False Sharing: different threads on different cores modify different memory addresses residing on the same cache line — invalidating the other threads cache line.

True Sharing: Different cores write to the same address causing the cache line to be ping ponged between the cores.

CS 61C Fall 2019

Parallelism

Discussion 12: November 18, 2019

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.

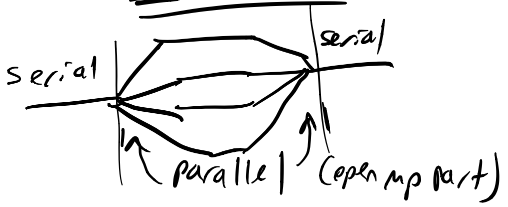
OpenMP provides an easy interface for using multithreading within C programs. Some examples of OpenMP directives:

like cache blocking but for threads

- 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



```
NOTE: The opening curly brace needs to be on a newline or else there
    will be a compile-time error!
```

- 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
for (int i = 0; i < n; i++) {
    ...
}
```

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

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
```

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

```
{
    for (int i = 0; i < n; i++)
        arr[i] = i;
}
```

since all threads perfor the same constant action, 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 chunked, 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] -= 1;
    }
}
```

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 (int i = 0; i < n; i++) {
        product *= arr[i];
    }
    return product;
}
```

(a) What is wrong with this code?

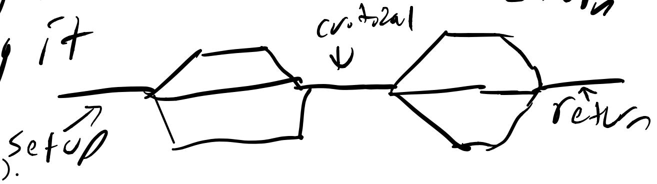
The code has the shared variable product.

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 (int i = 0; i < n; i++) {
        #pragma omp critical
        product *= arr[i];
    }
    return product;
}
```

adding the critical section sequentializes 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 Amdahl's Law

In the programs we write, there are sections of code that are naturally able to be sped up. However, there are likely sections that just can't be optimized any further to maintain correctness. In the end, the overall program speedup is the number that matters, and we can determine this using Amdahl's Law:

$$\text{True Speedup} = \frac{1}{S + \frac{1-S}{P}}$$

where S is the non-sped-up part and P is the speedup factor (determined by the number of cores, threads, etc.).

2.1 You are going to run a convolutional network to classify a set of 100,000 images using a computer with $\underline{\overset{p}{32}\text{ threads}}$. You notice that $\underline{\overset{1-S}{99\%}}$ of the execution of your project code can be parallelized on these threads. What is the speedup?

$1/(0.01 + 0.99/32) \approx 1/0.04 = 25$

$S = 100\% - 99\% = 1\% = 0.01$

$p = 32$

2.2 You run a profiling program on a different program to find out what percent of this program each function takes. You get the following results:

Function	% Time
f	30%
g	10%
h	60%

(a) We don't know if these functions can actually be parallelized. However, assuming all of them can be, which one would benefit the most from parallelism?

h This fn has the largest amount of time spent so the effort to speed this up will show greater benefit overall.

(b) Let's assume that we verified that your chosen function can actually be parallelized. What speedup would you get if you parallelized just this function with 8 threads?

$1-S = 60\%$ so $S = 40\%$

$1/(0.4 + 0.6/8) \approx 2.1$

$p = 8$ (8 threads)