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 (context for topics to come)
  – appreciate how easy parallel programming can be
  – appreciate how difficult “good” parallel programming can be

• Notices
  – HW5, due Friday 4/26 midnight
  – get going on Lab 4, now less than 2 weeks left

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

- Today’s general-purpose multicore processors are MIMD, symmetric, shared memory
  - individual cores follow classic von Neuman
  - common access to physical address space and mem
  - threads on different cores communicate by writing and reading agreed-upon mem locations
Single Program Multiple Data

- SPMD is MIMD except all threads based on the same program image
- On SMP, SPMD starts as a single-thread process and its memory
- Independent “threads of execution” (think program counters, regfile and stacks) spawned
  - **same process memory**—same EA in different threads refers to shared program and data locations
  - different threads run concurrently (on different cores) or interleaved

SPMD just one of many options; prevalent and easy to start on
E.g., POSIX Threads Create and Join

```c
long count = 0;         // globals are in memory and 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

• Memory consistency model says for each read which write bound the value to be returned
  – intuitively: a read should return value of “most recent” write to the same address
  – straight forward for a single thread

• In a shared-memory multicore, cores C1/C2/C3 perform following streams of reads and writes

  C1:     . . . . . . W(x) . . . . . .
  C2:  . . . W(x), W(x), W(y), R(x), R(y) . .
  C3:    . . . W(y), W(x), W(y), W(x) . .

Which is the last write to x before R(x) by C2?

Ordering determines what can be seen by reads, but what is observed by reads determines ordering!!
Sequential Consistency (SC)

• A thread perceives its own memory ops in program order (of course)

• Memory ops from threads in program order can be interleaved arbitrarily; different interleaving allowed on different runs, i.e., nondeterminism

• For each run, all threads must not disagree on any orderings observed

• Switch Model:

```
C_0  C_1  C_2          C_{n-1}
```

point of serialization
SC Example: what can and cannot be

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

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{store}(X, \ 1); )</td>
<td>( \text{vy} = \text{load}(Y); )</td>
</tr>
<tr>
<td>( \text{store}(Y, \ 1); )</td>
<td>( \text{vx} = \text{load}(X); )</td>
</tr>
</tbody>
</table>

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

- Threads **T1** and **T2** communicate via shared memory locations **X** and **Y**
  - **T1** produces result in **X** to be consumed by **T2**
  - **T1** signals readiness to **T2** by setting **Y**

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y is initially 0</td>
<td>.....</td>
</tr>
<tr>
<td>.....</td>
<td>ready=load Y</td>
</tr>
<tr>
<td>compute ( v )</td>
<td>do {</td>
</tr>
<tr>
<td>store ((X, v))</td>
<td>} while (!ready)</td>
</tr>
<tr>
<td>store ((Y, 1))</td>
<td>data = load X</td>
</tr>
<tr>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

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

18-447-S24-L22-S9, James C. Hoe, CMU/ECE/CALCM, ©2024
Easy to think about hard to build

- Where is “point of serialization” if memory ops don’t always go to memory or even onto a “bus”?
- SC restricts many memory reordering optimizations taken-for-granted in sequential execution (e.g., non-blocking miss)
Weak Consistency (WC)

- WC imposes only uniprocessor memory ordering requirements: \( R(x) < W(x); W(x) < R(x); W(x) < W(x) \)
- Program inserts explicit memory fence instructions to force serialization when it matters

![Diagram showing memory operations and fences]

- If serialization is rare, cheap(hw)/slow fences okay, e.g., fully drain/restart pipeline and buffers
- Intermediate models exist between SC and WC
Embarrassingly Parallel Processing

• Summing 10,000 numbers from array $A[]$

• In sequential algorithm

