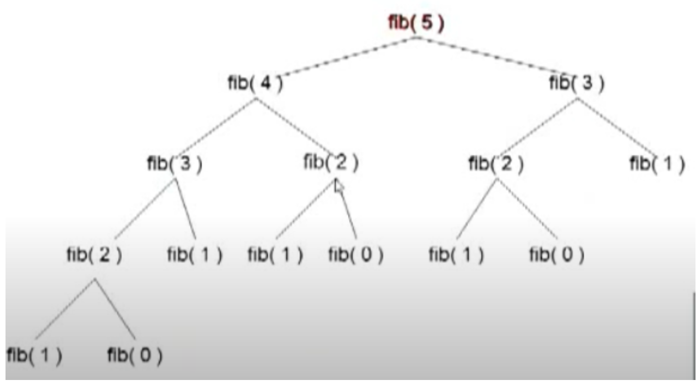
Question 2:

Consider the program P7Q2

Traverses a linked list computing a sequence of ***Fibonacci numbers*** at each node.

Parallelize this program using constructs described so far.

You may evaluate the performances using different scheduling types.



Instructions:

- Construct codes for running parallel processes on calculating the fibonacci numbers.
- Print consumption time for serial processing and parallel processing.

Reference Output: when PS = 35,

```
Microsoft Visual Studio Debug Console
Process linked list
  Each linked list node will be processed by function 'processwork()'
  Each ll node will compute 5 fibonacci numbers beginning with 35
Serial Compute Time: 8.243098 seconds
 8 threads
35 : 9227465
36 : 14930352
37 : 24157817
38 : 39088169
39 : 63245986
40 : 102334155
Parellel Compute Time: 4.799399 seconds
```

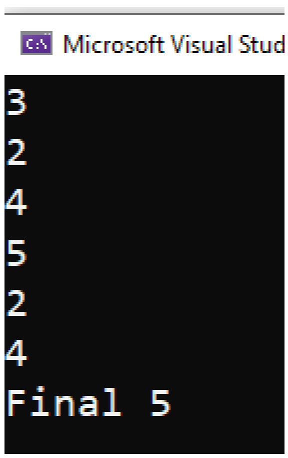
Question 3:

Debug the program P7Q3. So, it displays the initialized values in the parallel region.

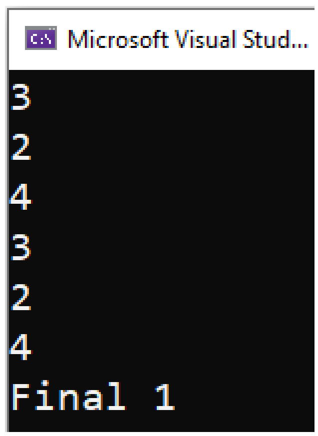
- Add *a* = *a*+*b* in the parallel region. Display *a* outside of parallel region
- Notice the difference between using firstprivate and without using firstprivate.

Reference answer: assume that ***omp_set_num_threads(2)***

Without firstprivate



With firstprivate



Basic Matrix Multiplication

Question 4:

Implement a basic dense matrix multiplication routine. Optimizations such as tiling and usage of shared memory are not required for this question.

Edit the code in the code tab to perform the following:

- allocate device memory
  cudaMalloc((void**)&a_d, size);
- copy host memory to device
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
  matrixMultiplySimple << <grid, block >> >(a_d, b_d, c_d, width);
- copy results from device to host
  cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);
- deallocate device memory
  cudaFree(a_d);

Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

Obtain the code P7Q4.

Output:

Microsoft Visual Studio Debug Console

```
Number of threads: 121 (11x11)
Number of blocks: 361 (19x19)
Time to calculate results on GPU: 0.172448 ms
Time to calculate results on CPU: 0.001536 ms
```

* Adjust the THREADS_PER_BLOCK and width in main() function to achieve GPU time lesser than CPU time.

```
Number of threads: 4096 (64x64)
Number of blocks: 169 (13x13)
Time to calculate results on GPU: 0.000960 ms
Time to calculate results on CPU: 0.002240 ms
Error: CPU and GPU results do not match
```

Tiled Matrix Multiplication

Question 5:

Edit the code P7Q5 to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory
- free memories
- implement the matrix-matrix multiplication routine using shared memory and tiling

Instructions about where to place each part of the code is demarcated by the
//@@ comment lines.

Example output:

```
Enter m n n k :
100 100 100 100
GPU time= 0.135072 ms
CPU time= 3.000000 ms
Results are equal!
```