Tieing the Threads Together
Review

- Sequential software is slow software
  - SIMD and MIMD are paths to higher performance
- MIMD thru: multithreading processor cores (increases utilization), Multicore processors (more cores per chip)
- OpenMP as simple parallel extension to C
  - Pragmas for forking multiple Threads
  - ≈ C: small so easy to learn, but not very high level and it’s easy to get into trouble
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 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.
- (more later)
Lock Synchronization

• Use a “Lock” to grant access to a region (critical section) so that only one thread can operate at a time
  • Need all processors to be able to access the lock, so use a location in shared memory as the lock

• Processors read lock and either wait (if locked) or set lock and go into critical section
  • 0 means lock is free / open / unlocked / lock off
  • 1 means lock is set / closed / locked / lock on
Alternate Synchronization: Semaphore

- A semaphore $s$ has a value
  - $v(s)$
    - Atomic: Increments the semaphore by 1
  - $p(s)$
    - Atomic: Check if value $> 0$ and if so, reduce by 1 and proceed, else retry
- $v$ and $p$ are Dutch in origin (although the names are conflicting)
Synchronization in MIPS

- **Load linked:** \texttt{ll rt,\text{off}(rs)}
- **Store conditional:** \texttt{sc rt,\text{off}(rs)}
  - Returns \texttt{1} (success) if location has not changed since the \texttt{ll}
  - Returns \texttt{0} (failure) if location has changed

- **Note that** \texttt{sc clobbers} the register value being stored \( (rt) \)!
  - Need to have a copy elsewhere if you plan on repeating on failure or using value later
A related problem: Deadlock

- Consider the following: A dozen lawyers are sitting around a table for dinner
- Between each lawyer is a chopstick
  - Original version is ‘Dining Philosophers’ by Dijkstra, but changing to lawyers is a Berkeley innovation…
- Each lawyer grabs the chopstick to the right and then to the left…
- What if every lawyer only grabs the first chopstick?
- Result is **deadlock**: each lawyer is waiting on another to release a chopstick
Solutions for Deadlock…

- Structure your program so it doesn’t occur!
  - EG, rather than going “Right then left” go “even than odd”
    - Now the system will always progress

- Or have each lawyer give up after a \textit{random} time, drop the chopstick, \textit{randomly} wait and try again
  - Need randomization to prevent “livelock”: technique used by Ethernet to arbitrate access

- Centrally arbitrate access
  - A waiter tells each lawyer which chopstick to take: may limit potential parallelism

- Watch for deadlock and respond
  - A waiter is standing by to shoot a lawyer if deadlock occurs…
    - Which is why a long-forgotten Berkeley OS prof changed it to “lawyers”
“Shooting Laywers” For Fun and Profit

• The notion of “detect that things stopped working, and just kill either a thread or the whole thing” is amazingly effective
  • I do it all the time for long-running things
• Its great: Its easy to get things 99.9% right…
  • But that last .1% is a F@#)(*# pain.
• So instead of hunting down the last bugs that only crop up after 8 (or 80, or 800) hours of running...
  • Just write a little monitor program and just kill & restart things!
Announcements

• Review Session: Tomorrow (4/4)
  • 7-9 PM, Hearst Field Annex A1

• Midterm 2: Thursday (4/6)
  • 7-9 PM, see Piazza announcements for room assignments
OpenMP Programming Model - Review

• Fork - Join Model:

• OpenMP programs begin as single process (master thread) and executes sequentially until the first parallel region construct is encountered
  • FORK: 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 threads
  • JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
parallel Pragma and Scope - Review

- Basic OpenMP construct for parallelization:
  ```c
  #pragma omp parallel
  { /* code goes here
      Brackets needed because the
      pragma applies to a single C statement,
      and needs to start on the line after #pragma */
  }
  ```
  - Each thread runs a copy of code within the block
  - Thread scheduling is non-deterministic

- OpenMP default is `shared` variables
  - To make private, need to declare with pragma:
    ```c
    #pragma omp parallel private (x)
    ```
Example: Calculating $\pi$

Mathematically, we know that:

$$\int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi$$

Where each rectangle has width $\Delta x$ and height $F(x_i)$ at the middle of interval $i$. 

