CS61C | Thread-Level Parallelism
Spring 2025 | Discussion 11

1 Thread-Level Parallelism

For each question below, state whether the program is:

1) Always Correct, Sometimes Correct, Always Incorrect

2) Faster than Serial, Slower than Serial

Assume the number of threads can be any integer greater than 1 and that no thread will complete in its entirety before another thread starts executing. `arr` is an `int[]` of length `n`.

1.1

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

1) Always Correct

2) Slower than Serial

The values will be correct at the end of the loop since each thread is writing the same values.

Note that there is no `for` directive, so every thread executes this loop in its entirety. The overhead of creating and managing threads will slow down the execution time to be slower than serial.

1.2

```
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];
```

1) Sometimes Correct

2) Slower than Serial

Sometimes correct: the loop has dependencies from previous data, so each thread would have to wait for its previous dependency to finish which does not occur in this loop. However, there exists a thread ordering where they execute in such a way that they complete each iteration in sequential order.

Even if this happened, this would still be slower than serial due to the multithreading overhead required.

1.3

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

1) Always Correct

2) Faster than Serial

The **for** directive automatically makes loop variables (such as the index) private, so this will work properly. The **for** directive splits up the iterations of the loop to optimize for efficiency, and there will be no data races.

1.4

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

1) Sometimes Correct

2) Slower than Serial

Because each thread shares the array pointer, there is a data race when incrementing the array pointer. If multiple threads are executed such that they all execute the first line, `*arr = i;` before the second line, `arr++`;, they will clobber each other's outputs by overwriting what the other threads wrote in the same position. However, there is a thread execution order that will not encounter data races, though it will be slower than serial.

2 Critical Sections

2.1 Consider the following multithreaded code to compute the product over all elements of an array.

```
// Assume arr has length 8*n.
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
                          * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7];
        product *= subproduct;
    }
    return product;
}
```

(a) What is wrong with this code?

The code has the shared variable **product**, which can cause data races when multiple threads access it simultaneously.

(b) Fix the code using **#pragma omp critical**. On which line should you place the directive to create the critical section?

```
// Assume arr has length 8*n.
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double subproduct = arr[i*8] * arr[i*8+1]
                        * arr[i*8+2] * arr[i*8+3]
                        * arr[i*8+4] * arr[i*8+5]
                        * arr[i*8+6] * arr[i*8+7];
        #pragma omp critical
        product *= subproduct;
    }
    return product;
}
```

2.2 When added to a **#pragma omp parallel for** statement, the **reduction(operation: var)** directive creates and optimizes the critical section for a for loop, given a variable that should be in the critical section and the operation being performed on that variable. An example is given below.

```
// Assume arr has length n
int fast_sum(int *arr, int n) {
    int result = 0;
    #pragma omp parallel for reduction(+: result)
    for (int i = 0; i < n; i++) {
        result += arr[i];
    }
    return result;
}
```

Fix **fast_product** by adding the **reduction(operation: var)** directive to the **#pragma omp parallel for** statement. Which variable should be in the critical section, and what is the operation being performed?

```
// Assume arr has length 8*n.
double fast_product(double *arr, int n) {
    double product = 1;

    __________________________________________
    for (int i = 0; i < n; i++) {
        double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
                        * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7];
        product *= subproduct;
    }
    return product;
}
```

```
double fast_product(double *arr, int n) {
    double product = 1;
    #pragma omp parallel for reduction (*:product)
    for (int i = 0; i < n; i++) {
        double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
                        * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7];
        product *= subproduct;
    }
    return product;
}
```

2.3 Take a look at the following code which is run with two threads:

```
#define N 5

void func() {
  int A[N] = {1, 2, 3, 4, 5};
  int x = 0;
  #pragma omp parallel
  {
    for (int i = 0; i < N; i += 1) {
      x += A[i];
      A[i] = 0;
    }
  }
}
```

What are the maximum and minimum values that `x` can have at the end of `func`?

Each of the 2 threads will independently:

- Read value from X
- Read value from A
- Add value to `x`
- Zero out value in A
- Do the loop for 5 iterations each

Maximum: `x = 30` – if thread 1 reads from `x`, reads from the array, and adds to `x`, and then thread 2 reads from the new `x`, reads from the array, and adds to `x` *before* the array entry gets zeroed, then `x` will have the value of `x += A[i] + A[i]`.

Minimum: `x = 0` – thread 2 reads from `x` getting the value `x = 0` *but* halts and waits for thread 1 to completely finish (setting all array entries to 0). When thread 2 resumes execution, it will add its current value for `x` to a zeroed `A[i]` which will be `0 + 0 = 0` at all iterations of the loop.

3 OpenMProgramming

Consider the following C function:

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    C[i] = A[i] - B[i];
  }
}
```

3.1 Manually rewrite the loop to split the work equally across N different threads.

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {
  #pragma omp parallel
  {
    int N = OMP_NUM_THREADS;
    int tid = omp_get_thread_num();

    for (int i = tid; i < ARRAY_LEN; i += N) {
      C[i] = A[i] - B[i];
    }
  }
}
```

3.2 Now, split the work across `N` threads using a `#pragma` directive:

```
#define ARRAY_LEN 1000

void mystery(int32_t *A, int32_t *B, int32_t *C) {
  #pragma omp parallel for
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    C[i] = A[i] - B[i];
  }
}
```

3.3 Instead of saving the product to an array `C`, we now want to XOR the subtraction of all the elements of `A` and `B`.

```
#define ARRAY_LEN 1000

int mystery(int32_t *A, int32_t *B) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    result ^= A[i] - B[i];
  }
  return result;
}
```

What is the issue with the above implementation and how can we fix it?

There is a race condition for the `result` variable.

3.4 Solve the problem above in two different methods using OpenMP:

(a)

```
int mystery(int32_t *A, int32_t *B) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    #pragma omp critical
    result ^= A[i] - B[i];
  }
  return result;
}
```

(b)

```
int mystery(int32_t *A, int32_t *B) {
  int result = 0;
  #pragma omp parallel for reduction(^:result)
  for (int i = 0; i < ARRAY_LEN; i += 1) {
    result ^= A[i] - B[i];
  }
  return result;
}
```

3.5 Assume we run the above **mystery** function with 8 threads. The parallel portion accounts for 80% of the program and is 8x as fast as the naive implementation. Use Amdahl's Law to calculate the speedup of the full program where

$$\text{Speedup} = \frac{1}{\left(1 - \text{frac}_{\text{optimized}}\right) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}}$$

$$\begin{aligned}\text{Speedup} &= \frac{1}{\left(1 - \text{frac}_{\text{optimized}}\right) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}} \\ &= \frac{1}{(1-0.8) + \frac{0.8}{8}} \\ &= \frac{1}{0.2 + 0.1} \\ &= 3.333\text{x speedup!}\end{aligned}$$

3.6 What is the maximum speedup we can achieve if we use unlimited threads in the parallel section for an infinite performance increase? Assume the parallel portion still accounts for 80% of our program.

$$\begin{aligned}\text{Speedup} &= \frac{1}{\left(1 - \text{frac}_{\text{optimized}}\right) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}} \\ &= \frac{1}{(1-0.8) + \frac{0.8}{9999999...}} \\ &= \frac{1}{0.2} \\ &= 5\text{x maximum speedup!}\end{aligned}$$

3.7 What does the above result tell you about using parallelism to optimize programs?

Programs can only be as fast as their serial portion.