# 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:**

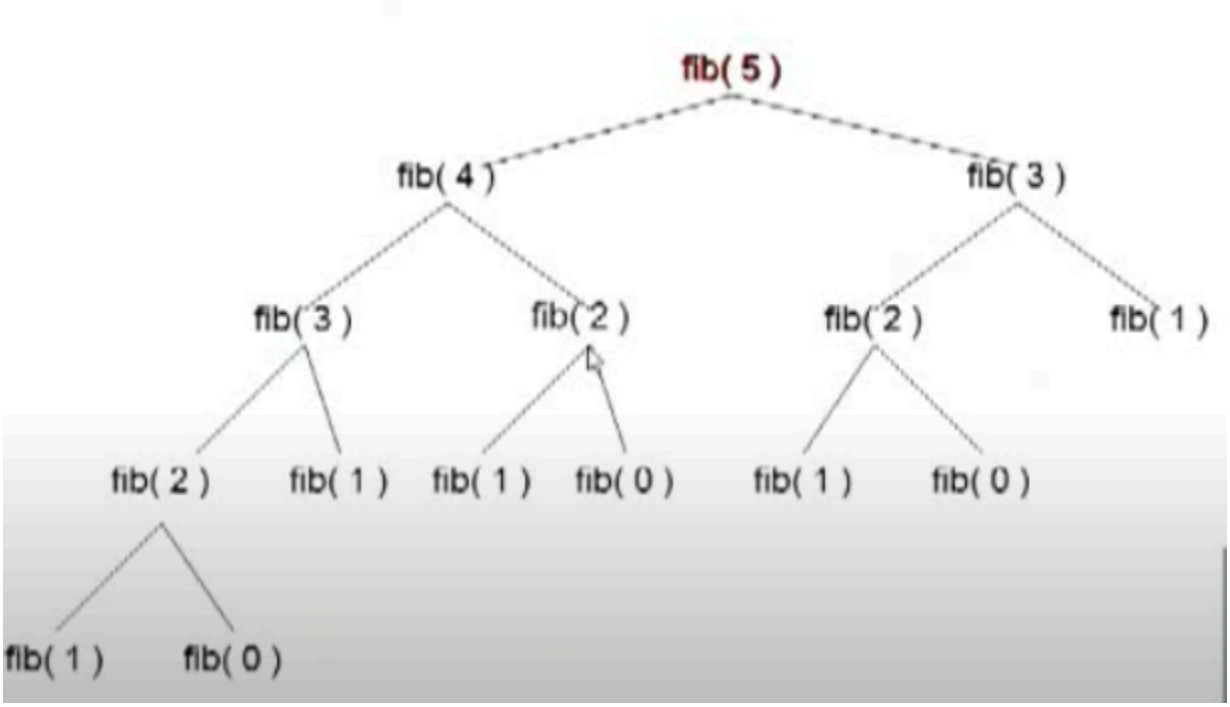
# P7 - Introduction to Parallel Computing

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.



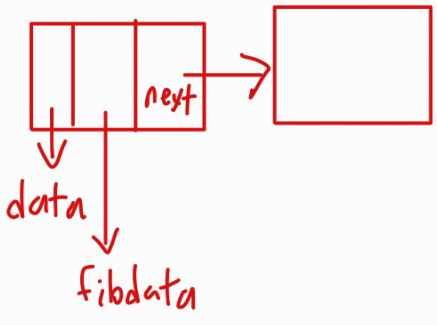

Q3 Debug the program P7Q3. So, it displays the initialized values in the parallel region.

