Peer Instruction: Synchronization

• Consider the following code when executed concurrently by two threads:

```
lw $t0,0($s0)
addi $t0,$t0,1
sw $t0,0($s0)
```

(Mem[$s0] contains value 100)

• What possible values can result in Mem[$s0]?

Peer Instruction: Synchronization, Answer

• Consider the following code when executed concurrently by two threads:

```
lw $t0,0($s0)
addi $t0,$t0,1
sw $t0,0($s0)
```

(Mem[$s0] contains value 100)

• What possible values can result in Mem[$s0]?

- 102 if one thread starts executing after the other completely finishes.
- 101 if both threads execute the lw before either thread executes the sw. One thread will see “stale data”

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: the lock
• Processors read lock to see if 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

Synchronization in MIPS

• Load linked: 11 rt, offset(rs)
• Store conditional: sc rt, offset(rs)
  – Succeeds if location not changed since the lw
  – 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: $s4 ↔ ($s1)
```
try: add $t0,$zero,$s4 ;copy exchange value
ll $t1,0($s1)    ;load linked
sc $t0,0($s1)    ;store conditional
beq $t0,$zero,try ;branch store fails
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

- Example: MIPS sequence for implementing a T&S at ($s1)
  
  Try:
  
  addiu $t0, $zero, 1
  ll $t1, 0($s1)
  bne $t1, $zero, Try
  sc $t0, 0($s1)
  beq $t0, $zero, try

  Locked:
  
  critical section
  sw $zero, 0($s1)

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Break
- Common Pitfalls
- Summary

What is OpenMP?

- API used for multi-threaded, shared memory parallelism
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
- Standardized
- See http://www.openmp.org/
  http://computing.llnl.gov/tutorials/openMP/
Shared Memory Model with Explicit Thread-based Parallelism

- Multiple threads in a shared memory environment, 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:
  - Code can only be run in shared memory environments!
  - Compiler must support OpenMP (e.g., gcc 4.2)

OpenMP Built On C’s “pragma” Mechanism

- Pragmas are a preprocessor mechanism that C provides for language extensions
- Many different uses: structure packing, symbol aliasing, floating point exception modes, ...
- Example:

```
#pragma omp parallel for
```

- Good for OpenMP because compilers that don’t recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

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

- #pragma omp parallel for

- Shares iterations of a loop across the team

- Each section executed by a separate thread

- Serializes the execution of a thread

Example: C for loop

```
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 0-99 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: Parallel for pragma

```
#pragma omp parallel for
```

- 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” Statement and Scope

- The general parallel construct: Each thread executes a copy of the code within the block below.
  ```
  #pragma omp parallel
  { #pragma omp for //This is the only directive in the parallel section
    for(i=0;i<len;i++) { … } }
  ```
- OpenMP default is shared variables
  - To make private, need to declare with pragma
  ```
  #pragma omp parallel private (x)
  ```

Parallel Statement Shorthand

- Can be shortened to
  ```
  #pragma omp parallel for
  for(i=0;i<len;i++) { … } 
  ```

Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Break
- Common Pitfalls
- Summary

Administrivia

- Project #2: Matrix Multiply Performance Improvement
  - Work in groups of two!
  - Part 1: Due July 24 (this Sunday)
  - Part 2: Due July 31
- HW #3 also due July 27
- Closely packed due dates, try to get ahead of schedule for the project.
Intel reveals new 50 core Knight’s Corner co-processor, to compete with Nvidia’s multi-hundred core “general purpose” Tesla GPU.

“The main advantage that Intel touts vs. Tesla is that because MIC is just a bunch of x86 cores, it’s easy for users to port their existing toolchains to it.”


---

**Agenda**

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Break
- Common Pitfalls
- Summary

---

**OpenMP Examples**

- Hello World
  - **OMP Get Environment (hidden slides)**
- For Loop Workshare
- Section Workshare
- Sequential and Parallel Numerical Calculation of Pi

---

**Hello World in OpenMP**

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

int main (void) {
  int nthreads, tid;
  /* Fork team of threads with each 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 */
  }
}
```

---

**Get Environment Information in OpenMP**

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

int main (int argc, char *argv[])
{
  int nthreads, tid, proc, mast, inpar, dynamic, nested;
  #pragma omp parallel private(nthreads, tid)
  {
    tid = omp_get_thread_num(); /* Obtain thread number */
    if (tid == 0) { /* Only master thread does this */
      printf("Thread %d getting environment info...\n", tid);
      /* Get environment information */
      proc = omp_get_num_procs();
      nthreads = omp_get_num_threads();
      mast = omp_get_max_threads();
      inpar = omp_in_parallel();
      dynamic = omp_get_dynamic();
      nested = omp_get_nested();
    }
  }
}
```
Get Environment Information in OpenMP

