Lab 10

Computer Architecture I ShanghaiTech University
Lab 9 Lab 10 Lab 11

Objectives:

- 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

Download the files: dotp.c, hello.c, Makefile and v_add.c.

Multi-threading programming using OpenMP

OpenMP stands for Open specification for Multi-Processing. It is a framework that offers a C interface. It is not a built-in part of the language – most OpenMP features are directives to the compiler.

Benefits of multi-threaded programming using OpenMP include:

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

In this lab, we will practice on basic usage of OpenMP.

Exercise 1 - OpenMP Hello World

Consider the implementation of Hello World (`hello.c`):

```
int main ()
{
  #pragma omp parallel
  {
    int thread_ID = omp_get_thread_num ();
    printf (" hello world %d\n", thread_ID);
  }
  return 0;
}
```

This program will fork off the default number of threads and each thread will print out "hello world" in addition to which thread number it is. 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 in your program. The `#pragma` tells the compiler that the rest of the line is a directive, and in this case it is omp parallel. `omp` declares that it is for OpenMP

and `parallel` says the following code block (what is contained in { }) can be executed in parallel. Give it a try:

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

If you run `./hello` a couple of times, you should see that the numbers are not always in numerical order and will most likely vary across runs. This is because within the parallel region, OpenMP does the code in parallel and as a result does not enforce an ordering across all the threads. It is also vital to note that the variable `thread_ID` is local to a specific 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 will be global and accessible by all the threads.

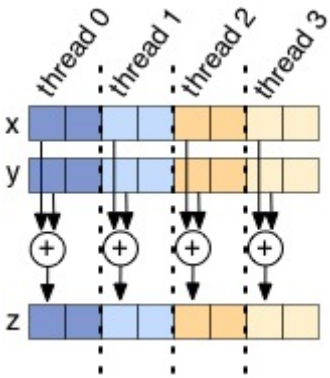
Exercise 2 - Vector Addition

Vector addition is a naturally parallel computation, so it makes for a good first exercise. The `v_add` function inside `v_add.c` will return the array that is the cell-by-cell addition of its inputs x and y. 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];
  }
}
```

You can run this (`make v_add` followed by `./v_add`) and the testing framework will vary the number of threads and time it. You will see that this actually seems to do worse as we increase the number of threads. The issue is that each thread is executing all of the code within the `omp parallel` block, meaning if we have 8 threads, we will actually be adding the vectors 8 times.

Rather than have each thread run the entire for loop, we need to split up the for loop across all the threads so each thread does only a portion of the work.



Your task is to optimize `v_add.c` (speedup may plateau as the number of threads continues to increase). To aid you in this process, two useful OpenMP functions are:

- `int omp_get_num_threads()`
- `int omp_get_thread_num()`

Divide up the work for each thread through two different methods (write different code for each of these methods):

1. First task, **slicing**: have each thread handle adjacent sums: i.e. Thread 0 will add the elements at indices `i` such that `i % omp_get_num_threads()` is `0`, Thread 1 will add the elements where `i % omp_get_num_threads()` is `1`, etc.
2. Second task, **chunking**: if there are N threads, break the vectors into N contiguous chunks, and have each thread only add that chunk (like the figure above).

Hints:

- Use the two functions we listed above somehow in the for loop to choose which elements each thread handles.
- You may need a special case to prevent going out of bounds for `v_add_optimized_chunks`. Don't be afraid to write one.
- Thinking about false sharing – read more here and here.

For this exercise, we are asking you to manually split the work amongst threads since this is a common pattern used in software optimization. The designers of OpenMP actually made the `#pragma omp for` directive to automatically split up independent work. Here is the function rewritten using it. **You may NOT use this directive in your solution to this exercise**.

```
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];
}
```

Test the performance of your code with:

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

Checkoff

- Show the TA your code for `v_add` that manually splits up the work (both methods).
- Run your code to show that it gets parallel speedup (second method).

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`, but the challenge is how to sum up all of the products into the same variable (reduction). 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 `dotp.c`, protects the sum with a critical section.

Try out the code (`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?

In this exercise, you have 2 tasks:

1. Fix the performance problem without using OpenMP's built-in Reduction keyword.
2. Fix the performance problem using OpenMP's built-in Reduction keyword. (Note that your code should no longer contain `#pragma omp critical`)

Checkoff

- Show the TA your manual fix to `dotp.c` that gets speedup over the single threaded case.
- Show the TA your Reduction keyword fix for `dotp.c`, and explain the difference in performance.

Schwertfeger, Sören <*soerensch AT shanghaitech.edu.cn*>
Chundong Wang <*wangchd AT shanghaitech.edu.cn*>
Modeled after UC Berkeley's CS61C.
Last modified: 2020-05-01