CS 61C
Fall 2018

Parallelism, SDS

Discussion 9: October 22, 2018

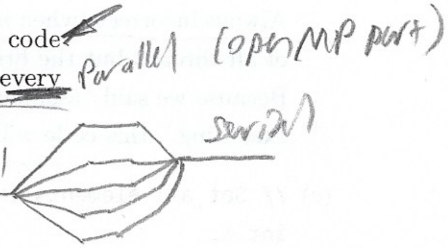
★ False sharing: different threads on different processors modify variables residing on the same cache line — invalidating the other threads cache block

like cache blocking but for threads.

1 Thread-Level Parallelism

As powerful as data level parallelization is, it can be quite inflexible, as not all applications have data that can be vectorized. Multithreading, or running a single piece of software on multiple hardware threads, is much more powerful and versatile. OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

The `parallel` directive indicates that each thread should run a copy of the code within the block. If a for loop is put within the block, **every** thread will run every iteration of the for loop.



```
#pragma omp parallel {
    ...
}
```

← generic format

The `parallel for` directive will split up iterations of a for loop over various threads. Every thread will run **different** iterations of the for loop. The following two code snippets are equivalent.

```
#pragma omp parallel for                #pragma omp parallel {
for (int i = 0; i < n; i++) {           #pragma omp for
    ...                                     for (int i =0; i < n; i++) { ... }
}                                       }
```

{ should be on new line.

There are two functions you can call that may be useful to you:

- `int omp_get_thread_num()` will return the number of the thread executing the code
- `int omp_get_num_threads()` will return the number of total hardware threads executing the code

★ You can manually chunk the for loop if you don't want to use omp parallel for.
↳ could be better if you have a chunking of inconsistent data.

1.1 For each question below, state and justify whether the program is **sometimes incorrect**, **always incorrect**, **slower than serial**, **faster than serial**, or **none of the above**. Assume the default number of threads is greater than 1. Assume no thread will complete before another thread starts executing. Assume `arr` is an `int[]` of length `n`.

(a)

```
// Set element i of arr to i
#pragma omp parallel
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

Every thread will run this for loop so will finish w/ slowest thread so slower than serial.

Since all threads set each item to the same value, we do not have incorrect answers from false sharing

Slower than serial: There is no **for** directive, so every thread executes this loop in its entirety. n threads running n loops at the same time will actually execute in the same time as 1 thread running 1 loop. Despite the possibility of false sharing, the values should all be correct at the end of the loop. Furthermore, the existence of parallel overhead due to the extra number of threads could slow down the execution time.

(b)
```
// Set arr to be an array of Fibonacci numbers.
arr[0] = 0;
arr[1] = 1;
#pragma omp parallel for
for (int i = 2; i < n; i++)
    arr[i] = arr[i-1] + arr[i - 2];
```

auto chunking.

Data dependency so when it gets chunks, it will be using incorrect values,

Always incorrect (when $n > 4$): Loop has data dependencies, so the calculation of all threads but the first one will depend on data from the previous thread. Because we said "assume no thread will complete before another thread starts executing," this code will always read incorrect values.

(c)
```
// Set all elements in arr to 0;
int i;
#pragma omp parallel for
for (i = 0; i < n; i++)
    arr[i] = 0;
```

auto chunking so each thread can work on a chunk at a time.

deterministically sets items w/o needing to use any prev calc data,

Faster than serial: The **for** directive actually automatically makes loop variables (such as the index) private, so this will work properly. The **for** directive splits up the iterations of the loop into continuous chunks for each thread, so there will be no data dependencies or false sharing.

1.2 What potential issue can arise from this code?

```
// Decrements element i of arr. n is a multiple of omp_get_num_threads()
#pragma omp parallel
{
int threadCount = omp_get_num_threads();
int myThread = omp_get_thread_num();
for (int i = 0; i < n; i++) {
    if (i % threadCount == myThread) arr[i] *= arr[i];
    }
}
```

Say had 4 core cpu, each thread would load in a part of the for loop but would be updating possibly the same cache block.

False sharing arises because different threads can modify elements located in the same memory block simultaneously. This is a problem because some threads may have incorrect values in their cache block when they modify the value arr[i], invalidating the cache block.

1.3

```
// Assume n holds the length of arr
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        product *= arr[n];
    }
    return product;
}
```

(a) What is wrong with this code?

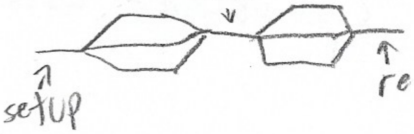
The code has the shared variable product.

cause two threads may try to update product at the same time instead of sequentially.

(b) Fix the code using **#pragma** omp critical

```
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        #pragma omp critical
        product *= arr[i];
    }
    return product;
}
```

adding the critical section sequentalizes it causing each thread to take its turn updating it



(c) Fix the code using **#pragma** omp reduction(operation: var).

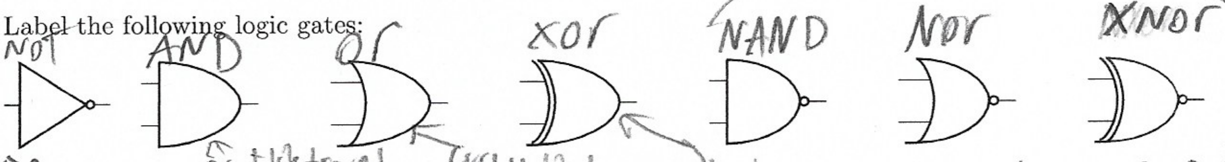
```
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for reduction(*: product)
    for (i = 0; i < n; i++) {
        product *= arr[i];
    }
    return product;
}
```

This prevents the need of a critical/sequential section

2 Logic Gates

same shape as and, or, xor but w/ circle at end

2.1 Label the following logic gates:



triangle w/ circle at end

flat like trand

curves like trand

double curved like trand

NOT, AND, OR, XOR, NAND, NOR, XNOR

circle means the output is inverted.

2.2 Convert the following to boolean expressions:

(a) NAND

NAND		
0	0	1
0	1	1
1	0	1
1	1	0

$\bar{A}\bar{B} + \bar{A}B + A\bar{B}$

(b) XOR

xor

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

$\bar{A}B + A\bar{B}$

(c) XNOR

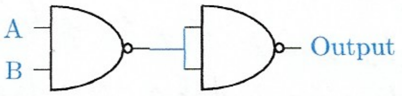
$\bar{A}\bar{B} + AB$

→

ab		xor out
0	0	1
0	1	0
1	0	0
1	1	1

NAND can flip input if you put it to both terminals.

2.3 Create an AND gate using only NAND gates.



Want: AB from $\bar{A}\bar{B}+\bar{A}B+A\bar{B}$

Nand again

Nand			
a	b	out	
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

2.4 How many different two-input logic gates can there be? How many n-input logic gates?

A truth table with n inputs has 2^n rows. Each logic gate has a 0 or a 1 at each of these rows. Imagining a function as a 2^n-bit number, we count 2^{2^n} total functions, or 16 in the case of $n = 2$.

2^n rows b/c for each input there is 2 possibilities compounded with all the other inputs $(2 \cdot 2 \cdot 2 \cdot \cdots)$ a 2 for each input