



# Lab 10: Thread-Level Parallelism

Deadline: Friday, November 19, 04:00:00 PM PT

## Goals

- Learn about basic OpenMP directives.
- Write code to learn two ways of how `#pragma omp for` could be implemented. Learn about false sharing.
- Learn about basic multi-processing programming.

## Setup

**Warning:** We strongly recommend working on the hive machines for this lab. OpenMP may perform differently on your local machine, and installing it can be annoying.

In your `labs` directory, pull the files for this lab with:

```
$ git pull starter main
```

## Info: OpenMP

OpenMP stands for **O**pen specification for **M**ulti-**P**rocessing. It is a framework that offers a C interface. It is not a built-in part of the C language -- most OpenMP features are compiler directives. (One example of a compiler directive you've seen in the past is `#include`.)

Benefits of multi-threaded programming using OpenMP include:

- A very simple interface that allows a programmer to separate a program into serial regions and parallel regions.
- Convenient synchronization control (data race bugs in threads are very hard to trace).

In this lab, we'll practice some basic usage of OpenMP. OpenMP is already installed on hive machines, so we strongly recommend that you work on the hive machines for this lab.

---

## Exercise 1: OpenMP Hello World

Consider the sample hello world program (`hello.c`), 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.

You can change the number of OpenMP threads by setting the environment variable `OMP_NUM_THREADS` or by using the `omp_set_num_threads` function before the parallel section in your program.

Try running the program:

```
$ make hello
$ ./hello
```

If you run `./hello` a couple of times, you may notice that the printed numbers are not always in increasing order and will most likely vary across runs. This is because we didn't specify any sort of synchronization options, so OpenMP will not enforce any execution order. (More on that later.) It is also vital to note that the variable `thread_id` is defined inside the parallel block, so it is local to each thread and not shared across all threads. In general with OpenMP, variables declared inside the parallel block will be private to each thread, but variables declared outside a parallel block will be shared across all threads. Again, there are ways to override this, but more on that later.

## Exercise 2: Vector Addition

Vector addition is a naturally parallel computation, since it's an elementwise operation (element `i` of the result vector does not depend on elements `j != i`), so it makes for a good first exercise. The `v_add()` functions inside `omp_apps.c` will store the sum of input vectors `x` and `y` into the result vector `z`. A first attempt at this might look like:

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel
    {
        for(int i=0; i<ARRAY_SIZE; i++)
        {
            z[i] = x[i] + y[i];
        }
    }
}
```

Try running the tests:

```
$ make v_add
$ ./v_add
```

The testing framework will time the function execution for different thread counts. You should observe that this implementation performs worse as we increase the number of threads! Why?

The issue is that each thread is executing all of the code within the `omp parallel` block. If we have 8 threads, we'll actually be performing the same vector addition 8 times! Not only that, but various threads writing to the same variables in memory may cause a decrease in performance due to cache synchronization. Rather than have each thread run every iteration of the `for` loop, we need to split the `for` loop iterations across all the threads so each thread does only a portion of the work.

The good news is that OpenMP has built-in functionality for dividing up the work of `for` loops among threads. The bad news is that for this exercise, we'd like you to first try splitting the for loop iterations among threads manually. Let's try to understand a couple methods we could use to divide the work, and the benefits/drawbacks of each way. You cannot use the OpenMP `for` directive for this exercise, but this is how you'd use that directive:

```
void v_add(double* x, double* y, double* z) {
    #pragma omp parallel for
    for(int i=0; i<ARRAY_SIZE; i++)
    {
        z[i] = x[i] + y[i];
    }
}
```

