# ECE 449/590 – OOP and Machine Learning
# Lecture 25 OpenMP

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 21, 2022

## Outline

## Reading Assignment

- This lecture: Introduction to OpenMP
  - Most of today's lecture is based on Tim Mattson's presentation "The OpenMP Common Core: A hands on exploration".
  - The whole presentation and the slides can be found at `https://www.youtube.com/watch?v=I2EaVMjZRRY`
- Next week: no in-person lectures
  - Please watch the video "Introduction to GPU Architecture and Programming Models", Tim Warburton, Virginia Tech `https://www.youtube.com/watch?v=uvVy3CqpVbM`

## Outline

## Parallel and Distributed Computing

- Parallel and distributed computing are two different paradigms to utilize computing resources beyond a single core.
  - Multiple cores available from a modern processor.
  - Multiple machines connected by a network.
- Parallel computing: Single Program Multiple Data (SPMD)
  - All cores the same binary program, though each may work on a different chunk of data.
- Distributed computing: coordination of heterogeneous tasks
  - Different programs run on different cores and machines.
- For both paradigms, we rely on communication mechanisms to pass data around and to coordinate tasks (synchronization).

## Shared Memory and Message Passing

- Communication via shared memory.
  - Data are shared in a common memory space that all cores are able to access.
  - Synchronization is the critical issue – need to use synchronization primitives like mutex and barriers.
- Communication via message passing.
  - Tasks communicate by sending and receiving messages only.
  - Synchronization is implied by the message.
  - Data must be received first before computation may start.
- Which one is more intuitive to work with in software? Which one is more close to actual hardware?

## OpenMP

- Thread: what runs on a single core.
  - A program utilizes multiple cores by creating multiple threads.
  - Threads created from the same program may communicate through shared memory.
  - Multiple threads may run on a single core with the help of operating system, though for best computational efficiency, almost always a single thread is used per core.
- OS APIs to manage threads and to support synchronization are usually too cumbersome for parallel applications.
- OpenMP: An API for Writing Multithreaded Applications
  - A set of compiler directives and library routines for parallel application programmers
  - Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
  - Standardizes established SMP practice + vectorization and heterogeneous device programming

## Outline

## OpenMP Hello World!

```
// hw.cpp
#include <omp.h>
#include <stdio.h>

int main()
{
    #pragma omp parallel
    {
        printf("Hello, world!\n");
    }
    return 0;
}
```

- Compile: `g++ -fopenmp hw.cpp -o hw`
- Run: `./hw`
- Control number of threads: `OMP_NUM_THREADS=16 ./hw`

## Know Who You Are

```
// hw2.cpp
#include <omp.h>
#include <stdio.h>

int main()
{
    #pragma omp parallel
    {
        int n = omp_get_num_threads();
        int i = omp_get_thread_num();
        printf("Thread %d of %d: Hello, world!\n", i, n);
    }
    return 0;
}
```

- Within an `omp parallel` block,
  - `omp_get_num_threads()` tells how many threads are there.
  - `omp_get_thread_num()` tells the index of this thread.

## Anomaly

```
// hw3.cpp
#include <omp.h>
#include <stdio.h>

int main()
{
    #pragma omp parallel
    {
        int n = omp_get_num_threads();
        int i = omp_get_thread_num();
        printf("Thread %d of %d: ", i, n);
        printf("Hello, world!\n");
    }
    return 0;
}
```

- What output do you expect and what output do you see?
- The need for synchronization
  - Synchronization inside `printf()` ensures each call is completed atomically as a single operation.
  - However, different calls from different threads may interleave.
  - What about `std::cout`?

## Fork-Join

```
// hw4.cpp
#include <omp.h>
#include <stdio.h>
int main() {
    #pragma omp parallel
    {
        int n = omp_get_num_threads(); int i = omp_get_thread_num();
        printf("Thread %d of %d: working on task 1\n", i, n);
    }
    printf("Done with task 1\n");
    #pragma omp parallel
    {
        int n = omp_get_num_threads(); int i = omp_get_thread_num();
        printf("Thread %d of %d: working on task 2\n", i, n);
    }
    printf("Done with task 2\n");
    return 0;
}
```

- There may be multiple `omp parallel` blocks and you may have usual sequential code in-between.
  - Only threads inside an `omp parallel` block are executed parallelly, and one must wait for all of them to finish before exiting the block.

## Outline

## Calculating $\pi$

```
// pi.cpp
#include <omp.h>
#include <stdio.h>
int main()
{
    const size_t num_steps= 4000000000LL;
    const double step = 1.0/num_steps;
    double sec = omp_get_wtime();
    double sum = 0;
    for (size_t i = 0; i < num_steps; ++i)
    {
        double x = (i+0.5)*step;
        sum = sum+4.0/(1.0+x*x);
    }
    double pi = step*sum;
    sec = omp_get_wtime()-sec;
    printf("pi = %.16f, time %.3f\n", pi, sec);
    return 0;
}
```

- Compile and optimize: `g++ -fopenmp -O2 pi.cpp -o pi`
- Use `omp_get_wtime()` to get the wall-clock time in seconds.

## Calculating $\pi$ with OpenMP

```
// pi_omp.cpp
...
    const int max_threads = 100;
    double sum[max_threads];
    int num_threads = 0;
    #pragma omp parallel
    {
        int n = std::min(omp_get_num_threads(), max_threads);
        int k = omp_get_thread_num();
        if (k < max_threads) {
            sum[k] = 0;
            for (size_t i = k; i < num_steps; i += n) {
                double x = (i+0.5)*step;
                sum[k] = sum[k]+4.0/(1.0+x*x);
            }
        }
        if (k == 0) num_threads = n;
    }

    double pi = 0;
    for (size_t k = 0; k < num_threads; ++k)
        pi += step*sum[k];
...
```

## Synchronization

```
// pi_syn.cpp
...
    double pi = 0;
    #pragma omp parallel
    {
        int n = omp_get_num_threads();
        int k = omp_get_thread_num();
        double sum = 0;
        for (size_t i = k; i < num_steps; i += n) {
            double x = (i+0.5)*step;
            sum = sum+4.0/(1.0+x*x);
        }
        #pragma omp critical
        {
            pi += step*sum;
        }
    }
...
```

- We may simplify the previous code by using a local `sum`.
  - Update the shared `pi` from within the threads.
  - Use the `omp critical` block to ensure only one thread is allowed to update `pi` as a time.

## Loop Parallelization

```
// pi_loop.cpp
...
    double sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (size_t i = 0; i < num_steps; ++i)
    {
        double x = (i+0.5)*step;
        sum = sum+4.0/(1.0+x*x);
    }
    double pi = step*sum;
...
```

- That loop is such a typical one that OpenMP can parallelize it automatically if you give a little bit of hint.
  - Use `for` in `omp parallel` to request for parallelization.
  - Use `reduction(+: sum)` to tell the compiler `sum` will be generated from the loop as a summation, and appropriate synchronization should be applied.

## Summary and Advice

- OpenMP are widely available and widely used now.
  - Try different settings of `OMP_NUM_THREADS` with our Project 4–6 to see how `numpy` perform differently.
- Many embarrassingly parallel tasks like simple for loops can be parallelized by OpenMP automatically with a little bit of hint.
  - E.g. many tensor operators for machine learning.
  - The challenge is to optimize memory and cache performance.
- There is a function `omp_set_num_threads()` allowing to set how many threads you would like to use.
  - This is seldomly a good idea as consumers have very diverse choices of machines.
  - Your code should work on most settings and you should allow consumers to choose via `OMP_NUM_THREADS`.