HW8.3. Threadsafe Programming

For each of the following code snippets, select one of the following that best describes its performance when run with 8 threads, compared to a single-threaded solution:

Incorrect result: The code runs into a data race, or otherwise has some chance of yielding an incorrect result. A solution with any chance of incorrect results is not considered correct, and as such is considered to be worse performance than a single-threaded solution.

Correct result, slower than serial: The code reaches the correct result, but does not speed up the computation.

Correct result, faster than serial: The code reaches the correct result, and speeds up the computation.

Q1.1:

```
//Set element i of arr to i
int A[1048576];
#pragma omp parallel
{
    for (int i = 0; i < 1048576; i++){
        A[i] = i;
    }
}
```

- (a) Incorrect result
- (b) Correct result, slower than serial ✅
- (c) Correct result, faster than serial

✓ 100%

Q1.2:

```
//Set all elements in arr to 17
int A[1048576];
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 1048576; i++){
        A[i] = 17;
    }
}
```

- (a) Incorrect result
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial ✅

✓ 100%

Q1.3:

```
//Set all elements in arr to 17
int A[1048576];
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        A[i] = 17;
    }
```

- (a) Incorrect result
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial ✅

✓ 100%

Q1.4:

Homework 8

Assessment overview

Total points: 50/100

Score: 50%

Question

Value: 20

History: 20

Awarded points: 20/20

Report an error in this question

Previous question

Next question

Attached files

No attached files

Attach a file

Attach text

```
//Set arr to be an array of Fibonacci numbers.
int A[1048576];
A[0] = 0;
A[1] = 1;
#pragma omp parallel for
    for (int i = 2; i < 1048576; i++){
        A[i] = A[i - 1] + A[i - 2];
    }
```

- (a) Incorrect result ✅
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial

✓100%

Q1.5:

```
int A[1048576];
initializedata(&A);
int j;
#pragma omp parallel for
    for (int i = 0; i < 1024; i++) {
        for (j = 0; j < 1024; j++) {
            A[i + j*1024] = i + j*1024;
        }
    }
```

- (a) Incorrect result ✅
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial

✓100%

Q1.6:

```
int A[1048576];
initializedata(&A);
#pragma omp parallel for
    for (int i = 0; i < 1024; i++) {
        for (int j = 0; j < 1024; j++) {
            A[i + j*1024] = i + j*1024;
        }
    }
```

- (a) Incorrect result
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial ✅

✓100%

Q1.7:

```
//Negates all elements of A
int A[1048576];
initializedata(&A);
#pragma omp parallel
{
    for (int i = 0; i < 1048576; i++)
        if(i%omp_get_num_threads() == omp_get_thread_num())
            A[i] = -A[i];
}
```

- (a) Incorrect result
- (b) Correct result, slower than serial ✅
- (c) Correct result, faster than serial

✓100%

Q1.8:

```
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        sum += A[i];
    }
```

- (a) Incorrect result ✅
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial

✓ 100%

Q1.9:

```
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        #pragma omp critical
        sum += A[i];
    }
```

- (a) Incorrect result
- (b) Correct result, slower than serial ✅
- (c) Correct result, faster than serial

✓ 100%

Q1.10:

```
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel
{
    int private_sum = 0;
    #pragma omp for
    for (int i = 0; i < 1048576; i++){
        private_sum += A[i];
    }
    #pragma omp critical
    sum += private_sum;
}
```

- (a) Incorrect result
- (b) Correct result, slower than serial
- (c) Correct result, faster than serial ✅

✓ 100%

Try a new variant

Correct answer

Q1.1:

```
//Set element i of arr to i
int A[1048576];
#pragma omp parallel
{
    for (int i = 0; i < 1048576; i++){
        A[i] = i;
    }
}
```

(b) Correct result, slower than serial

Q1.1: Each thread ends up running the entire for loop, since there is no "parallel for". As such, the only difference between the naive and the threaded solution is the overhead in multithreading, so the threaded solution is slower.

Q1.2:

```
//Set all elements in arr to 17
int A[1048576];
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 1048576; i++){
        A[i] = 17;
    }
}
```

(c) Correct result, faster than serial

Q1.2: This works as expected; the `for` directive automatically splits the loop between threads.

