CS 61C Fall 2024 — PLP, TLP — Discussion 11

1 Thread-Level Parallelism

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 exact order of execution across all threads, as well as the number of iterations each thread performs, are both non-deterministic, as the OpenMP library load balances threads for performance. 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

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 number of threads can be any integer greater than 1. Assume no thread will complete in its entirety 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;
}
```

(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];
```

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

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

2 Locks and Critical Sections

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

```
1  // Assume arr has length 8*n.
2  double fast_product(double *arr, int n) {
3      double product = 1;
4      #pragma omp parallel for
5      for (int i = 0; i < n; i++) {
6          double subproduct = arr[i*8]*arr[i*8+1]*arr[i*8+2]*arr[i*8+3]
7                          * arr[i*8+4]*arr[i*8+5]*arr[i*8+6]*arr[i*8+7];
8          product *= subproduct;
9      }
10     return product;
11 }
```

(a) What is wrong with this code?

(b) Fix the code using `#pragma omp critical`. What line would you place the directive on to create that critical section?

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 the code 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?

3 Multi-Process Code

One advantage of process-level parallelism is that we have freedom to do complex tasks without worrying about race conditions in memory due to processes not sharing memory. Examine the code snippet below to answer the questions.

```
int x = 10;
int y = 0;

// Split into two processes

if (/* Is Process 1 */) { y++; }
if (/* Is Process 2 */) { x--; }
```

3.1 After the code segment completes, what will be the values of `x` and `y` for Process 1?

3.2 After the code segment completes, what will be the values of `x` and `y` for Process 2?

4 Manager-Worker Framework

Recall the manager-worker pseudocode:

Manager:

```
setup
while there is work to do:
    wait for a worker to ask for work
    find the next task to do
    assign that task
for each worker:
    wait for a worker to ask for work
    tell the worker that work is done
teardown
```

Worker:

```
setup
done = False
while not done:
    ask manager for work
    if reply is a task:
        do the task
    if reply is work is done:
        done = True
teardown
```

Out of all the steps above, one step is notably more interesting than the rest — how the manager chooses the next task to do. For this part, assume we have the following list of tasks, which each take some specified amount of time to complete:

Task #	1	2	3	4	5	6	7	8
Time (s)	8	2	1	3	2	1	1	6

Suppose that we have 1 manager and 2 workers. List out the tasks assigned to each worker, and the total amount of time taken if the manager assigns tasks... (if multiple tasks can be assigned, the one with the smallest number is chosen)

4.1 ...by choosing the task that takes the **shortest** amount of time to do.

4.2 ...by choosing the task that takes the **longest** amount of time to do.

4.3 ...by choosing the task with the **smallest** task number.

4.4 Compare the above approaches to assigning work. Which ones were the fastest? The slowest? Are there any benefits / approaches to each of these techniques beyond completion time?