ECE 587 – Hardware/Software Co-Design
Lecture 11 OpenMP

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

February 16, 2026

Outline

Reading Assignment

- This lecture: OpenMP
  - Most of today's material is based on Tim Mattson's presentation "The OpenMP Common Core: A hands on exploration" available at `https://www.youtube.com/watch?v=I2EaVMjZRRY`
- Next lecture: Memory Optimizations

Outline

GEMM Parallelization with Threads

```
void gemm_ikj_th(const float* A, const float* B, float* C, std::size_t N) {
    std::fill(C, C + N * N, 0.0f);
    std::vector<std::shared_ptr<std::thread>> threads;
    for (std::size_t i = 0; i < N; ++i) {
        threads.push_back(std::make_shared<std::thread>([&, i] {
            for (std::size_t k = 0; k < N; ++k) {
                float a_ik = A[i * N + k];
                for (std::size_t j = 0; j < N; ++j) {
                    C[i * N + j] += a_ik * B[k * N + j];
                }
            }
        }));
    }
    for (auto& t : threads) t->join();
}
```

- For matrices that are large enough, we could simply parallelize the outermost GEMM loop.
- Create one thread for each iteration of the outer loop, and join them at the end.
- Any issues?

Managing Threads

- While multiple cores in a CPU can be utilized through OS threads, sometimes it is not convenient.
- Threads are managed individually even when they complete very similar tasks.
- Without careful management, there will be a lot of overhead.
  - Creating and destroying thread are costly for OS.
  - Allowing more threads to run concurrently than the number of cores causes costly context switches and cache misses.
- To minimize the overhead we usually use a thread pool that controls how many threads are running concurrently.
  - Any easier way if the computation shows obvious parallelism like GEMM?

From Threads to OpenMP

- On a single machine, OpenMP is more convenient toward parallel computing workloads than threads.
- OpenMP: an API for writing multicore programs
  - A set of compiler directives and library routines for parallel application programmers
  - Greatly simplifies writing multicore programs in Fortran, C and C++
  - Standardizes established SMP practice + vectorization and heterogeneous device programming

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=4 ./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 π

```
// 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 π 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.

GEMM Parallelization with OpenMP

```
void gemm_ikj_omp(const float* A, const float* B, float* C, std::size_t N) {
    std::fill(C, C + N * N, 0.0f);
    #pragma omp parallel for
    for (std::size_t i = 0; i < N; ++i) {
        for (std::size_t k = 0; k < N; ++k) {
            float a_ik = A[i * N + k];
            for (std::size_t j = 0; j < N; ++j) {
                C[i * N + j] += a_ik * B[k * N + j];
            }
        }
    }
}
```

- Can we get linear speedup as number of cores increase?
  - Memory heirarchy consists of multiple levels of caches.
  - Additional cache and memory available for individual cores may lead to superlinear speedup.
  - Cache and memory shared among cores may impact speedup due to limited capacity and bandwidth.

Summary

- OpenMP are widely available and widely used now.
- Many embarrassingly parallel tasks like simple for loops can be parallelized by OpenMP automatically with a little bit of hint.
  - The challenge is to optimize memory and cache performance.