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;
}
```

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. The values should all be correct at the end of the loop since each thread is writing the same values. Furthermore, the existence of parallel overhead due to the extra number of threads will 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];
```

Sometimes incorrect: While the loop has dependencies from previous data, in a interweaved scheme where the threads take turns completing each iteration in sequential order (e.g.

```
for (int i = omp_get_thread_num(); i < n; i += omp_get_num_threads())
```

is the work allocation per thread and the order of execution is based on the shared variable **i** from 2 to **n**), each thread will have the correctly updated shared **arr** to compute the next Fibonacci number. Note that this scheme would still be slower than serial due to the amount of overhead required as the threads need to wait for each other's execution to finish as well as deal with coherency issues regarding the shared data.

(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 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.

(d)

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

Sometimes incorrect: Because we are not indexing into the array, there is a data race to increment 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, taking a similar interweaved scheme as in 4.1b, there is an order that will not encounter data races, though it will be slower than serial.

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?

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`. What line would you place the directive on to create that critical section?

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

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.

```
1  // Assume arr has length n
2  int fast_sum(int *arr, int n) {
3      int result = 0;
4      #pragma omp parallel for reduction(+: result)
5      for (int i = 0; i < n; i++) {
6          result += arr[i];
7      }
8      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?

```
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;
}
```

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?

```
x = 10;
y = 1;
```

Notice that only the value of `y` changes. This is because when we create a new processes, it is given a separate address space. This enforces the separation between processes that provides security within a system.

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

```
x = 9;
y = 0;
```

Notice that only the value of `x` changes. This is when a new process is created, it is initialized with a separate address space.

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.

The tasks would be assigned as follows:

<table>
<tr><td>Time (sec)</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>Worker 1</td><td>3</td><td>7</td><td colspan="2">5</td><td colspan="6">8</td><td colspan="4">Idle</td></tr>
<tr><td>Worker 2</td><td>6</td><td colspan="2">2</td><td colspan="3">4</td><td colspan="8">1</td></tr>
</table>

In total, this takes 14 seconds to complete all the tasks.

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

The tasks would be assigned as follows:

<table>
<tr><td>Time (sec)</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td></tr>
<tr><td>Worker 1</td><td colspan="8">1</td><td colspan="2">2</td><td>6</td><td>7</td></tr>
<tr><td>Worker 2</td><td colspan="6">8</td><td colspan="3">4</td><td colspan="2">5</td><td>3</td></tr>
</table>

In total, this takes 12 seconds to complete all the tasks.

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

The tasks would be assigned as follows:

<table>
<tr><td>Time (sec)</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td>Worker 1</td><td colspan="8">1</td><td>6</td><td colspan="6">8</td></tr>
<tr><td>Worker 2</td><td colspan="2">2</td><td>3</td><td colspan="3">4</td><td colspan="2">5</td><td>7</td><td colspan="6">Idle</td></tr>
</table>

In total, this takes 15 seconds to complete all the tasks.

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?

The fastest method was assigning the task that takes the longest to complete, and the slowest was assigning tasks sequentially. As a preview to CS162, one other thing to consider could be that maybe we want lower tasks to finish first (e.g. if these tasks arrived first).