

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

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

- Q1 Debug the code downloaded from [here](#) 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
```

P7 - Introduction to Parallel Computing

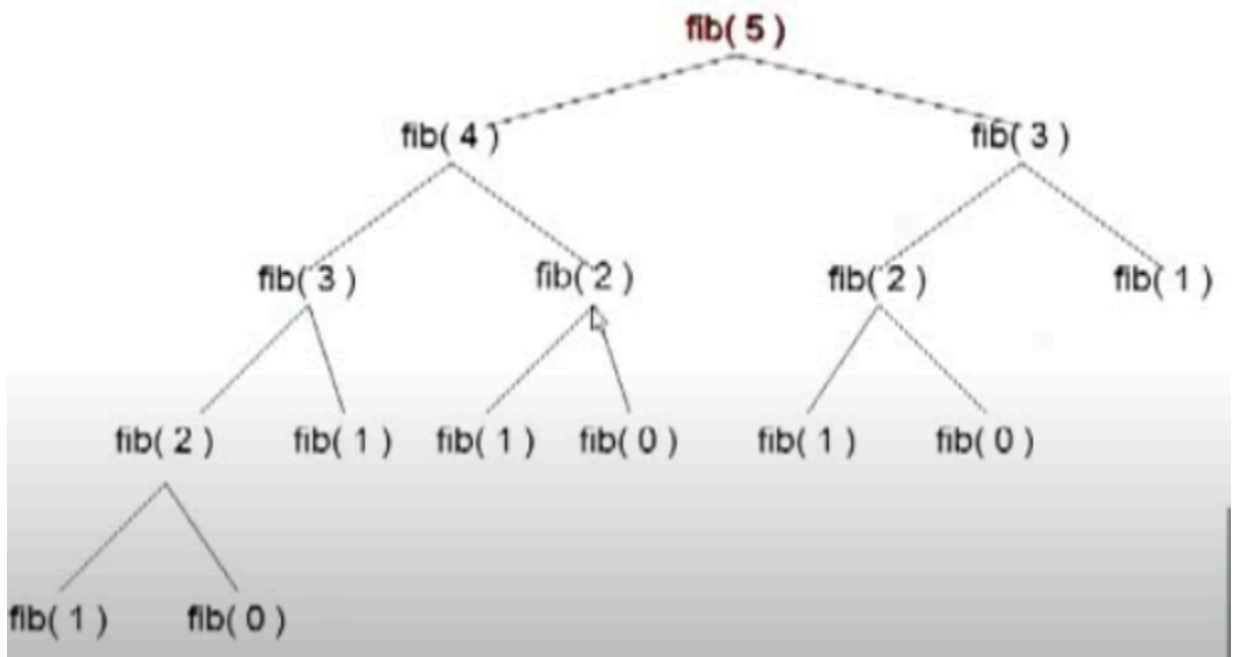
Consider simple list traversal

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.

- Q2 Consider the program [P7Q2.c](#)
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](#).



- Q3 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`.

Basic Matrix Multiplication

Q4 Implement a basic dense matrix multiplication routine. Optimizations such as tiling and usage of shared memory are not required for this question.

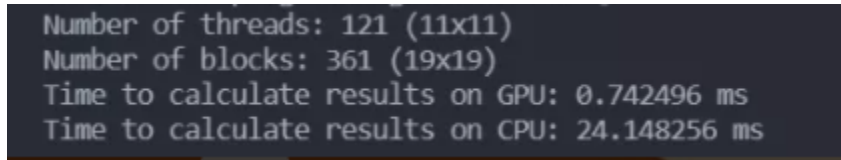
Edit the code in the code tab to perform the following:

- 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`.

Download the code [here](#).

Output:



```
Number of threads: 121 (11x11)
Number of blocks: 361 (19x19)
Time to calculate results on GPU: 0.742496 ms
Time to calculate results on CPU: 24.148256 ms
```

Tiled Matrix Multiplication

Q5 Edit the code [here](#) 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
 - 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`.