```
}
```

The two differences between this example and the previous one are: 1), the `for` at the end of the directive, and 2), the lack of opening and closing brackets around the parallel section. The `#pragma omp parallel for` directive needs only to be placed directly before a `for` loop; brackets are not required. Again, remember that for this exercise, you cannot use this directive.

Your task is to optimize `v_add()`. Remember that speedup may increase slower after a certain number of threads, since part of the whole program is not parallelizable. Your implementations should use the following 2 functions -- don't hardcode thread counts or thread IDs:

- `int omp_get_num_threads()` - returns the current **total** number of OpenMP threads. Note that this will be `1` outside of an OpenMP `parallel` section.
- `int omp_get_thread_num()` - returns the thread number of the current thread, commonly used as thread ID.

## Action Items

The methods you'll be implementing are in `omp_apps.c`.

Start with implementing `v_add_optimized_adjacent()`, which separates the vectors into element-wise **slices**. Each thread handles "adjacent" sums; in other words, thread 0 will add the elements at indices where `i % num_threads == 0`, thread 1 will add the elements at indices where `i % num_threads == 1`, and so on.

Then, implement `v_add_optimized_chunks()` which separates the vectors into contiguous **chunks**. As an example, if we have 4 threads, thread 0 will handle the first quarter of the elements in the array, thread 1 will handle the second quarter, and so on. Remember to handle the "tail case" -- depending on your chunking logic, the last chunk may have slightly more or less elements than the others.

## Questions

1. Which version of your code runs faster, chunks or adjacent? Put your answer (either "chunks" or "adjacent") in `answers/ex2.txt`. Be prepared to relate your answer to cache coherency, false sharing, and/or other memory considerations.

---

## Exercise 3: Dot Product

The next task is to compute the dot product of two vectors. At first glance, implementing this might seem not too different from `v_add`, since we should now just perform elementwise multiplication

instead of addition. But the challenge is how to sum up all of the products into the same variable (reduction) to get our final answer. A sloppy handling of reduction may lead to **data races**: all the threads are trying to read and write to the same address simultaneously.

One solution is to use a **critical section**. The code in a critical section can only be executed by a single thread at any given time. Thus, having a critical section naturally prevents multiple threads from reading and writing to the same data, a problem that would otherwise lead to data races. One way to avoid data races is to use the `critical` primitive provided by OpenMP. An implementation, `dotp_naive()` in `omp_apps.c`, protects the sum with a critical section.

Try running the tests:

```
$ make dotp
$ ./dotp
```

Notice how the performance gets much worse as the number of threads goes up? By putting all of the work of reduction in a critical section, we have flattened the parallelism and made it so only one thread can do useful work at a time (not exactly the idea behind thread-level parallelism). This contention is problematic; each thread is constantly fighting for the critical section and only one is making any progress at any given time. As the number of threads goes up, so does the contention, and the performance pays the price. Can we reduce the number of times that each thread needs to use a critical section?

## Action Items

The methods you'll be implementing are in `omp_apps.c`.

First, let's fix this performance problem **without** using the `reduction` keyword. Write your implementation in `dotp_manual_optimized()`. Remember that we want to reduce the number of times each thread enters the `critical` section.

Next, let's fix this problem using OpenMP's built-in `reduction` keyword. Write your implementation in `dotp_reduction_optimized()`. Note that your code should no longer contain `#pragma omp critical`.

**Note:** In the starter code, you'll see the "expanded" version of the `#pragma omp parallel for` directive, which is written as a `#pragma omp parallel {}` section with a nested `#pragma omp for` directive before the for loop. These two are functionally equivalent.

---

## Exercise 4: Feedback Form

We are working to improve the labs for next semester, so please fill out [this survey](#) to tell us about your experience with Lab 10. The survey will be collecting your email to verify that you have submitted it, but your responses will be anonymized before the data is analyzed. Thank you!

---

## Submission

Save, commit, and push your work, then submit to the **Lab 10** assignment on Gradescope.

The lab autograder tests are slightly modified versions of the tests you have locally. In addition to correctness, it looks for some basic improvements in performance. In particular:

- A **2x** speedup from the naive benchmark to the fastest adjacent/chunks runtime (out of all different number of threads) for `v_add`.
  - A **9x** speedup from the naive benchmark to the fastest manual/reduction runtime (out of all different number of threads) for `dotp`.
- 

## Checkoff

---

☐ Dark Mode