18-447 Lecture 22:
1 Lecture Worth of Parallel Programming Primer

James C. Hoe
Department of ECE
Carnegie Mellon University

Housekeeping

- Your goal today
  - see basic concepts in shared-memory multithreading
  - appreciate how easy parallel programming can be
  - appreciate how difficult “good” parallel programming can be

- Notices
  - Final Exam: Thursday, May 11, 2017, 5:30pm~8:30p.m
  - graded Midterm 2 and solutions at Course Hub
  - regrades accepted until Wed 4/26

- Readings
  - P&H Ch 6
Shared-Memory Multicores

- Current general-purpose multicore microprocessors are based on shared memory
  - individual cores follow classic von Neuman
  - cores have access to a common physical address space and memory
  - processes/threads on different cores can communicate by writing and reading agreed-upon memory locations

Single Program Multiple Data

- SPMD is MIMD except all threads based on the same program image
- In an SMP, execution starts as a single-thread process and its memory
  - independent “threads of execution” (think program counters, regfile and stacks) spawned within the same process memory
  - different threads run on different cores concurrently (to speedup a task)
  - same effective address in different threads refer to same program and data locations

Many other choices; this is prevalent and easy to start on
E.g., POSIX Threads Create and Join

```c
long count=0; // globals are shared

void *foo(void *arg) { return count = count + (long)arg; }

int main(){
    pthread_t tid[HOWMANY]; // array of thread IDs
    long i;
    void *retval;

    // spawn children threads
    for(i=0; i<HOWMANY; i++ )
        pthread_create( &tid[i], // ID to be set
                        NULL, // attribute (default)
                        foo, // fxn to run by thread
                        (void*)i); // ptr-size arg to fxn

    // wait for children threads to exit
    for (i=0; i<HOWMANY; i++ )
        pthread_join( tid[i], // ID to wait on
                      &retval); // ptr-size return value
}
```

Memory Consistency

- A memory consistency model says for each read which write bound the value to be returned
  - intuitively: a read should return the value of the “most recent” write to the same memory address
  - straight forward for a single thread
- Suppose in a shared-memory multicore, cores C1/C2/C3 perform their respective streams of reads and writes
  - C1: ........W(x) ........
  - C2: ....W(x), W(x), W(y), R(x), R(y) ...
  - C3: ........W(x), W(y), W(x) ...

Who performed the last write to x before R(x) by C2?

- How do you establish a global ordering of memory reads and writes? Do you need one?
Sequential Consistency (SC)

- A thread perceives its own memory ops in program order (of course)
- Memory ops from different threads can be interleaved arbitrarily; different interleaving are allowed on different runs, i.e., nondeterminism
- But for each run, all threads must not disagree on any orderings observed

Switch Model:

```
Switch Model: P0 P1 P2 ... Pn-1
```

Example: what can and cannot be

- Threads T1 and T2 and shared locations X and Y (initially X = 0, Y = 0)

  T1: ................................................ T2: ................................................
  store(X, 1);  vy = load(Y);
  store(Y, 1);  vx = load(X);
  ....  ....

- SC says
  - vy and vx may get different values from run to run e.g., (vy=0, vx=0), (vy=0, vx=1), or (vy=1, vx=1)
  - but if vy is 1 then vx cannot be 0
An Useful Example

- Consider threads T1 and T2 communicating via shared memory locations X and Y
  - T1 produces a result in X to be consumed by T2
  - T1 signals readiness to T2 by setting Y

T1:
  
  Y is initially 0
  ....
  compute v
  store (X, v)
  store (Y, 1)
  ....

T2:
  
  ......
  do {
  ready=load Y
  } while (!ready)
  data = load X
  ......

- This works because SC says T1 and T2 must see the stores to X and Y in the same order

This is so natural and obvious, why talk about it?

Easy to think about hard to build

- Where is the “point of serialization” for SC if memory ops don’t always go onto the bus?
- SC restricts many memory reordering optimizations taken-for-granted in sequential programming
  e.g., can a pipeline continue past a cache-write-miss on X to start a read (to Y) later in the program order
Weak Consistency (WC)

- WC processors only obey basic uniprocessor memory dependence, i.e., $R_i(x) < W_i(x)$, $W_i(x) < R_j(x)$, $W_i(x) < W_j(x)$
- Program insert explicit memory fence instructions to force serialization when it matters

\[
\begin{align*}
T1: & \quad Y \text{ is initially 0} \\
& \quad \text{......} \quad \text{......} \\
& \quad \text{compute } V \\
& \quad \text{store } (X, V) \\
& \quad \text{fence} \\
& \quad \text{store } (Y, 1) \\
T2: & \quad \text{......} \quad \text{do \{} \\
& \quad \text{ready=load } Y \\
& \quad \text{fence} \\
& \quad \text{data = load } X
\end{align*}
\]

- Assuming serialization is rare, low-cost (low-performance) fences okay, e.g., on a fence drain/restart pipeline

A range of intermediate models between SC and WC exist

Embarrassingly Parallel Processing

- Summing 10,000 numbers in an array $A[]$
- In sequential algorithm
  \[
  \text{for } (i=0; i<10000; i=i+1) \\
  \text{sum = sum } + \ A[i];
  \]
- Ideally
  - assuming “+” takes 1 unit time and everything else free
  - $T_1 = 10,000$
  - $T_\infty = \lceil \log_2 10,000 \rceil = 14$ using binary reduction
  - $T_1/T_\infty = 714$

  recall if $T_1/T_\infty >> P$ then $S \approx P$

- So at $P=100$ << $T_1/T_\infty$
  - expect $T_{P=100} \approx T_1/P=100$ or $S_{P=100} \approx P=100$
Shared-Memory Pthreads Strategy

