Jaguar's days as a CPU-only supercomputer are numbered. Over the next year, the 2.3 petaflop machine at the Oak Ridge National Lab (ORNL) will be upgraded by Cray with the new NVIDIA "Kepler" GPUs, [...] The transformed supercomputer will be renamed Titan and should deliver in the neighborhood of 20 peak petaflops sometime in late 2012.

"It's all about power efficiency"

CS 61C: Great Ideas in Computer Architecture (Machine Structures)
Lecture 22
Thread Level Parallelism III

Instructors:
Michael Franklin
Dan Garcia
http://inst.eecs.Berkeley.edu/~cs61c/Fa11
Review

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor (Multicore) uses Shared Memory (single address space)
- Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern
- MOESI Protocol ensures cache consistency and has optimizations for common cases.

Threads

- thread of execution: smallest unit of processing scheduled by operating system
- Threads have their own state or context:
  - Program counter, Register file, Stack pointer,
- Threads share: a memory address space
- Note: A “process” is a heavier-weight construct, which has its own address space. A process typically contains one or more threads.
  - Not to be confused with a processor, which is a physical device (i.e., a core)
Multithreading

- On a **single processor**, multithreading occurs *by time-division multiplexing*:
  - Processor switched between different threads
    - may be “pre-emptive” or “non pre-emptive”
  - *Context switching* happens frequently enough that user perceives threads as running at the same time
- On a **multiprocessor**, threads run at the same time, with each processor running a thread
Multithreading vs. Multicore

- Basic idea: Processor resources are expensive and should not be left idle
- For example: Long latency to memory on cache miss?
  - Hardware switches threads to bring in other useful work while waiting for cache miss
  - Cost of thread context switch must be much less than cache miss latency
- Put in redundant hardware so don’t have to save context on every thread switch:
  - PC, Registers, ...
- Attractive for apps with abundant TLP

Data Races and Synchronization

- Two memory accesses form a data race if from different threads, to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread ran first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
Lock and Unlock Synchronization

- Lock used to create region (critical section) where only one thread can operate
- Given shared memory, use memory location as synchronization point: lock or semaphore
- Thread reads lock to see if it must wait, or OK to go into critical section (and set to locked)
  - 0 => lock is free / open / unlocked / lock off
  - 1 => lock is set / closed / locked / lock on

Set the lock
Critical section (only one thread gets to execute this section of code at a time)
e.g., change shared variables
Unset the lock

Possible Lock/Unlock Implementation

- Lock (aka busy wait):
  ```
  addiu $t1,$zero,1  ; t1 = 1 means Locked
  Loop:  lw $t0,lock($s0)  ; load lock
         bne $t0,$zero,Loop ; loop if locked
  Lock:  sw $t1,lock($s0)  ; Unlocked, so lock
  ```
- Unlock:
  ```
  sw $zero,lock($s0)
  ```
- Any problems with this?
Peer Instruction: What Happens?

```
addiu $t1,$zero,1 ; t1 = 1 means locked
Loop: lw $t0, lock($s0) ; load lock
       bne $t0, $zero, Loop ; loop if lock <> 0
Lock: sw $t1, lock($s0) ; set lock and continue
       to critical section
```

I. It works great! Ensures that at most one thread enters the critical section at a time.

II. Infinite Loop, since no change to lock before bne

III. Doesn’t work because a different thread on another core could see lock == 0 before sw changes it to 1; so both go to critical section

IV. Doesn’t work because a different thread on this same core could see lock == 0 before sw changes it to 1; so both go to critical section

(A) I only  (B) II only  (C) III only  (D) IV only  (E) III and IV

Possible Lock Problem

- Thread 1
  
  addiu $t1,$zero,1  
  Loop: lw $t0, lock($s0)  
  bne $t0,$zero,Loop  
  Lock: sw $t1,lock($s0)

- Thread 2
  
  addiu $t1,$zero,1  
  Loop: lw $t0,lock($s0)  
  bne $t0,$zero,Loop  
  Lock: sw $t1,lock($s0)

Both threads think they have set the lock
Exclusive access not guaranteed!
Help! Hardware Synchronization

- Hardware support required to prevent interloper (either thread on other core or thread on same core) from changing the value
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write

- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

Synchronization in MIPS

- Load linked: \texttt{ll rt,offset(rs)}
- Store conditional: \texttt{sc rt,offset(rs)}
  - Succeeds if location not changed since the \texttt{ll}
    - Returns 1 in rt (clobbers register value being stored)
  - Fails if location has changed
    - Returns 0 in rt (clobbers register value being stored)

- Example: atomic swap (to test/set lock variable)

Exchange contents of reg and mem: \texttt{$s4 \leftrightarrow ($s1)}

\texttt{try: add $t0,$zero,$s4 ;copy exchange value}
\texttt{1l $t1,0($s1) ;load linked}
\texttt{sc $t0,0($s1) ;store conditional}
\texttt{beq $t0,$zero,try ;branch store fails}
\texttt{add $s4,$zero,$t1 ;put load value in $s4}
Test-and-Set

- In a single atomic operation:
  - **Test** to see if a memory location is set (contains a 1)
  - **Set** it (to 1) if it isn’t (it contained a zero when tested)
  - Otherwise indicate that the Set failed, so the program can try again
  - No other instruction can modify the memory location, including another Test-and-Set instruction

- Useful for implementing lock operations

---

Test-and-Set in MIPS

- Single atomic operation
- Example: MIPS sequence for implementing a T&S at ($s1)

  **Try:**
  ```
  addiu $t0,$zero,1
  li  $t1,0($s1)
  bne $t1,$zero,Try
  sc  $t0,0($s1)
  beq $t0,$zero,try
  **Locked:**
  critical section
  sw $zero,0($s1)
  ```
What is OpenMP?

