EECS 570 Programming Assignment 1

University of Michigan

January 19, 2018
Announcements

- Sign up for final project groups today *before midnight*
  - [https://docs.google.com/spreadsheets/d/17PyzgXuaTSygavqEUqJG9dPD9FUBRft8oUTKdAeqabw/edit?usp=sharing](https://docs.google.com/spreadsheets/d/17PyzgXuaTSygavqEUqJG9dPD9FUBRft8oUTKdAeqabw/edit?usp=sharing)
  - A team must have an identity!
- Project proposal due Wednesday 1/24
Overview

1. Medical Imaging using Ultrasound
   - Introduction
   - Transmission and Reception

2. Intel MIC Architecture
   - Architectural Overview
   - Programming the MIC

3. Introduction to POSIX Threads
   - Thread Creation and Joining
   - Synchronization Primitives
Portable Medical Imaging Devices

- Medical imaging moving towards portability
  - MEDICS (X-Ray CT) [Dasika ’10]
  - Handheld 2D Ultrasound [Fuller ’09]

- Not just a matter of convenience
  - Improved patient health [Gunnarsson ’00, Weinreb ’08]
  - Access in developing countries

- Why ultrasound?
  - Low transmit power [Nelson ’10]
  - No danger or side-effects
Ultrasound: Transmission and Reception

Image Space

Focal Points

Receive Transducer

Transmit Transducer

Receive Raw Channel Data
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Each transducer stores an array of raw received data
Ultrasound: Transmission and Reception

Image reconstructed from data based on round-trip delay
Ultrasound: Transmission and Reception

Images from each transducer combined to produce the full frame.
Delay Index Calculation

- Iterate through all image points for each transducer and calculate delay index $\tau_P$

$$\tau_P = \frac{f_s}{c}(R_P + \sqrt{R_P^2 + X_i^2 - 2R_PX_i \sin \theta})$$

- Often done with lookup tables (LUTs) instead
- 50 GB LUT required for target 3D system
Intel Xeon Phi Coprocessors and the MIC Architecture
Multi-core Intel Xeon processor

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- \( \geq 12 \) cores/socket \( \approx 3 \text{ GHz} \)
- 2-way hyper-threading
- 256-bit AVX vectors

Many-core Intel Xeon Phi coprocessor

- C/C++/Fortran; OpenMP/MPI
- Special Linux \( \mu \text{OS} \) distribution
- 6-16 GB cached GDDR5 RAM
- 57-61 cores at \( \approx 1 \text{ GHz} \)
- 4-way hyper-threading
- 512-bit IMCI vectors
Xeon Phi Programming Models

- Native coprocessor applications
  - Compile with -mmic
  - Run with micnativeloadex or scp+ssh
  - The way to go for MPI applications without offload

- Explicit offload
  - Functions, global variables require __attribute__((target(mic)))
  - Initiate offload, data marshalling with #pragma offload
  - Only bitwise-copyable data can be shared

- Clusters and multiple coprocessors
  - #pragma offload target(mic:i)
  - Use threads to offload to multiple coprocessors
  - Run native MPI applications
Intel MIC Architecture

Programming the MIC

Xeon Phi Programming Models

- Native coprocessor applications
  - Compile with `-mmic`
  - Run with `micnativeloadex` or `scp+ssh`
  - The way to go for MPI applications without offload

- Explicit offload
  - Functions, global variables require `__attribute__((target(mic)))`
  - Initiate offload, data marshalling with `#pragma offload`
  - Only bitwise-copyable data can be shared

- Clusters and multiple coprocessors
  - `#pragma offload target(mic:i)`
  - Use threads to offload to multiple coprocessors
  - Run native MPI applications
Native Execution

Example ("Hello World" application)
#include <stdio.h>
#include <unistd.h>
int main() {
    printf("Hello world! I have %ld logical cores.\n", sysconf(_SC_NPROCESSORS_ONLN ));
}

Example (compile and run on host)
user@host% icc -o hello hello.c
user@host% ./hello
Hello world! I have 32 logical cores.
user@host% _
Native Execution

Compile and run the same code on the coprocessor in native mode:

Example (compile and run on coprocessor)

```sh
user@host% icc -o hello.mic hello.c -mmic
user@host% micnativeloadex hello.mic -t 300 -d 0
Hello world! I have 240 logical cores.
user@host% _
```

- Use `-mmic` to produce executable for MIC architecture
- Use `micnativeloadex` to run the executable on the coprocessor
- Native MPI applications work the same way (need Intel MPI library)
What is a thread?

Independently executing stream of instructions
Schedulable unit of execution for the operating system

Pthreads - the POSIX threading interface
Provides system calls to create and synchronize threads
Communication happens strictly through shared memory
Specifically, using pointers to shared data
What is a thread?