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

02_Fibonacci.cpp

```cpp
#include <stdlib.h> // malloc, free
#include <iostream> // cout
#include <omp.h>

using namespace std;

#define N 5   // number of elements in the linked list
#define FS 38 // fibonacci number to be computed
```

```cpp
struct node
{
    int data;          // n value in the fibonacci sequence, eg. 38, 39, 40, 41, 42
    int fibdata;       // result of the fibonacci computation, eg. 39088169
    struct node *next; // pointer to the next node in the linked list
};
```




```cpp
int fib(int n)
{
    int x, y;
    if (n < 2)
    {
        return (n);
    }
    else
    {
        x = fib(n - 1);
        y = fib(n - 2);
        return (x + y);
    }
}
```

→ if n=0, return 0
→ if n=1, return 1

What is fibonacci

n=10

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

$f_0$ $f_1$ $f_2$ $f_3$ $f_4$ $f_5$ $f_6$ $f_7$ $f_8$ $f_9$ $f_{10}$

$f_n = f_{n-1} + f_{n-2}$

eg: $f_{10} = f_9 + f_8$

$= 34 + 21$

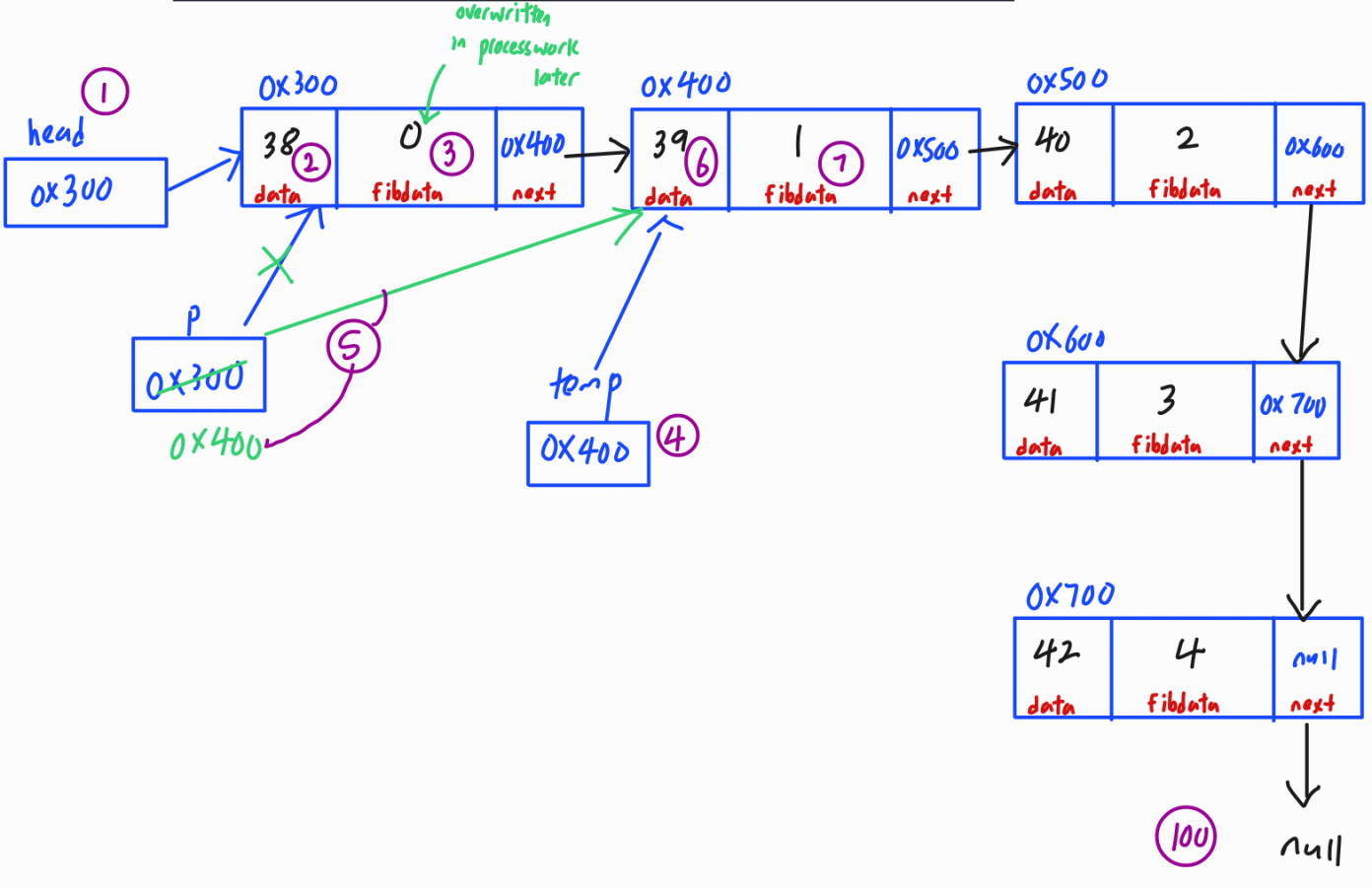
$= 55$

```
38 : 39088169 = 37 x 113 x 9349
39 : 63245986 = 2 x 233 x 135721
40 : 102334155 = 3 x 5 x 7 x 11 x 41 x 2161
41 : 165580141 = 2789 x 59369
42 : 267914296 = 2^3 x 13 x 29 x 211 x 421
```