Numerical Integration
Sequential Calculation of $\pi$ in C

```c
#include <stdio.h>      /* Serial Code */
static long num_steps = 100000;
double step;
void main () {
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double)num_steps;
    for (i = 1; i <= num_steps; i++) {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0 + x*x);
    }
    pi = sum / num_steps;
    printf ("pi = %6.12f\n", pi);
}
```

\[
\int_0^1 \frac{4.0}{1+x^2} \, dx = \pi
\]

\[
\sum_{i=0}^{N} F(x_i) \Delta x \approx \pi
\]
#include <omp.h>
static long num_steps = 100000; double step;

void main () {
    int i, threads;   double  x, pi, *sum;
    threads = omp_get_max_threads()
    sum = malloc(sizeof(double) * threads)
    step = 1.0/(double) num_steps;
    #pragma omp parallel private ( i, x )
    {
        int id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+threads)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);  free(sum)
}
OpenMP Directives (Work-Sharing)

- These are defined within a parallel section

  - **FORK**
  - **DO/for loop**
  - **JOIN**

  **master thread**

  Shares iterations of a loop across the threads

  - **FORK**
  - **SECTIONS**
  - **JOIN**

  **master thread**

  Each section is executed by a separate thread

  - **FORK**
  - **SINGLE**
  - **JOIN**

  **master thread**

  Serializes the execution of a thread, so why bother?
Parallel Statement Shorthand

```c
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < len; i++) { ... }
}
```

can be shortened to:

```c
#pragma omp parallel for
for (i = 0; i < len; i++) { ... }
```

- Also works for sections

This is the only directive in the parallel section
Building Block: `for` loop

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

- Breaks `for` loop into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 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

In general, don’t jump outside of any pragma block
Parallel for pragma

- `#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 “barrier” 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?
OpenMP Timing

• Elapsed wall clock time:
  ```c
  double omp_get_wtime(void);
  ```

• Returns elapsed wall clock time in seconds
• Time is measured per thread, **no guarantee** can be made that two distinct threads measure the same time
• Time is measured from “some time in the past,” so subtract results of two calls to `omp_get_wtime` to get elapsed time
Matrix Multiply in OpenMP

```c
start_time = omp_get_wtime();
#pragma omp parallel for private(tmp, j, k)
    for (i=0; i<M; i++){
        for (j=0; j<N; j++){n
            tmp = 0.0;
            for( k=0; k<P; k++){n
                /* C(i,j) = sum(over k) A(i,k) * B(k,j)*/n
                tmp += A[i][k] * B[k][j];
            }
            C[i][j] = tmp;
        }
    }
run_time = omp_get_wtime() - start_time;
```

Outer loop spread across N threads; inner loops inside a single thread.
Notes on Matrix Multiply Example

• More performance optimizations available:
  • Higher compiler optimization (-O2, -O3) to reduce number of instructions executed
    • Why should you optimize it when the compiler can do it for you…
  • Cache blocking to improve memory performance
    • Take advantage of both spatial and temporal locality
  • Using SIMD SSE instructions to raise floating point computation rate (Data Level Parallelism/SIMD parallelism)
    • Or hell, just call somebody’s existing CUDA-based library function…
OpenMP Reduction

- double avg, sum=0.0, A[MAX]; int i;
  #pragma omp parallel for private (sum)
  for (i = 0; i <= MAX; i++)
    sum += A[i];
  avg = sum/MAX;  // bug

- Reduction: specifies that 1 or more variables that are private to each thread are subject of reduction operation at end of parallel region:
  reduction(operation:var) where
  - Operation: operator to perform on the variables (var) at the end of the parallel region
  - Var: One or more variables on which to perform scalar reduction.

