CMSC416: Introduction to Parallel Computing

Topic: OpenMP
Date: March 05, 2024

## Updates

- Project 1:
  - Deadline extended until Friday, March 8th
  - Do not merge nodes with the same name (using groupby i.e.) before tallying your values. You'll get different answers and output.
  - Don't need to add the shebang #!/usr/bin/python3 to our scripts, they'll run python on the files
- Some student notes are formatted and up on the class website. Slides are still the primary (and trusted) source of information, but the student notes are available for reference, taken with a grain of salt.

### Quiz 1 Clarification(s)

MPI can indeed be run on shared memory architectures. That's what we do on Zaratan – each MPI process shares a fraction of the shared memory pool, even though it only has 128 cores.

## OpenMP

The OMP reduction minus (-) operator just does a plus sum operation, according to the OMP documentation.

Q: Why can't we share a local variable across all OMP threads?
A: It causes a data race, which can cause values to become mangled and take on older / out-of-date values depending on the random timing of the threads

## Loop Scheduling

Loop scheduling can be used to assign different loop iterations to different OMP threads.

**Static scheduling** dictates to OMP that it should try to divide the work as evenly as possible (so think # iterations / # threads)

**Dynamic scheduling** queues blocks of data internally for threads to work through. Once a thread finishes processing its block, it'll pick up the next block from the internal queue. This means that faster threads can pick up more work

**Guided scheduling** is similar to dynamic, but we start with a larger chunk size and decrease it gradually as the workload decreases over time

**Auto scheduling** is up to the compiler's discretion

We can use the `OMP_SCHEDULE` environment variable to dynamically control all non-hardcoded schedule loops in our program. This means that we don't have to recompile our code if we want to test different schedulers, which can be useful for performance testing.

## Parallelization

When parallelizing loops, ask yourself: are there going to be race conditions between any of the local variables?

**Calculate the value of** $\pi = \int_0^1 \frac{4}{1+x^2}$

```
int main(int argc, char *argv[])
{
    ...

    n = 10000;
    h   = 1.0 / (double) n;
    sum = 0.0;

    #pragma omp parallel for firstprivate(h) private(x) reduction(+: sum)
    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;

    ...
}
```

Most variables in an OMP program are shared by default, unless we make it private! In this example, the i variable doesn't need to be bothered, since loop variables are private by default in OMP.

```
#pragma omp parallel for firstprivate(h) private(x) reduction(+: sum)
```

This OMP directive sums up all of the individual values of `sum` (from each thread private sum), and stores the sum in the master thread `sum` variable.

x should be a local variable to each thread. We have to tell OMP that, otherwise our threads will treat it as a shared variable and start overwriting it

`firstprivate(h)` is not required in this example, since it's constant and exists already before the thread is dispatched. But if we wanted to copy over some value before our thread logic begins, we can still use `firstprivate(h)`

```
#pragma omp parallel [cluse [clause] …]
      structured block
```

This directive is very similar to `#pragma omp parallel for`, but we have to manually specify more clauses. If we're doing just simple things (like prints) we don't really have to specify, but if we have a complex block of code, we will usually have to manually parallelize the code block.

Braces & functions are considered code blocks. We'll generally be using parallel for, less of the generic parallel directive.

## Synchronization

We need synchronization when there are shared accesses to shared data (that could be inconsistent)

```
#pragma omp critical [(name)]
      structured block
```

This block marks a region as mutually exclusive, which specifies that the code block should only be ran one thread at a time. This inherently makes our code more sequential, so use it sparingly. We should do it on smaller chunks of code. If we use it on bigger chunks of code we end up losing out on parallelization.

```
#pragma omp atomic
      expression
```

Same as the critical directive above, but for one specific variable / expression

## GPU Support

OpenMP can now operate on GPU cores. GPUs tend to have a large number of low compute power cores, meaning they have a lot more parallelization potential compared to standard CPU architectures

```
#pragma omp target teams distribute parallel for
      for loop
```

`target` specifies we should run on an accelerator / device. `teams distribute` creates a team of worker threads and distributes work amongst them.

## Parallelizing Matrix Multiplication

Big use case in AI & ML – often done on "tensors". We have 3 2D vectors: 2 input matrices, A and B, followed by 1 output matrix, C.

The dimensions have to match up. So the number of rows in A == number of cols in B, and the number of cols in A == number of rows in B.

Since our systems are usually either row-major or col-major, one of the arrays will result in terrible cache performance. For rows in A, if we have long rows, they might not fit entirely in cache, requiring more

memory usage. For cols in B, we have to access all the different rows, which may not be able to fit in cache entirely

Optimization: chunk matrices into blocks to improve cache performance. Each block has smaller dimensions, which allows for the rows and cols to be stored in cache. When we back out to the outer block logic, we can bring in chunked entries from each row in B when multiplying with A, which reduces the overall number of cache misses in this kind of complex calculation

As such, choosing the right block size is very important!