CS61C Spring 2025 — Precheck: TLP and Concurrency

Discussion 11

1 Pre-Check: T/F?

1.1 Multithreaded code will always outperform single-threaded code.

False. Not all code lends itself to parallelization. Code with many dependencies may require long critical sections which may be slower than single-threaded code. Additionally, spawning new threads takes time that will slow down the program.

1.2 Race conditions occur when multiple threads attempt to modify the same resource at the same time.

True. Race conditions can have unintended side-effects when writing parallel code. An example is given below:

```
void race_condition(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    result += a[i];
  }
}
```

Accessing `a[i]`, adding `result + a[i]`, and storing back to `result` are separate operations during which threads may operate concurrently. These can be solved with critical sections!

1.3 Every thread has its own set of registers, program counter, heap and global variables

False. Each thread has its own set of registers and program counter, but the heap and global variables are shared amongst all threads.

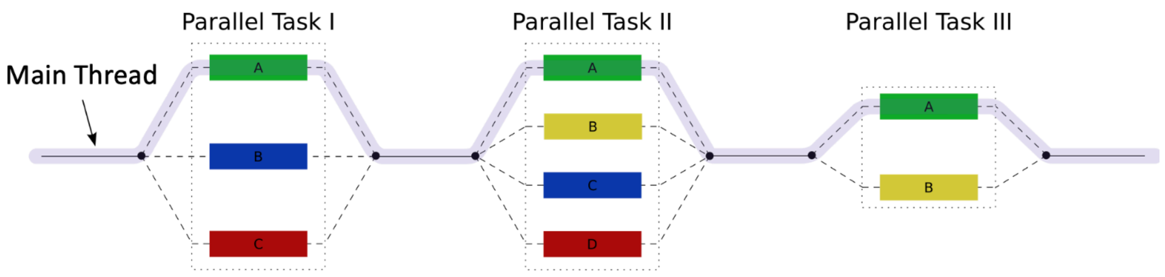
1.4 Multithreaded programs can still execute on single-core processors

True. Single-core processors will interleave its execution between different threads.

2 Thread-Level Parallelism

Multithreading improves performance by utilizing a form of parallelism called **thread-level parallelism**. The key idea behind thread-level parallelism is splitting from a single line of execution to multiple lines executing concurrently.

Multithreaded programs will start with a main thread that will spawn multiple threads when parallelism is required (See Lecture 31).



In this class, we use the OpenMP library to create and manage threads. Consider the sample hello world program, which prints "hello world from thread #" from each thread:

```
int main() {
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("hello world from thread %d\n", thread_id);
    }
}
```

This program will create a team of parallel threads. Each thread prints out a hello world message, along with its own thread number.

Let's break down the `pragma omp parallel` line:

- `pragma` tells the compiler that the rest of the line is a directive.
- `omp` declares that the directive is for OpenMP.
- `parallel` says that the following block statement – the part inside the curly braces ({/}) – should be executed in parallel by different threads.

Note that each thread has its own registers (including stack pointer) and program counter (PC). However, memory in the heap or global variables are shared amongst all threads.

3 OpenMP

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++) { ... }
  }
  ```

- The `critical` directive only allows a single thread to access the following line / block of code at once. It is useful to prevent race conditions when modifying resources shared between threads.

  ```
  // Assume arr has length n
  int fast_sum(int *arr, int n) {
      int result = 0;
      int result2 = 0;
      #pragma omp parallel for
      for (int i = 0; i < n; i++) {
          #pragma omp critical
          result += arr[i];  // limited to one thread access at once
      }
      return result;
  }
  ```

- The `parallel for reduction(operation : var)` directive creates and optimizes the critical section for a 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;
  }
  ```

Additionally, there are two OpenMP functions 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

4 Race Conditions

Two threads attempting to access the same memory can result in a race condition. Consider the following:

```
void example(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    result += a[i];
  }
}
```

For a single thread to compute the line `result += a[i]`:
(a) Read the value `a[i]`
(b) Compute the value `result + a[i]`
(c) Store the new value back to `result`

Different threads often have their instructions interleaved. Without defining a critical section, these threads may be executing steps (a), (b), and (c) at any time and the value of `result` may be indeterminate.

To fix this, we can introduce a critical section to limit one thread to executing `result += a[i]` at any given time:

```
void example(int n, int *a) {
  int result = 0;
  #pragma omp parallel for
  for (int i = 0; i < n; i += 1) {
    #pragma omp critical
    result += a[i];
  }
}
```

It can also be fixed with the `reduction` directive (see above)!