- Independently executing stream of instructions
- Schedulable unit of execution for the operating system
Introduction to POSIX Threads

- What is a thread?
  - Independently executing stream of instructions
  - Schedulable unit of execution for the operating system

- Pthreads - the POSIX threading interface
  - Provides system calls to create and synchronize threads
  - Communication happens strictly through shared memory
    - Specifically, using pointers to shared data
Creating Threads

- Pthread create function signature
  
  ```
  int pthread_create(pthread_t*, const pthread_attr_t*,
                     void* (*)(void*), void*);
  ```

Example

```
errcode = pthread_create(&thread_obj, &thread_attr,
                         &thread_func, &func_arg);
```
Creating Threads

- Pthread create function signature
  ```c
  int pthread_create(pthread_t*, const pthread_attr_t*,
                     void* (*)(void*), void*);
  ```

Example

```c
errcode = pthread_create(&thread_obj, &thread_attr,
                         &thread_func, &func_arg);
```

- `thread_obj` is the thread object or handle (used to halt, etc.)
- `thread_attr` specifies various attributes
  - Default values obtained by passing a NULL pointer
- `thread_func` is a pointer to the function to be run (takes and returns `void*`)
- `func_arg` is a pointer to an argument that is passed to `thread_func` when it starts
- `errcode` is be set to non-zero if `pthread_create()` fails
Shared Data and Threads

- Objects allocated on the heap may be shared (by passing pointers)
- Variables on the stack are private; passing pointers to those between threads can lead to problems
- How to pass multiple arguments to a thread?
  - One way: create a “thread data” struct
  - Pass a pointer to the struct object to each thread

Example

```c
typedef struct _thread_data_t{
    int thread_id, value;
    char* message;
} thread_data_t;
...
thread_data_t td;
/* initialize elements of thread_data_t object */
pthread_create(&thread_obj, NULL, thread_func, &td);
...```
Joining Threads

- Pthread join function signature

  ```c
  int pthread_join(pthread_t thread_obj,
                  void** retval);
  ```

Example

```c
errcode = pthread_join(thread_obj, NULL);
```
Joining Threads

- **Pthread join function signature**
  
  ```c
  int pthread_join(pthread_t thread_obj,
                  void** retval);
  ```

**Example**

```c
errcode = pthread_join(thread_obj, NULL);
```

- The function waits for the thread object `thread_obj` to terminate
- If `retval` is not NULL, then `pthread_join()` copies the exit status
- `errcode` is set to non-zero if `pthread_join()` fails
Multithreaded “Hello World”

Example ("Hello World" application)

```c
void* func(void* arg) {
    printf("Hello World!\n");
    return NULL;
}

int main() {
    pthread_t threads[2]; int i;
    for(i = 0; i < 2; ++i) {
        pthread_create(&threads[i], NULL, func, NULL);
    }
    for(i = 0; i < 2; ++i) {
        pthread_join(threads[i], NULL);
    }
}
```

Compile using gcc -pthread
Let’s run a “Hello World” program through the Phi!
Synchronization Primitives I - Mutexes

- Mutual exclusion (mutex), a.k.a. locks
  - Threads working mostly independently may need to access shared data
    ```c
    mutex *m = alloc_and_init();
    acquire(m);
    /* modify shared data */
    release(m);
    ```
  - e.g. Producer-consumer model
    - Coke machine example: single person refills coke (producer), multiple people buy coke (consumer)

- Is there any problem with holding multiple mutexes?
Mutual exclusion (mutex), a.k.a. locks

- Threads working mostly independently may need to access shared data

```c
mutex *m = alloc_and_init();
acquire(m);
/* modify shared data */
release(m);
```

- e.g. Producer-consumer model
  - Coke machine example: single person refills coke (producer), multiple people buy coke (consumer)

Multiple mutexes may be held, but may lead to deadlock
Synchronization Primitives I - Mutexes

Example (mutex creation)
#include <pthread.h>
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&myMutex, NULL);

Example (mutex usage)
pthread_mutex_lock(&myMutex);
/* access critical data */
pthread_mutex_unlock(&myMutex);

Example (mutex deallocation)
pthread_mutex_destroy(&myMutex);
A barrier object allows global synchronization between threads
- Wait for all threads to reach a point in computation
- After that, launch all threads simultaneously to continue execution

Common when running multiple copies of the same function in parallel
- Single Program Multiple Data (SPMD) paradigm

Simple use of barriers: all threads hit the same barrier
work_on_my_problem();
barrier_wait();
get_data_from_others();
barrier_wait();

More complicated: barriers on branches (or loops)
if(thread_id % 2 == 0) {
  work_on_problem_1();
  barrier_wait();
} else {
  barrier_wait();
}
Synchronization Primitives II - Barriers

Example (static barrier initialization with 3 threads)

```c
pthread_barrier_t barrier = PTHREAD_BARRIER_INITIALIZER(3);
```

Example (dynamic barrier initialization with 3 threads)

```c
pthread_barrier_t myBarrier;
pthread_barrier_init(&myBarrier, NULL, 3);
```

Example (barrier usage)

```c
pthread_barrier_wait(&myBarrier);
```

Example (barrier deallocation)

```c
pthread_barrier_destroy(&myBarrier);
```
Pthreads Summary

- Initialize every pthread object you use
  - e.g. pthread_mutex_t, pthread_barrier_t

- Do not spawn threads for small jobs
  - Thread creation overhead is non-trivial
  - Too many threads can lead to performance degradation (Amdahl’s law)

- Work through a tutorial!
  - https://computing.llnl.gov/tutorials/pthreads/
Questions?
Programming Assignment I due 2/2 11:59 PM