```
for (i=0; i<10000; i=i+1)
    sum = sum + $A[i];$
```

• Assuming “+” is 1 unit-time; everything else free
  
  - $T_1$=10,000
  - $T_\infty = \lceil \log_2 10,000 \rceil = 14$ (using associativity of “+”)
  - $P_{avg} = \frac{T_1}{T_\infty} = 714$

• Ideally, at $p=100 << \frac{T_1}{T_\infty}$
  
  expect $T_{100} \approx \frac{T_1}{p}=100$ or $S_{100} \approx p=100$

recall if $\frac{T_1}{T_\infty} \gg p$ then $S \approx p$
Shared-Memory Pthreads Strategy 1

- 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 $psum[0] \sim psum[99]$
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];
    }
}
**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;
}
```

*This looks data parallel?*
Performance Analysis

- Summing 10,000 numbers on 100 cores
  - 100 threads performs 100 +’s each in parallel
  - parent thread performs 100 +’s sequentially
  - $T_{100} = 100 + 100$
  - $S_{100} = 50$

- If 100,000 num on 100 cores
  - $T_{100} = 1000 + 100$
  - $S_{100} = 90.9$

- If 10,000 num on 10 cores
  - $T_{10} = 1000 + 10$
  - $S_{10} = 9.9$

- Don’t forget,
  - *fork* and *join* are not free
  - moving data (even thru shared memory) not free
Amdahl’s Law: a lesson on speedup

- If only a fraction $f$ (of time) is speedup by $s$

$$\text{time}_{\text{improved}} = \text{time}_{\text{original}} \cdot \left( (1-f) + \frac{f}{s} \right)$$

$$S_{\text{effective}} = \frac{1}{(1-f) + \frac{f}{s}}$$

- if $f$ is small, $s$ doesn’t matter

- even when $f$ is large, diminishing return on $s$; eventually “1-$f$” dominates
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;
}
```

Assume SC for simplicity
Data Races

- On last slide \textit{sum} is read and updated by all threads at around the same time.
- Let’s try just 2 threads T1 and T2, \textit{sum} is initially 0.

<table>
<thead>
<tr>
<th>T1: compute \textit{v}</th>
<th>T2: compute \textit{w}</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp=load \textit{sum}</td>
<td>temp=load \textit{sum}</td>
</tr>
<tr>
<td>temp=temp+\textit{v}</td>
<td>temp=temp+\textit{w}</td>
</tr>
<tr>
<td>store (\textit{sum}, temp)</td>
<td>store (\textit{sum}, temp)</td>
</tr>
</tbody>
</table>

- What are the possible final values of \textit{sum}?
  - \textit{v}+\textit{w} or \textit{v} or \textit{w} depending on the interleaving of the read/modify/write sequence in T1 and T2.
- To work, RMW regions needs to be \textit{atomic}

\textit{i.e., no intervening reads/writes by other threads}.
Critical Sections

- Special “lock” variables and lock/unlock operators to demarcate a “critical section” that only one thread can enter at a time, e.g.,

```c
pthread_mutex_lock(&lockvar);
sum = sum + psum[id]; // atomic RMW
pthread_mutex_unlock(&lockvar);
```

- `lock()` blocks until `lockvar` is free or freed (released by previous owner)
- on `unlock()`, if multiple `lock()` pending, only 1 should succeed; the rest keep waiting
- Strategy 2 is now correct but actually slower

Reduction still sequential plus extra cost of locking and unlocking
Strategy 3: Parallel Reduction
(assume “+” associative and commutative)

// at the end of sumParallel()
remain=p;

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 +’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
  – $T_{10} = 1000 + 4$
  – $S_{10} = 10.0$

First-order analysis! Don’t bet on this.
Message Passing

- Private address space and memory per processor
- Parallel threads on different processors communicate by explicit sending and receiving of messages
Example using Matched Send/Receive

```c
if (id==0) // assume node-0 has A initially
    for (i=1;i<p;i=i+1)
        SEND(i, &A[SHARE*i], SHARE*sizeof(double));
else
    RECEIVE(0,A[]) // receive into local array

sum=0;
for(i=0;i<SHARE;i=i+1) sum=sum+A[i];

remain=p;
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);

SHARE=HOWMANY/p
```

[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 . . . . .