Q1.3:

```
//Set all elements in arr to 17
int A[1048576];
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        A[i] = 17;
    }
```

(c) Correct result, faster than serial

Q1.3: This is the same implementation as the previous question. The two directives can be combined in a single `#pragma omp parallel for` directive.

Q1.4:

```
//Set arr to be an array of Fibonacci numbers.
int A[1048576];
A[0] = 0;
A[1] = 1;
#pragma omp parallel for
    for (int i = 2; i < 1048576; i++){
        A[i] = A[i - 1] + A[i - 2];
    }
```

(a) Incorrect result

Q1.4: There are data dependencies between each step, so this causes a likely data race. A solution similar to the one in the SIMD question could speed things up (splitting into four threads, with each thread handling only every fourth value), but would also likely slow things down due to cache coherency issues.

Q1.5:

```
int A[1048576];
initializedata(&A);
int j;
#pragma omp parallel for
    for (int i = 0; i < 1024; i++) {
        for (j = 0; j < 1024; j++) {
            A[i + j*1024] = i + j*1024;
        }
    }
```

(a) Incorrect result

Q1.5: The `for` directive makes `i` private, but `j` is still shared among threads. If one thread increments `j` before another thread is finished writing, then the code does not work.

Q1.6:

```c
int A[1048576];
initializedata(&A);
#pragma omp parallel for
    for (int i = 0; i < 1024; i++) {
        for (int j = 0; j < 1024; j++) {
            A[i + j*1024] = i + j*1024;
        }
    }
```

(c) Correct result, faster than serial

Q1.6: This fixes the issue from the previous problem. Now, both `i` and `j` are private variables per thread.

Q1.7:

```c
//Negates all elements of A
int A[1048576];
initializedata(&A);
#pragma omp parallel
{
    for (int i = 0; i < 1048576; i++)
        if(i%omp_get_num_threads() == omp_get_thread_num())
            A[i] = -A[i];
}
```

(b) Correct result, slower than serial

Q1.7: This looks like it would be faster, but in practice, this will probably be slower, due to several inefficiencies. For one, we are still going through all the iterations anyway, and just not doing anything on some iterations; it still takes a decent amount of time to run through the iteration, though; as such, it's better to modify the loop itself to skip iterations. The second, more important issue, is that due to cache coherency issues, we lose almost all of the benefits of the L1 and L2 caches since the data keeps getting invalidated/dirtied. The L1 cache is around 10 times faster than the L3 cache, so this likely is slightly slower than the naive approach.

Q1.8:

```c
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        sum += A[i];
    }
```

(a) Incorrect result

Q1.8: This has data race issues since multiple threads will try to write to the same location `sum` at the same time. The order of accesses is non-deterministic which can lead to different results each time you execute the program.

Q1.9:

```c
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel for
    for (int i = 0; i < 1048576; i++){
        #pragma omp critical
        sum += A[i];
    }
```

(b) Correct result, slower than serial

Q1.9: The `#pragma omp critical` directive will ensure that only one thread can execute the `sum` update. While this solves the data race issues highlighted in the previous problem, the fact that only one thread can enter the critical section at a time will negate any benefits from multi-threading. Multi-threading overhead and false sharing will make this approach slower than the non-threaded version.

Q1.10:

```
int A[1048576];
initializedata(&A);
int sum = 0;
#pragma omp parallel
{
    int private_sum = 0;
    #pragma omp for
    for (int i = 0; i < 1048576; i++){
        private_sum += A[i];
    }
    #pragma omp critical
    sum += private_sum;
}
```

(c) Correct result, faster than serial

Q1.10: `private_sum` is a private variable since it is declared inside the `#pragma omp parallel` directive. The `omp for` directive then divides the for loop into the different threads. Each thread will get to update their `private_sum` variable. The final summation is done within the critical statement once the loop for all threads is done. This is the correct multi-threaded solution compared to the previous two questions.

Submitted answer 3 **correct: 100%**

Submitted at 2022-11-09 03:31:17 (PST)

show