• API used for multi-threaded, shared memory parallelism
  – Compiler Directives
  – Runtime Library Routines
  – Environment Variables
• Portable
• Standardized
Shared Memory Model with Explicit Thread-based Parallelism

• Shared memory process consists of multiple threads, explicit programming model with full programmer control over parallelization

• Pros:
  – Takes advantage of shared memory, programmer need not worry (that much) about data placement
  – Programming model is “serial-like” and thus conceptually simpler than alternatives (e.g., message passing/MPI)
  – Compiler directives are generally simple and easy to use
  – Legacy serial code does not need to be rewritten

• Cons:
  – Codes can only be run in shared memory environments!
  – Compiler must support OpenMP (e.g., gcc 4.2)

OpenMP Programming Model

• Fork - Join Model:

  • OpenMP programs begin as single process: master thread;
  Executes sequentially until the first parallel region construct is encountered
  – FORK: the master thread then creates a team of parallel threads
  – Statements in program that are enclosed by the parallel region construct are executed in parallel among the various team threads
  – JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
OpenMP Directives

- **FORK**
  - **DO for loop**
  - **JOIN**
  - shares iterations of a loop across the team

- **FORK**
  - **SECTIONS**
  - **JOIN**
  - each section executed by a separate thread

- **FORK**
  - **SINGLE**
  - **JOIN**
  - serializes the execution of a thread

---

**Building Block: C for loop**

```c
for (i=0; i<max; i++) zero[i] = 0;
```

- Break *for loop* into chunks, and allocate each to a separate thread
  - E.g., if max = 100, with two threads, assign 0-49 to thread 0, 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e., No break, return, exit, goto statements
OpenMP Extends C with Pragmas

- Pragmas are a mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes
- Good mechanism for OpenMP because compilers that don’t recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

OpenMP: Parallel for pragma

```c
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is `private` per thread (Why?)
- Implicit synchronization at end of for loop
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, ..., (max/n)-1;
  - Thread 1 gets max/n, max/n+1, ..., 2*(max/n)-1
  - Why?
Thread Creation

- How many threads will OpenMP create?
- Defined by OMP_NUM_THREADS environment variable (or in code procedure call)
- Set this variable to the maximum number of threads you want OpenMP to use
- Usually equals the number of cores in the underlying HW on which the program is run

OMP_NUM_THREADS

- Shell command to set number threads:
  ```
  export OMP_NUM_THREADS=x
  ```
- Shell command check number threads:
  ```
  echo $OMP_NUM_THREADS
  ```
- OpenMP intrinsic to set number of threads:
  ```
 omp_num_threads(x);
  ```
- OpenMP intrinsic to get number of threads:
  ```
  num_th = omp_get_num_threads();
  ```
- OpenMP intrinsic to get Thread ID number:
  ```
  th_ID = omp_get_thread_num();
  ```
Parallel Threads and Scope

Each thread executes a copy of the code within the structured block

```c
#include <omp.h>
main () {
    int nthreads, tid;
    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        /* Only master thread does this */
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
}
```

Matrix Multiply in OpenMP

```c
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, i, j, k)
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        for (k=0; k<Pdim; k++){
            /* C(i,j) = sum(over k) A(i,k) * B(k,j) */
            tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));
        }
        *(C+(i*Ndim+j)) = tmp;
    }
} /* All threads join master thread and terminate */
run_time = omp_get_wtime() - start_time;
```
Notes on Matrix Multiply Example

More performance optimizations available
• Higher compiler optimization (-O2, -O3) to reduce number of instructions executed
• Cache blocking to improve memory performance
• Using SIMD SSE3 Instructions to raise floating point computation rate

And in Conclusion, ... 

• Sequential software is slow software
  – SIMD and MIMD only path to higher performance
• Multiprocessor/Multicore uses Shared Memory
  – Cache coherency implements shared memory even with multiple copies in multiple caches
  – False sharing a concern; watch block size!
• Data races lead to subtle parallel bugs
• Synchronization via atomic operations:
  – MIPS does it with Load Linked + Store Conditional
• OpenMP as simple parallel extension to C
  – Threads, Parallel for, private, critical sections, ...
OpenMP Pitfall #1: Data Dependencies

- Consider the following code:
  
  ```
  a[0] = 1;
  for(i=1; i<5; i++)
  a[i] = i + a[i-1];
  ```

- There are dependencies between loop iterations
- Sections of loops split between threads will not necessarily execute in order
- Out of order loop execution will result in undefined behavior
OpenMP Pitfall #2: Avoiding Dependencies by Using Private Variables

• Consider the following loop:

```c
#pragma omp parallel for
{
    for(i=0; i<n; i++){
        temp = 2.0*a[i];
        a[i] = temp;
        b[i] = c[i]/temp;
    }
}
```

• Threads share common address space: will be modifying temp simultaneously; solution:

```c
#pragma omp parallel for private(temp)
{
    for(i=0; i<n; i++){
        temp = 2.0*a[i];
        a[i] = temp;
        b[i] = c[i]/temp;
    }
}
```

OpenMP Pitfall #3: Updating Shared Variables Simultaneously

• Now consider a global sum:

```c
for(i=0; i<n; i++)
    sum = sum + a[i];
```

• This can be done by surrounding the summation by a critical section, but for convenience, OpenMP also provides the reduction clause:

```c
#pragma omp parallel for reduction(+:sum)
{
    for(i=0; i<n; i++)
        sum = sum + a[i];
}
```

• Compiler can generate highly efficient code for reduction
OpenMP Pitfall #3: Parallel Overhead

• Spawning and releasing threads results in significant overhead
• Therefore, you want to make your parallel regions as large as possible
  – Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)
  – Coarse granularity is your friend!