https://r-knott.surrey.ac.uk/fibonacci/fibtable.html

```c
void processwork(struct node *p)
{
    int n;
    n = p->data;          // get the n value(data) from the node
    p->fibdata = fib(n); // compute the fibonacci value and put it into fibdata in the node
}
```

```c
struct node *init_list()
{
    int i;
    struct node *head = NULL;
    struct node *temp = NULL;
    struct node *p;

    head = (struct node *)malloc(sizeof(struct node));
    p = head;    (1)
    p->data = FS; (2) // dereferencing put into the node
    p->fibdata = 0;(3) // deferencing put into the node
    for (i = 0; i < N - 1; i++)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        p->next = temp; (4)
        p = temp; (5)
        p->data = FS + i + 1;(6)
        p->fibdata = i + 1;  (7)
    }
    p->next = NULL; (100)
    return head;
}
```

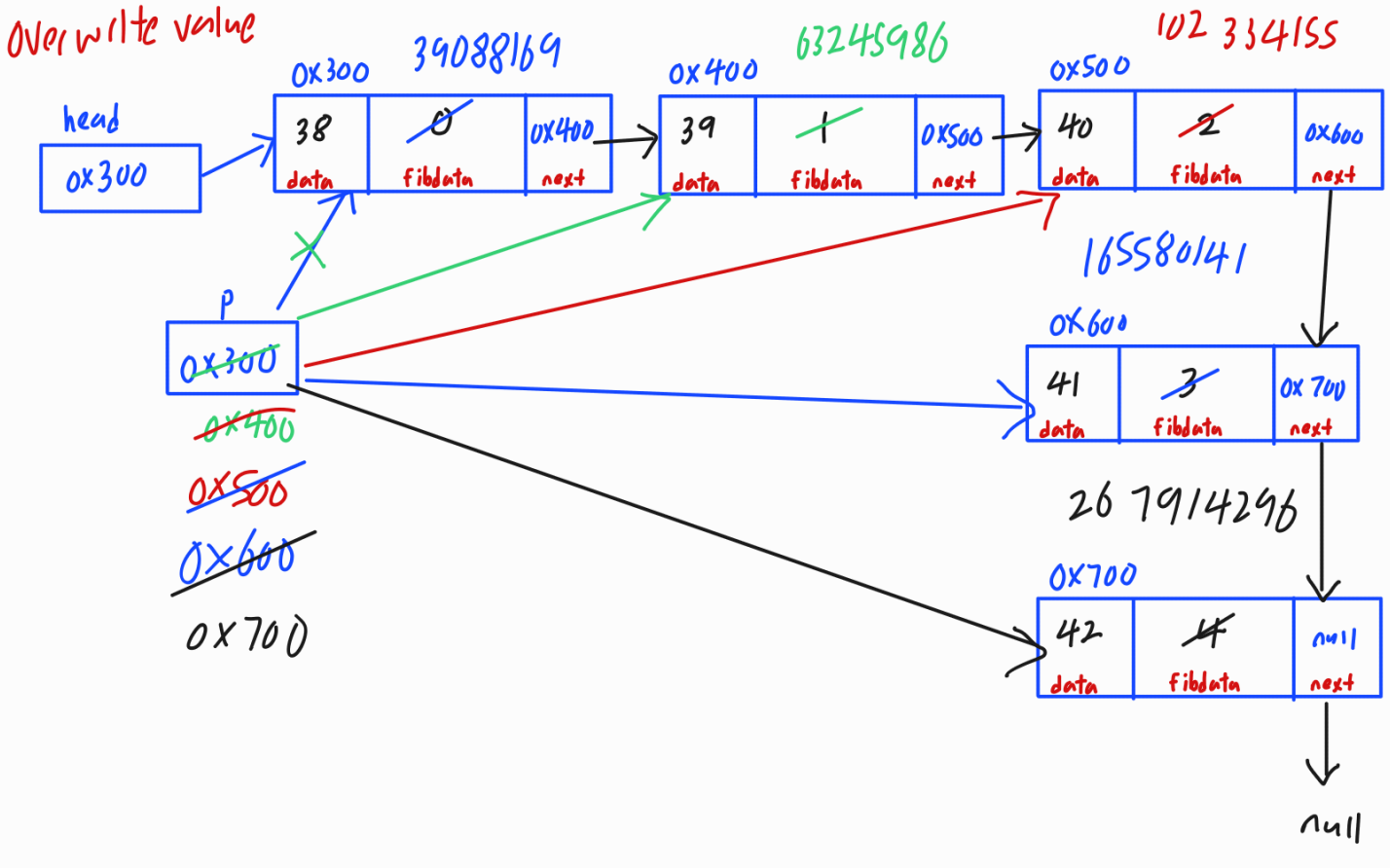


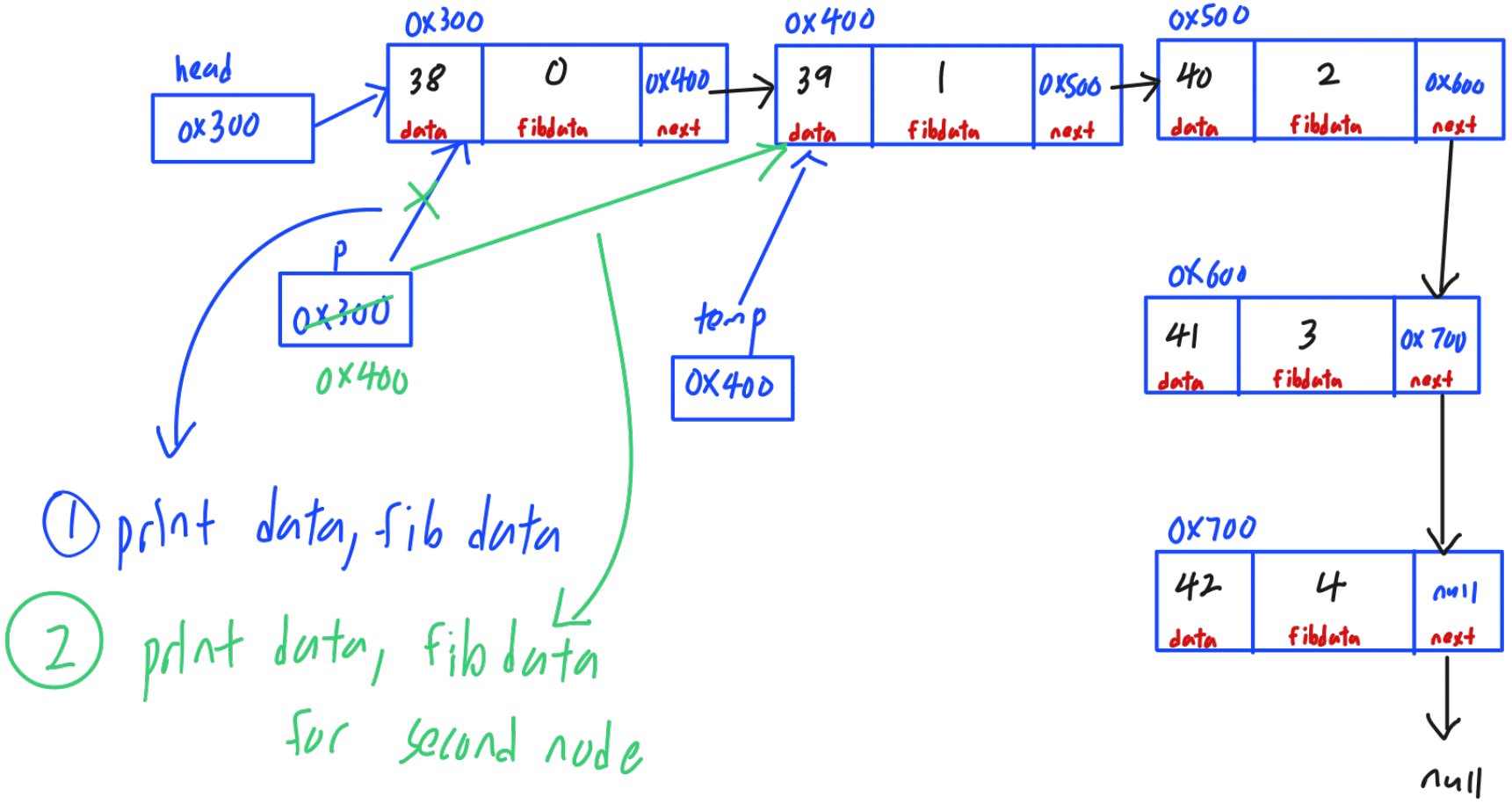
```c
double calc_fib_serial()
{
    double start, end;
    struct node *p = NULL;
    struct node *temp = NULL;
    struct node *head = NULL;

    printf("Process linked list\n");
    printf("  Each linked list node will be processed by function 'processwork()'\n");
    printf("  Each ll node will compute %d fibonacci numbers beginning with %d\n", N, FS);

    p = init_list();
    head = p;

    start = omp_get_wtime();

    while (p != NULL)
    {
        processwork(p);
        p = p->next;
    }

    end = omp_get_wtime();
    p = head;
    while (p != NULL)
    {
        printf("%d : %d\n", p->data, p->fibdata);
        temp = p->next;
        free(p);
        p = temp;
    }
    free(p);

    printf("Compute Time: %f seconds\n", end - start);

    return end - start;
}
```

