High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 1

Welcome!

CSE 6230 – High Performance Computing: Tools and Applications

- Practical, hands-on course on parallel programming.
- We will develop our skills using real scientific applications.

Forms of parallelism

- Multiple compute nodes connected via a network
- Multiple chips on a compute node
- Accelerators and co-processors on a compute node
- Multiple cores on a chip
- Multiple functional units in a core (leading to instructions that can be performed at the same time)
- SIMD units in a core (same operation on multiple data items at the same time)

Shared memory vs. distributed memory

- Shared memory parallelism: multiple *threads* of a *process* run on a single node
- Distributed memory parallelism: multiple processes run on multiple nodes (e.g., one process per node)

Course Topics

- Review of POSIX threads
- Advanced OpenMP
- Advanced MPI, including nonblocking collectives, one-sided/RMA and MPI shared memory
- Global Arrays, PGAS languages
- Task-based runtime systems
- Hybrid programming (MPI+OpenMP, MPI+MPI)
- SIMD programming with intrinsics
- Intel Xeon Phi (KNC) offloading
- Intel tools and libraries: VTune, MKL, compiler vectorization reports, etc.
- Other programming models, parallel languages, and tools
- Applications in PDE simulations
- Applications in dynamic particle simulations
- Applications in quantum chemistry

Grading

- 20% Exercises. Assigned after most lectures and due approximately 36 hours later. These are designed to help you get hands-on experience with the material in the lecture. Graded on 2 point scale. You can miss two exercises without penalty. *First exercise will be assigned today!*
- 30% Mini-projects. About 3 during the semester.
- 50% Project (with presentation and report). Individual projects chosen from a set of pre-defined research questions given in class.

What you need to succeed in this course

- Desire to learn how to make programs run fast
- Curiosity to investigate performance anomalies
- Expertise in C or C++ programming
- Familiarity with using the Linux command line, including:
  - using shell and environment variables
  - shell scripting
  - git revision control
- Not afraid of matrix operations and reading Matlab code
- Not afraid to get your hands dirty!

Related courses

- CSE 6220 – High Performance Computing
  - emphasis on parallel algorithms
- CSE 6230 – High Performance Computing: Tools and Applications
  - hands-on parallel programming
  - this course!
- CSE 6010 – Computational Problem Solving
  - C programming, data structures, algorithms
  - Module on HPC

Measuring execution time: Best practices

- Measure runs multiple times and report and average (and a measure of the deviation of the deviation is large and cannot be reduced)
- May be allowable to throw out the timing of the first iteration (if the intention is to measure time with a warm cache)
- Be careful of clock granularity if what you are measuring is just a few instructions

Shared memory parallel programming

- POSIX threads
- OpenMP
- Shared memory MPI
- Global arrays (logically shared; physically distributed)
- Many others

POSIX threads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int work(void)
{
    int i, j;
    int sum = 0;
    for (i=0; i<10000; i++)
        for (j=0; j<10000; j++)
            sum++;
    return sum;
}
```

POSIX threads

```c
// signature of thread start routine must be "void *foo(void *data)"
void *thread_worker(void *data)
{
     (void) work();
     printf("%s\n", (char *) data);
     return NULL;
}

int main()
{
     pthread_t thread1, thread2;
     void *thread_return1, *thread_return2;
     int  iret1, iret2;
     char *message1 = "data for thread 1";
     char *message2 = "data for thread 2";

     // spawn threads
     iret1 = pthread_create(&thread1, NULL, thread_worker, (void *) message1);
     iret2 = pthread_create(&thread2, NULL, thread_worker, (void *) message2);

     printf("thread 1, pthread_create: %d\n", iret1);
     printf("thread 2, pthread_create: %d\n", iret2);

     // wait for spawned threads to finish
     iret1 = pthread_join(thread1, NULL);
     iret2 = pthread_join(thread2, NULL);

     printf("thread 1, pthread_join: %d\n", iret1);
     printf("thread 2, pthread_join: %d\n", iret2);

     return 0;
}
```

POSIX threads

Compile using

```
gcc -pthread filename.c
```

C++11 threads

```cpp
#include <iostream>
#include <thread>

void worker(int id) {
    std::cout << "Hello from " << id << std::endl;
}

int main() {
    // declare/construct a variable of type thread
    std::thread t(worker, 5);

    // join thread with main thread
    t.join();

    return 0;
}
```

C++11 threads

- Compile using

```
g++ -std=c++11 -pthread filename.cpp
```

Exercise 1

- Write a program that computes $x = x + \alpha y$ where x and y are input vectors and α is a scalar. The program uses pthreads or C++11 threads to parallelize the computation.
- Graph the computation time vs. number of threads used. For this, consider the following questions:
  - Length of the vectors?
  - Maximum number of threads to use?
  - Best way to perform the timings?
- Submit a short report with the following sections:
  - Listing of your program.
  - Graph of the computation time vs. number of threads used.
  - Discussion on whether or not your results are expected.
- *Due at the beginning of the next class*