Thread Level Parallelism: OpenMP

Instructor:
Michael Greenbaum
You Are Here!

- **Software**
  - **Parallel Requests**
    - Assigned to computer
    - e.g., Search “Katz”
  - **Parallel Threads**
    - Assigned to core
    - e.g., Lookup, Ads
  - **Parallel Instructions**
    - >1 instruction @ one time
    - e.g., 5 pipelined instructions
  - **Parallel Data**
    - >1 data item @ one time
    - e.g., Add of 4 pairs of words
  - **Hardware descriptions**
    - All gates functioning in parallel at same time

- **Hardware**
  - **Warehouse Scale Computer**
  - **Harness Parallelism & Achieve High Performance**
  - **Core**
  - **Memory** (Cache)
  - **Instruction Unit(s)**
    - $A_0 + B_0$, $A_1 + B_1$, $A_2 + B_2$, $A_3 + B_3$
  - **Input/Output**
  - **Functional Unit(s)**
  - **Main Memory**
  - **Logic Gates**

7/19/2011

Summer 2011 -- Lecture #17
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)

<table>
<thead>
<tr>
<th>Color</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>100, 101, or 102</td>
</tr>
<tr>
<td>Blue</td>
<td>101 or 102</td>
</tr>
<tr>
<td>Yellow</td>
<td>100 or 102</td>
</tr>
<tr>
<td>Red</td>
<td>100 or 101</td>
</tr>
<tr>
<td>Purple</td>
<td>102</td>
</tr>
</tbody>
</table>

• 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]?

<table>
<thead>
<tr>
<th>Color</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>100, 101, or 102</td>
</tr>
<tr>
<td>Blue</td>
<td>101 or 102</td>
</tr>
<tr>
<td>Yellow</td>
<td>100 or 102</td>
</tr>
<tr>
<td>Red</td>
<td>100 or 101</td>
</tr>
<tr>
<td>Purple</td>
<td>102</td>
</tr>
</tbody>
</table>

- 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: \( \text{ll } rt,\text{offset}(rs) \)
• Store conditional: \( \text{sc } rt,\text{offset}(rs) \)
  – Succeeds if location not changed since the \( \text{ll} \)
    • 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: \( $s4 \leftrightarrow ($s1) \)

  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
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

OpenMP Specification

OpenMP language extensions

parallel control structures
- governs flow of control in the program
- parallel directive

work sharing
- distributes work among threads
- do/parallel do and section directives

data environment
- scopes variables
- shared and private clauses

synchronization
- coordinates thread execution
- critical and atomic directives
- barrier directive

runtime functions, env. variables
- runtime environment
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
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

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 0-49 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: 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 of threads:
  `export OMP_NUM_THREADS=x`
- Shell command check number of threads:
  `echo $OMP_NUM_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.

```c
#pragma omp parallel
{
    ID = omp_get_thread_num();
    foo(ID);
}
```

• OpenMP default is shared variables
  – To make private, need to declare with pragma

```c
#pragma omp parallel private (x)
```
Parallel Statement Shorthand

```c
#pragma omp parallel
{
  #pragma omp for  //This is the only
      //directive in the
      // parallel section
    for(i=0;i<len;i++) { ... }
}
```
can be shortened to
```c
#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.
cs61c in the News

• 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

#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 */
}
Hello World in OpenMP

localhost:OpenMP randykatz$ ./omp_hello
Hello World from thread = 0
Hello World from thread = 1
Number of threads = 2
Workshare for loop Example #1 in OpenMP

#include <omp.h>
#include <stdio.h>
#include <stdlib.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] = i * 1.0;
  chunk = CHUNKSIZE;
Workshare for loop Example #1 in OpenMP

