EECS 570 Programming Assignment 1

University of Michigan

January 14, 2022
Announcements

- Sign up for final project groups ASAP
  - [https://docs.google.com/spreadsheets/d/1NDgrDKN5uI5Ve9K8IGd1Gg6222hzBDksut93-9NT0tY/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1NDgrDKN5uI5Ve9K8IGd1Gg6222hzBDksut93-9NT0tY/edit?usp=sharing)
  - A team must have an identity!
- Project proposal due Monday 1/31
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
Medical Imaging using Ultrasound

**Introduction**

**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

Diagram showing the transmission and reception of ultrasound waves with focal points and channels.
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
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Ultrasound: Transmission and Reception
Medical Imaging using Ultrasound

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 $\tau_P$

\[
\tau_P = \frac{f_s}{c} \left( R_P + \sqrt{R_P^2 + X_i^2 - 2R_P X_i \sin \theta} \right)
\]

- Often done with lookup tables (LUTs) instead
- 50 GB LUT required for target 3D system
Intel MIC Architecture

Architectural Overview

Intel Xeon Phi Coprocessors and the MIC Architecture

- Intel Xeon Processor
- PCIe x16
- TCP/IP
- System Memory
- > 50 Cores
- Knights Corner
- Linux OS
- >= 8GB GDDR5 memory

University of Michigan
EECS 570 Programming Assignment 1
January 14, 2022
Intel MIC Architecture

Architectural Overview

Intex Xeon Processors and the MIC Architecture

Multi-core Intel Xeon processor

- C/C++/Fortran; OpenMP/MPI
- Standard Linux OS
- Up to 768 GB of DDR3 RAM
- ≥ 12 cores/socket ≈ 3 GHz
- 2-way hyper-threading
- 256-bit AVX vectors

Many-core Intel Xeon Phi coprocessor

- C/C++/Fortran; OpenMP/MPI
- Special Linux μOS distribution
- 6-16 GB cached GDDR5 RAM
- 57-61 cores at ≈ 1 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
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)

```c
#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)

```bash
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?
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
  
  ```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);
```
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
- errorcode is 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

\[
\text{int pthread_join(pthread_t thread_obj, } \\
\text{void** retval);}
\]

Example

\[
\text{errcode = pthread_join(thread_obj, NULL);} \\
\]

- 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`
Demo

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
    ```
    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?
Introduction to POSIX Threads

Synchronization Primitives

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)

- Multiple mutexes may be held, but may lead to deadlock

```
Thread A
lock(a) 1
lock(b) 2
Thread B
lock(b) 3
lock(a) 4
```
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 2/4 11:59 PM on Canvas
Vectorization (Single Instruction Multiple Data, SIMD, Parallelism)
SIMD Operations

SIMD — Single Instruction Multiple Data

Scalar Loop

```
1  for (i = 0; i < n; i++)
```

SIMD Loop

```
1  for (i = 0; i < n; i += 4)
2       A[i:(i+4)] = A[i:(i+4)] + B[i:(i+4)];
```

Each SIMD addition operator acts on 4 numbers at a time.
# Instruction Sets in Intel Architectures

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Year and Intel Processor</th>
<th>Vector registers</th>
<th>Packed Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMX</td>
<td>1997, Pentium</td>
<td>64-bit</td>
<td>8-, 16- and 32-bit integers</td>
</tr>
<tr>
<td>SSE</td>
<td>1999, Pentium III</td>
<td>128-bit</td>
<td>32-bit single precision FP</td>
</tr>
<tr>
<td>SSE2</td>
<td>2001, Pentium 4</td>
<td>128-bit</td>
<td>8 to 64-bit integers; SP &amp; DP FP</td>
</tr>
<tr>
<td>SSE3–SSE4.2</td>
<td>2004 – 2009</td>
<td>128-bit</td>
<td>(additional instructions)</td>
</tr>
<tr>
<td>AVX</td>
<td>2011, Sandy Bridge</td>
<td>256-bit</td>
<td>Single and double precision FP</td>
</tr>
<tr>
<td>AVX2</td>
<td>2013, Haswell</td>
<td>256-bit</td>
<td>(additional instructions)</td>
</tr>
<tr>
<td>IMCI</td>
<td>2012, Knights Corner</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
<tr>
<td>AVX-512</td>
<td>(future) Knights Landing</td>
<td>512-bit</td>
<td>32- and 64-bit integers; single &amp; double precision FP</td>
</tr>
</tbody>
</table>
Explicit Vectorization: Compiler Intrinsics

SSE2 Intrinsics

```
for (int i=0; i<n; i+=4) {
    __m128 Avec = _mm_load_ps(A+i);
    __m128 Bvec = _mm_load_ps(B+i);
    Avec = _mm_add_ps(Avec, Bvec);
    _mm_store_ps(A+i, Avec);
}
```

IMCI Intrinsics

```
for (int i=0; i<n; i+=16) {
    __m512 Avec = _mm512_load_ps(A+i);
    __m512 Bvec = _mm512_load_ps(B+i);
    Avec = _mm512_add_ps(Avec, Bvec);
    _mm512_store_ps(A+i, Avec);
}
```

- The arrays float A[n] and float B[n] are aligned on a 16-byte (SSE2) and 64-byte (IMCI) boundary
- n is a multiple of 4 for SSE and a multiple of 16 for IMCI
- Variables Avec and Bvec are
  128 = 4 × sizeof(float) bits in size for SSE2 and
  512 = 16 × sizeof(float) bits for the Intel Xeon Phi architecture