overwrite value

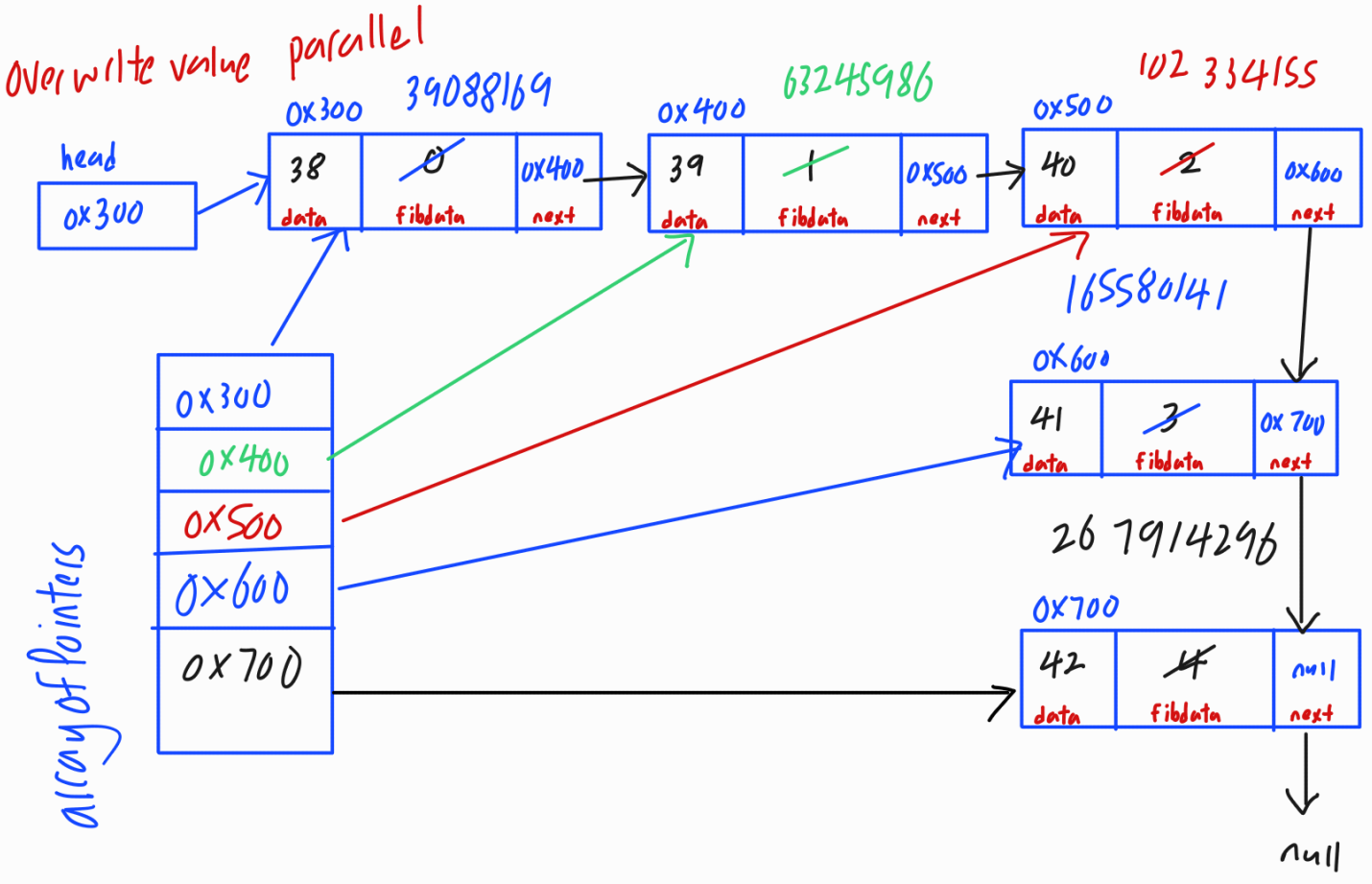
① ② ③ ④ ⑤

print value




0x300
38
data
0
fibdata
0x400
next
0x400
39
data
1
fibdata
0x500
next
0x500
40
data
2
fibdata
0x600
next
0x600
41
data
3
fibdata
0x700
next
0x700
42
data
4
fibdata
null
next
null
head
0x300
p
0x300
0x400
temp
0x400
1 print data, fib data
2 print data, fib data
for second node

```c
double calc_fib_parallel()
{
    double start, end;
    struct node *p = NULL;
    struct node *temp = NULL;
    struct node *head = NULL;      You, 3 hours ago • Complete Fibonancci OpenMP
    struct node **arrayOfPointers;

    printf("Process linked list\n");
    printf("  Each linked list node will be processed by function 'processwork()'\n");
    printf("  Each ll node will compute %d fibonacci numbers beginning with %d\n", N, FS);

    p = init_list();
    head = p;

    arrayOfPointers = (struct node **)malloc(N * sizeof(struct node *));

    start = omp_get_wtime();

    int i = 0;
    p = head;
    while (p != NULL)
    {
        arrayOfPointers[i++] = p;
        p = p->next;
    }

#pragma omp parallel for
    for (i = 0; i < N; i++)
    {
        processwork(arrayOfPointers[i]);
    }

    end = omp_get_wtime();
    p = head;
    while (p != NULL)
    {
        printf("%d : %d\n", p->data, p->fibdata);
        temp = p->next;
        free(p);
        p = temp;
    }
    free(p);

    printf("Compute Time: %f seconds\n", end - start);

    return end - start;
}
```