- Fork $P=100$ threads on a $P$-way shared memory multiprocessor
  - $A[10000]$ is in shared memory
  - $psum[100]$ is also in shared memory

- Child thread $i$ uses $psum[i]$ to compute its portion of the partial sum

- When all threads finish, parent sums $psum[0] \sim psum[99]$

Children Thread Code

```c
double A[ARRAY_SIZE];
double psum[P];

void *sumParallel(void *id) {
    long id=(long) id;
    long i;

    psum[id]=0;
    for(i=0;i<(ARRAY_SIZE/P);i++)
        psum[id]+=A[id*(ARRAY_SIZE/P) + i];

    return NULL;
}
```
Parent Code

double A[ARRAY_SIZE];
double psum[P];
double sum=0;
int main(){

    ... skipped pthreads boilerplate ...

    for(i=0; i<P; i++ )
        pthread_create( &tid[i],
                        NULL,
                        sumParallel,
                        (void*)i);

    for (i=0; i<P; i++ ) {
        pthread_join( tid[i], &retval);
        sum+=psum[i];
    }
}
Amdahl’s Law

- A program is rarely completely parallelizable
- Say a fraction $f$ is “perfectly” parallelizable by a factor of $p$ and the rest is sequential

$\text{time}_{\text{parallel}} = (1 - f) + \frac{f}{p}$

$S_{\text{parallel}} = \frac{1}{(1 - f) + \frac{f}{p}}$

Amdahl's Law says parallel speedup bound by $1/(1-f)$

Strategy 2: parallelizing the reduction

- How about asking each thread to do a bit of the reduction, i.e.,

```c
void *sumParallel(void *id) {
    long id=(long) id;
    long i;

    psum[id]=0;

    for(i=0;i<ARRAY_SIZE/P;i++)
        psum[id]+=A[id*ARRAY_SIZE/P+i];

    sum=sum+psum[id];

    return NULL;
}
```
Data Races

- On the last slide, sum is a shared variable read and updated by all threads at roughly the same time.
- Let’s try just 2 threads T1 and T2, sum is initially 0.

T1: compute v
    temp = load sum
    temp = temp + v
    store (sum, temp)

T2: compute w
    temp = load sum
    temp = temp + w
    store (sum, temp)

- What are the possible final values of sum?
  - v + w or v or w depending on the interleaving of the read/modify/write sequence in T1 and T2.
- To work as intended, RMW sequence needs to be uninterrupted, that is, between the time one thread reads sum and write back a new value, no other threads should read or write sum.

Critical Sections

- Special “lock” variables and acquire/release operators can mark off a “critical section” that only one thread can enter at a time, e.g.,
  ```
  pthread_mutex_lock(&lockvar);
  sum = sum + psum[id];
  pthread_mutex_unlock(&lockvar);
  ```
- Lock (aka acquire) and unlock (aka release)
  - call to lock() blocks until the lock variable is freed (released by the previous owner)
  - on an unlock, if multiple calls to lock() pending, only 1 should succeed; the rest keep waiting
- Strategy 2 is now correct but actually slower than before.

Reduction is distributed but still sequential, plus extra overhead cost of locking and unlocking.
Strategy 3:
parallel associative/commutative reduction

// at the end of SumParallel()
remain=100;
do {
    pthread_barrier_wait(&barrier);
    half=(remain+1)/2;
    if (id<(remain/2))
        psum[id]=psum[id]+psum[id+half];
    remain=half;
} while (remain>1);

Performance Analysis

- Summing 10,000 on 100 cores
  - 100 threads performs 100 +’s each in parallel, and
  - between 1~7 (plus a few) +’s each in the parallel reduction
  - $T_{100} = 100 + 7$
  - $S_{100} = 93.5$
- If summing 100,000 on 100 cores
  - $T_{100} = 1000 + 7$
  - $S_{100} = 99.3$
- If summing 10,000 on 10 cores
  - $T_{10} = 1000 + 4$
  - $S_{10} = 10.0$

Still too optimistic, communication is not free!!!
Message Passing

- Each processor has its own private address space and memory
- Threads on different processors communicate by explicit sending and receiving of messages

![Diagram of a message passing setup with interconnect and processors P0, P1, P2, Pk, M, NIU, tx fifo, rx fifo]

Matched Send and Receive

```c
if (id==0)        //assume node-0 has A initially
   for (i=1;i<100;i=i+1)
      SEND(i, &A[100*i], 100*sizeof(double));
else
   RECEIVE(0,A[])   //receive into local array
sum=0;
for(i=0;i<100;i=i+1) sum=sum+A[i];
remain=100;
do {
   BARRIER();
   half=(remain+1)/2;
   if (id>=half&&id<remain) SEND(id-half,sum,8);
   if (id<(remain/2)) {
      RECEIVE(id+half,&temp);
      sum=sum+temp;
   }
   remain=half;
} while (remain>1);
```
[based on P&H Ch 6 example]
Communication Cost

- Communication cost is a part of parallel execution

- Easier to perceive communication cost in message passing
  - overhead: takes time to send and receive data
  - latency: takes time for data to go from A to B
  - gap (1/bandwidth): takes time to push successive data through a finite bandwidth

- Same cost was also there in shared memory

To be continued . . . .