CS 61C Spring 2021 — Coherency and Atomics, TLP

Discussion 13: April 19, 2021

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Each hardware thread in the CPU uses a shared cache.

False, each thread has its own cache, which can lead to cache-incoherence.

1.2 Atomicity can only be guaranteed within a single RISC-V instruction.

False, load-reserve, store-conditional allows uninterrupted execution across multiple instructions.

1.3 The amount of speedup is directly proportional to the increase in number of threads.

False, usually there is some overhead in paralleling an operation. Amdahl's Law shows that true speedup is affected not only by the number of threads but also by the amount of code that cannot be sped up.

2 Coherency and Atomics

The benefits of multi-threading programming come only after you understand concurrency. Here are two of the most common concurrency issues:

1. **Cache-incoherence**: each hardware thread has its own cache, hence data modified in one thread may not be immediately reflected in the other. This can often be solved by bypassing the cache and writing directly to memory, i.e. using volatile keywords in many languages.
2. **Read-modify-write**: Read-modify-write is a very common pattern in programming. In the context of multi-threading programming, the **interleaving** of R, M, W stages often produces a lot of issues.

In order to solve the problems created by Read-modify-write, we have to rely on the idea of **uninterrupted execution**, also known as atomic execution.

In RISC-V, we have two categories of atomic instructions:

1. **Load-reserve, store-conditional**: allows us to have uninterrupted execution across multiple instructions
2. **Amo.swap**: allows for uninterrupted memory operations within a single instruction

Both of these can be used to achieve atomic primitives. Here are examples for each:

Test-and-set

```
Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

Compare-and-swap

```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0
    otherwise
cas:
    lr.w t0, (a0) # Load original value.
    bne t0, a1, fail # Doesnt match, so fail.
    sc.w a0, a2, (a0) # Try to update.
    jr ra # Return.
fail:
    li a0, 1 # Set return to failure.
    jr ra # Return.
```

Instruction definitions:

1. **L**oad-**r**eserve: Loads the four bytes at `M[R[rs1]]`, writes them to `R[rd]`, sign-extending the result and registers a reservation on that word in memory.
2. **S**tore-**c**onditional **rd, rs2, (rs1)**: Stores the four bytes in register `R[rs2]` to `M[R[rs1]]`, provided there exists a load reservation on that memory address. Writes 0 to `R[rd]` if the store succeeded, or a nonzero error code otherwise.
3. **Amoswap rd, rs2, (rs1)**: Atomically, puts the sign-extended word located at `M[R[rs1]]` into `R[rd]` and puts `R[rs2]` into `M[R[rs1]]`.

Explanations for both methodologies:

1. **Test-and-set**: We have a lock stored at the address specified by `a0`. We utilize `amoswap` to put in 1 and get the old value. If the old value was a 1, we would not have changed the value of the lock and we will realize that someone currently has the lock. If the old value was a 0, we will have just "locked" the lock and can continue with the critical section. When we are done, we put a 0 back into the lock to "unlock" it.
2. **Compare-and-swap**: CAS tries to first reserve the memory and gets the value stored and compares it to the expected value. If the expected value and the value that was stored do not match, the entire process fails and we must restart to update based on the new information. Otherwise, we register a reservation on the memory and try to store the new value. If the exit code is nonzero, something went wrong with the store and we must retry the entire LR/SC process. Otherwise with a zero exit code, we continue into the critical section, then release the lock.

2.1 Why do we need special instructions for these operations? Why can't we use normal load and store for `lr` and `sc`? Why can't we expand `amoswap` to a normal load and store?

For `lr` and `sc`, after `lr`, other threads cannot write to the location marked reserve, hence the value loaded from memory will be unchanged between `lr` and `sc`. For `amoswap`, it does load and store in one single CPU cycle, hence the operation is atomic and uninterruptable.

2.2 Now that we have atomic operations, let's try to experiment with them. Let us try to implement an algorithm that enforces ordered thread execution. This means that if we have four threads, thread 0 goes first, thread 1 goes next, etc. For this problem assume that `a1` holds the location of a piece of memory we have access to for the entire duration of our algorithm. Also, we can assume there exists a label `get_thread_num` that returns the thread's number in `a0`, and that we save `ra` to the stack before line 1 and restore `ra` immediately before line 21. Try to fill in the blanks below. Please use LR/SC for this problem:

```
1         addi t0, x0, 0
2
3         _________________________________ # Setup for the first (0-th) thread
4         ...
5         # Assume we now spawn 4 threads in this code
6         ...
7
8   Check: jal ______________________________ # Get the current thread number
9                                             # Get the ID of the next thread that should operate
10        _________________________________ # (make sure this can't get interfered with)
11
12        _________________________________
13  Done:  addi t0, t0, 1
14
15        _________________________________ # Set which thread is next to run
16
17        bne ____________________________
18        ...
19        # Assume we now join the 4 threads in this code
20        ...
21        jr ra
```

```
1         addi t0, x0, 0
2         sw t0, 0(a1)
3         ...
4         # Assume we now spawn 4 threads in this code
5         ...
6   Check: jal ra, get_thread_num
7         lr.w t0, (a1)
8         bne a0, t0, Check
9   Done:  addi t0, t0, 1
10        sc.w a0, t0, (a1)
```

```
11          bne a0, x0, Check
12          ...
13          # Assume we now join the 4 threads in this code
14          ...
15          jr ra
```

3 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
{
    ...
}
```

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

3.1 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;
}
```

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];
```

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;
```

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.

3.2 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;
    }
}
```

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.

3.3

```
// 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.

(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;
}
```

(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;
}
```

4 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.).

4.1 You are going to run a convolutional network to classify a set of 100,000 images using a computer with 32 threads. You notice that 99% of the execution of your project code can be parallelized on these threads. What is the speedup?

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

4.2 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`

(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?

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