/* Print environment information */
print("Number of processors = \$\{procs\}\$", proc);
print("Max threads = \$\{t\}\$", t);
print("Dynamic threads enabled? = \$\{d\}\$\$, dynamic");
print("Nested parallelism enabled? = \$\{n\}\$\$, nested");
}  /* Parallel Done */

Workshare for loop Example #1 in OpenMP

#include <omp.h>
#include <stdio.h>
define CHUNKSIZE 10
#define N 100
int main (int argc, char *argv[])
{ int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = c[i] = 0.0;
  chunk = CHUNKSIZE;

  #pragma omp parallel shared(a,b,c,nthreads,chunk)
  private(tid, i)
  
  { nthreads = omp_get_num_threads();
    #pragma omp for schedule(dynamic,chunk)
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }
  /* end of parallel section */

Workshare sections Example #2 in OpenMP

#include <omp.h>
#include <stdio.h>
define N 50
int main (int argc, char *argv[])
{ int i, nthreads, tid;
  float a[N], b[N], c[N], d[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = i * 1.5;
    b[i] = i + 22.35;
    c[i] = d[i] = 0.0;
    }
Workshare sections Example #2 in OpenMP

```c
#include <omp.h>

#pragma omp parallel private(i,tid)
{  tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }
    printf("Thread %d starting...\n",tid);
}

#pragma omp section
{  printf("Thread %d doing section 1\n",tid);
    for (i=0; i<NUM_THREADS; i++)
        c[i] = a[i] + b[i];
    printf("Thread %d: c[\%d] = \%f\n",tid,i,c[i]);
}

#pragma omp section
{  printf("Thread %d doing section 2\n",tid);
    for (i=0; i<NUM_THREADS; i++)
        d[i] = a[i] * b[i];
    printf("Thread %d: d[\%d] = \%f\n",tid,i,d[i]);
}

#pragma omp section
{  printf("Thread %d done\n",tid);
}
```

Workshare sections Example #2 in OpenMP

```
#define NUM_THREADS 2

int main (int argc, char *argv[]){
    int i, id;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    for (i=0; i < num_steps; i++)
        x = (i + 0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    pi = sum/num_steps;
    printf("pi = \%f\n", pi);
}
```

Calculating π using Numerical Integration

```
π = \int_0^1 \frac{4}{1 + x^2} \, dx
```

Riemann Sum Approximation:

\[
\int_a^b f(x) \, dx = \lim_{n \to \infty} \sum_{i=1}^n f(x_i^*) \Delta x_i
\]

P is a partition of [a, b] into n subintervals, \([x_0, x_1], [x_1, x_2], \ldots, [x_{n-1}, x_n]\) each of width \(x^*\) in the interval \([x_i, x_{i+1}]\) for each \(i\).

As ||P|| \to 0, n \to \infty and x \to 0.

\(x^*\) to be the midpoint of each subinterval, i.e. \(x^* = \frac{1}{2}(x_i + x_{i+1})\), for each \(i\) from 1 to \(n\).

Sequential Calculation of π in C

```
#include <stdio.h>

/* Serial Code */
static long num_steps = 100000;
double step;
int main (int argc, char *argv[]){
    int i, id;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    for (i=0; i < num_steps; i++)
        x = (i + 0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    pi = sum/num_steps;
    printf("\pi = \%f\n", pi);
}
```

OpenMP Version #1

```
#include <omp.h>

static long num_steps = 100000; double step;
#define NUM_THREADS 2

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

```c
#include <omp.h>
static long num_steps = 100000; double step;
define NUM_THREADS 2
main (int argc, char *argv[]){
    int i, id; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    #pragma omp parallel private(x)
    {
        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 = 0, pi = 0.0; i < NUM_THREADS; i++)
        pi += sum[i];
    printf("pi = %6.12f, pi /\n", pi/num_steps);
}
```

OpenMP Critical Section

```c
#include <omp.h>
int main(void)
{
    int x;
    x = 0;
    #pragma omp parallel
    {
        #pragma omp critical
        x = x + 1;
    }
}
```

OpenMP Version #2

```c
#include <omp.h>
static long num_steps = 100000; double step;
define NUM_THREADS 2
main (int argc, char *argv[]){
    int i, id; double x, sum, pi=0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel private (x, sum) 
    {
        id = omp_get_thread_num();
        for (i = id, sum = 0.0; i < num_steps; i = i + NUM_THREADS) {
            x = (i + 0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    
    #pragma omp critical
    pi += sum;
    
    printf("pi = %6.12f, pi/\n", pi/num_steps);
}
```

OpenMP Reduction

- **Reduction**: specifies that one 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