output: 02_Fibonacci.cpp

```
└─ ./02_Fibonacci
Process linked list
  Each linked list node will be processed by function 'processwork()'
  Each ll node will compute 5 fibonacci numbers beginning with 38
38 : 39088169
39 : 63245986
40 : 102334155
41 : 165580141
42 : 267914296
Compute Time: 4.837503 seconds
Process linked list
  Each linked list node will be processed by function 'processwork()'
  Each ll node will compute 5 fibonacci numbers beginning with 38
38 : 39088169
39 : 63245986
40 : 102334155
41 : 165580141
42 : 267914296
Compute Time: 2.370524 seconds
The Performance gain is 2.04069
```

03_FirstPrivate.cpp

```cpp
int main()
{
    int a = 1;
    int b = 2;
    int c = 4;

    // Without firstprivate
    printf("Without firstprivate:\n");
#pragma omp parallel num_threads(4)
    {
        printf("Thread %d: Initial values: a=%d, b=%d, c=%d\n", omp_get_thread_num(), a, b, c);
        a = a + b;
        printf("Thread %d: Updated a=%d\n", omp_get_thread_num(), a);
    }
    printf("Outside parallel region: a=%d\n", a);

    // Reset a
    a = 1;

    // With firstprivate
    printf("\nWith firstprivate:\n");
#pragma omp parallel firstprivate(a) num_threads(4)
    {
        printf("Thread %d: Initial values: a=%d, b=%d, c=%d\n", omp_get_thread_num(), a, b, c);
        a = a + b;
        printf("Thread %d: Updated a=%d\n", omp_get_thread_num(), a);
    }
    printf("Outside parallel region: a=%d\n", a);

    return 0;
}
```

```
└ ./03_FirstPrivate
Without firstprivate:
Thread 0: Initial values: a=1, b=2, c=4
Thread 0: Updated a=3
Thread 3: Initial values: a=1, b=2, c=4
Thread 3: Updated a=5
Thread 1: Initial values: a=1, b=2, c=4
Thread 1: Updated a=7
Thread 2: Initial values: a=1, b=2, c=4
Thread 2: Updated a=9
Outside parallel region: a=9

With firstprivate:
Thread 2: Initial values: a=1, b=2, c=4
Thread 2: Updated a=3
Thread 1: Initial values: a=1, b=2, c=4
Thread 1: Updated a=3
Thread 3: Initial values: a=1, b=2, c=4
Thread 3: Updated a=3
Thread 0: Initial values: a=1, b=2, c=4
Thread 0: Updated a=3
Outside parallel region: a=1
```