- double avg, sum=0.0, A[MAX]; int i;
  #pragma omp for reduction(+ : sum)
  for (i = 0; i <= MAX; i++)
    sum += A[i];
  avg = sum/MAX;
Calculating π Version (1) - review

```c
#include <omp.h>
#define NUM_THREADS 4
static long num_steps = 100000; double step;

void main () {
    int i;    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel private ( i, x )
    {
        int id = omp_get_thread_num();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=1; i<NUM_THREADS; i++)
        sum[0] += sum[i];  pi = sum[0] / num_steps
    printf ("pi = %6.12f\n", pi);
}
```
Version 2: parallel for, reduction

```c
#include <omp.h>
#include <stdio.h>

static long num_steps = 100000;

double step;

void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=1; i<= num_steps; i++) {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = sum / num_steps;
    printf ("pi = %6.8f\n", pi);
}
```
Simple Multi-core Processor

[Diagram showing two processors (Processor 0 and Processor 1) with control, datapath (PC, Registers, ALU), memory, and input/output interfaces.]

Processor 0
- Control
- Datapath
  - PC
  - Registers
  - ALU
- Memory
- Input

Processor 1
- Control
- Datapath
  - PC
  - Registers
  - ALU
- Memory
- Output

Processor 0 Memory Accesses
Processor 1 Memory Accesses

I/O-Memory Interfaces
Multiprocessor Caches

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory
Shared Memory and Caches

• What if?
  • Processors 1 and 2 read Memory[1000] (value 20)
Shared Memory and Caches

- Now:
  - Processor 0 writes Memory[1000] with 40

Problem?
Keeping Multiple Caches Coherent

• Architect’s job: shared memory
  => keep cache values coherent

• Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  • If only reading, many processors can have copies
  • If a processor writes, invalidate any other copies

• Write transactions from one processor, other caches “snoop” the common interconnect checking for tags they hold
  • Invalidate any copies of same address modified in other cache
Shared Memory and Caches

• Example, now with cache coherence
  • Processors 1 and 2 read Memory[1000]
  • Processor 0 writes Memory[1000] with 40
Clickers/Peer Instruction: Which statement is true?

- A: Using write-through caches removes the need for cache coherence
- B: Every processor store instruction must check contents of other caches
- C: Most processor load and store accesses only need to check in local private cache
- D: Only one processor can cache any memory location at one time
Cache Coherency Tracked by Block

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
Coherency Tracked by Cache Block

- Block ping-pongs between two caches even though processors are accessing disjoint variables
- Effect called *false sharing*
- How can you prevent it?
  - One reason why OpenMP breaks up loops the way it does
Review: Understanding Cache Misses: The 3Cs

- **Compulsory** (cold start or process migration, 1st reference):
  - First access to block, impossible to avoid; small effect for long-running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

- **Capacity** (not compulsory and…)
  - Cache cannot contain all blocks accessed by the program even with perfect replacement policy in fully associative cache
  - Solution: increase cache size (may increase access time)

- **Conflict** (not compulsory or capacity and…):
  - Multiple memory locations map to the same cache location: Wouldn’t be a miss if the cache was fully associative
  - Solution: increase cache size, associativity or replacement policy
Fourth “C” of Cache Misses: 
Coherence Misses

- Misses caused by coherence traffic with other processor
  - Also known as communication misses because represents data moving between processors working together on a parallel program
- For some parallel programs, coherence misses can dominate total misses
  - It gets even more complicated with multithreaded processors: You want separate threads on the same CPU to have common working set, otherwise you get what could be described as incoherence misses
And in Conclusion, ...

- Multiprocessor/Multicore uses Shared Memory
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!
- OpenMP as simple parallel extension to C
  - Threads, Parallel for, private, reductions ...
  - ≈ C: small so easy to learn, but not very high level and it’s easy to get into trouble
  - Much we didn’t cover – including other synchronization mechanisms (locks, etc.)