```c
#pragma omp for reduction(+:nSum) 
for (i = START ; i <= END; i++)
    nSum += i;
```
# OpenMP Version #3

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

//static long num_steps = 100000; double step;
int main (int argc, char *argv[]){
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i = 0; i < num_steps; i++){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = sum / num_steps;
    printf("pi = %6.8f
", pi);
}
```

Note: Don't have to declare for loop index variable i private, since that is default

---

# OpenMP Timing

- **omp_get_wtime** – Elapsed wall clock time
double omp_get_wtime(void);

#include <omp.h> // to get function

- 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". On POSIX compliant systems the seconds since the Epoch (00:00:00 UTC, January 1, 1970) are returned.

---

# Agenda

- OpenMP Introduction
- Administrivia
- OpenMP Examples
- Break
- Common Pitfalls
- Summary

---

# OpenMP Version #4

```c
#include <omp.h>
static long num_steps = 100000; double step;
int main (int argc, char *argv[]){
    int i, id; double x, pi, sum;
    sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel {
        #pragma omp for
        for (i = 0; i < num_steps; i++){
            x = (i + 0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    printf("pi = %6.12f\n", step * sum);
}
```

---

# OpenMP Version #4 (with bug)

```c
pi = 3.14159265 in 0.176 seconds  pi = 3.14159265 in 0.177 seconds
pi = 3.14159265 in 0.190 seconds  pi = 3.14159265 in 0.178 seconds
pi = 3.14159265 in 0.178 seconds  pi = 3.14159265 in 0.182 seconds
pi = 3.14159265 in 0.182 seconds  pi = 3.14159265 in 0.176 seconds
pi = 3.14159265 in 0.182 seconds  pi = 3.14159265 in 0.181 seconds
pi = 3.14159265 in 0.182 seconds  pi = 3.14159265 in 0.184 seconds
pi = 3.14159265 in 0.180 seconds  pi = 3.14159265 in 0.178 seconds
pi = 3.14159265 in 0.190 seconds  pi = 3.14159265 in 0.180 seconds
pi = 1.265327037088 pi = 1.961754300795
pi = 1.702630397118 pi = 1.961754300795
pi = 3.113547769550 pi = 1.961754300795
pi = 2.491334675594 pi = 1.961754300795
pi = 1.476765567928 pi = 1.961754300795
pi = 1.307626664756 pi = 1.961754300795
pi = 1.307626664756 pi = 1.961754300795
pi = 3.141592653598 pi = 1.961754300795
pi = 2.487932948250 pi = 1.961754300795
pi = 1.514950670570 pi = 1.961754300795
```

---

7/19/2011 Summer 2011 -- Lecture #17 49

7/19/2011 Summer 2011 -- Lecture #17 50

7/19/2011 Summer 2011 -- Lecture #17 51

7/19/2011 Summer 2011 -- Lecture #17 52

7/19/2011 Summer 2011 -- Lecture #17 53

7/19/2011 Summer 2011 -- Lecture #17 54

7/19/2011 Summer 2011 -- Lecture #17 55

7/19/2011 Summer 2011 -- Lecture #17 56
OpenMP Version #5
#include <omp.h>
static long num_steps = 100000; double step;
int main (int argc, char *argv[]){
    int i, id; double x, pi, sum, sum0;
    sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel private(x,sum0) shared (step,sum)
    {
        sum0 = 0.0;
        #pragma omp for
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5)*step;
            sum0 += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        {
            sum = sum + sum0;
        }
    }
    printf("pi = %6.12f
", step * sum);
}

Matrix Multiply in OpenMP
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*Mdim+j)) = tmp;
    }
}
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

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
Open MP Pitfall #2: Avoiding Sharing Issues 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 += 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 += a[i];
    }
  }
  ```

OpenMP Pitfall #4: 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!

```
c(
m, k) = \sum_{i=0}^{Ndim} \sum_{j=0}^{Mdim} A(i, k) * B(k, j) * C(m, j)
```

```c
for (i=0; i<Ndim; i++)
for (j=0; j<Mdim; j++)
for (k=0; k<Pdim; k++)
{ 
  C(m, j) = \sum_{i=0}^{Ndim} \sum_{k=0}^{Pdim} A(i, k) * B(k, j) * C(m, j) + 
  \sum_{i=0}^{Ndim} \sum_{k=0}^{Pdim} A(i, k) * B(k, j) * C(m, j);
}
```

```
run_time = omp_get_wtime() - start_time;
```

Way too much overhead in thread generation to have this statement run this frequently.

And in Conclusion, ...

- 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, ...