# 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.

```cpp
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <stdio.h>
#include <malloc.h>
#include <math.h>

#define THREADS_PER_BLOCK 128

void matrixMultiplyCPU(float *a, float *b, float *c, int width)
{
    float result;

    for (int row = 0; row < width; row++)
    {
        for (int col = 0; col < width; col++)
        {
            result = 0;
            for (int k = 0; k < width; k++)
            {
                result += a[row * width + k] * b[k * width + col];
            }
            c[row * width + col] = result;
        }
    }
}

__global__ void matrixMultiplySimple(float *a, float *b, float *c, int width)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    float result = 0;

    if (col < width && row < width)
    {
        for (int k = 0; k < width; k++)
        {
            result += a[row * width + k] * b[k * width + col];
        }
        c[row * width + col] = result;
    }
}
```

answer

```
int main()
{
    int width = 200; // Define width of square matrix
    // Initialise grid and block variables
    int sqrtThreads = (int)sqrt(THREADS_PER_BLOCK);
    int nBlocks = width / sqrtThreads;
    if (width % sqrtThreads != 0)
    { // Add an extra block if necessary
        nBlocks++;
    }
    dim3 grid(nBlocks, nBlocks, 1);
    dim3 block(sqrtThreads, sqrtThreads, 1); // Max number of threads per block

    // Initialise host pointers (dynamically allocated memory) and device pointers
    // Note: _h is matrix for the host (CPU) and _d is matrix for device (GPU)
    float *a_h;
    float *b_h;
    float *c_h; // GPU (computed in parallel) results
    float *d_h; // CPU (computed in serial) results
    float *a_d; // Device memory for A matrix
    float *b_d; // Device memory for B matrix
    float *c_d; // Device memory for C matrix

    int size; // Number of bytes required by arrays

    // Create timer
    cudaEvent_t start;
    cudaEvent_t stop;
    float elapsed1, elapsed2; //, elapsed3;

    // Print out information about blocks and threads
    printf("Number of threads: %i (%ix%i)\n", block.x * block.y, block.x, block.y);
    printf("Number of blocks: %i (%ix%i)\n", grid.x * grid.y, grid.x, grid.y);

    // Dynamically allocate host memory
    size = width * width * sizeof(float);

    a_h = (float *)malloc(size);
    b_h = (float *)malloc(size);
    c_h = (float *)malloc(size);
    d_h = (float *)malloc(size);

    // Load host arrays with data
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < width; j++)
        {
            a_h[i * width + j] = (float)i;
            b_h[i * width + j] = (float)i;
        }
    }

    //@@ Allocate device memory for a_d, b_d and c_d matrices
    cudaMalloc((void **)&a_d, size);
    cudaMalloc((void **)&b_d, size);
    cudaMalloc((void **)&c_d, size);

    //@@ Copy host memory to device memory from a_h and, b_h to a_d and b_d
    cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, size, cudaMemcpyHostToDevice);
```

answer

```
    // Start timer for GPU
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //@@ Launch kernel (GPU) passing in a_d, b_d, and c_d matrices
    matrixMultiplySimple<<<grid, block>>>(a_d, b_d, c_d, width);

    // Stop timer
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsed1, start, stop);

    // Print execution time
    printf("Time to calculate results on GPU: %f ms\n", elapsed1);

    //@@ Copy results to host from c_d to c_h matrix
    cudaMemcpy(c_h, c_d, size, cudaMemcpyDeviceToHost);

    // Start timer for CPU
    cudaEventRecord(start, 0);

    // Launch CPU code
    matrixMultiplyCPU(a_h, b_h, d_h, width);

    // Stop timer
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsed2, start, stop);

    // Print execution time
    printf("Time to calculate results on CPU: %f ms\n", elapsed2);

    // Compare results
    for (int i = 0; i < width * width; i++)
    {
        if (fabs(c_h[i] - d_h[i]) > 1e-5)
        {
            printf("Error: CPU and GPU results do not match at index number %d\n", i);
            break;
        }
    }

    //@@ Free memory
    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(c_d);
    free(a_h);
    free(b_h);
    free(c_h);
    free(d_h);

    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    return 0;
}
```

answer

answer

answer

output:

```
└─ ./04_BasicMatrixMultiplication
Number of threads: 121 (11x11)
Number of blocks: 361 (19x19)
Time to calculate results on GPU: 4.031168 ms
Time to calculate results on CPU: 0.002048 ms
```