```c
#pragma omp parallel shared(a,b,c,nthreads,chunk) 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 for schedule(dynamic,chunk)
    for (i=0; i<N; i++){
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }
} /* end of parallel section */
```

The iterations of a loop are scheduled dynamically across the team of threads. A thread will perform CHUNK iterations at a time before being scheduled for the next CHUNK of work.
Number of threads = 2
Thread 0 starting...
Thread 1 starting...
Thread 0: c[0]= 0.000000
Thread 1: c[10]= 20.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 1: c[12]= 24.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 1: c[14]= 28.000000
Thread 0: c[5]= 10.000000
Thread 1: c[15]= 30.000000
Thread 0: c[6]= 12.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 0: c[7]= 14.000000
Thread 1: c[19]= 38.000000
Thread 0: c[8]= 16.000000
Thread 1: c[20]= 40.000000
Thread 0: c[9]= 18.000000
Thread 1: c[21]= 42.000000
Thread 0: c[30]= 60.000000
Thread 1: c[22]= 44.000000
Thread 0: c[31]= 62.000000
Thread 1: c[23]= 46.000000
Thread 0: c[78]= 156.000000
Thread 1: c[25]= 50.000000
Thread 0: c[79]= 158.000000
Thread 1: c[26]= 52.000000
Thread 0: c[80]= 160.000000
Thread 1: c[27]= 54.000000
Thread 0: c[81]= 162.000000
Thread 1: c[28]= 56.000000
Thread 0: c[82]= 164.000000
Thread 1: c[29]= 58.000000
Thread 0: c[83]= 166.000000
Thread 1: c[30]= 60.000000
Thread 0: c[84]= 168.000000
Thread 1: c[31]= 62.000000
Thread 0: c[85]= 170.000000
Thread 1: c[32]= 64.000000
Thread 0: c[86]= 172.000000
Thread 1: c[33]= 66.000000
Thread 0: c[87]= 174.000000
Thread 1: c[34]= 68.000000
Thread 0: c[88]= 176.000000
Thread 1: c[35]= 70.000000
Thread 0: c[89]= 178.000000
Thread 1: c[36]= 72.000000
Thread 0: c[90]= 180.000000
Thread 1: c[37]= 74.000000
Thread 0: c[91]= 182.000000
Thread 1: c[38]= 76.000000
Thread 0: c[92]= 184.000000
Thread 1: c[39]= 78.000000
Thread 0: c[93]= 186.000000
Thread 1: c[40]= 80.000000
Thread 0: c[94]= 188.000000
Thread 1: c[41]= 82.000000
Thread 0: c[95]= 190.000000
Thread 1: c[42]= 84.000000
Thread 0: c[96]= 192.000000
Thread 1: c[43]= 86.000000
Thread 0: c[97]= 194.000000
Thread 1: c[44]= 88.000000
Thread 0: c[98]= 196.000000
Thread 1: c[45]= 90.000000
Thread 0: c[99]= 198.000000
Thread 1: c[46]= 92.000000
Workshare *sections* Example #2 in OpenMP

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.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;
    }
}
```
#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 sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++){
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }
    }
}
Workshare sections Example #2 in OpenMP

```c
#pragma omp section
{ printf("Thread %d doing section 2\n",tid);
  for (i=0; i<N; i++){
    d[i] = a[i] * b[i];
    printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
  }
}
/* end of sections */
printf("Thread %d done.\n",tid);
} /* end of parallel section */
```
Number of threads = 2
Thread 1 starting...
Thread 0 starting...
Thread 1 doing section 1
Thread 0 doing section 2
Thread 1: c[0]= 22.350000
Thread 0: d[0]= 0.000000
Thread 1: c[1]= 24.850000
Thread 0: d[1]= 35.025002
Thread 1: c[2]= 27.350000
Thread 0: d[2]= 73.050003
Thread 0: d[3]= 114.075005
Thread 1: c[4]= 32.349998
Thread 0: d[4]= 158.100006
Thread 1: c[5]= 34.849998
Thread 0: d[5]= 205.125000
Thread 1: c[6]= 37.349998
Thread 0: d[6]= 255.150009
Thread 0: d[7]= 308.175018
Thread 1: c[8]= 42.3225006
Thread 0: d[8]= 364.200012
Thread 0: d[9]= 423.225006
Thread 0: d[10]= 485.249969
Thread 0: d[12]= 618.299988
Thread 0: d[13]= 689.324951
Thread 0: d[14]= 763.349976
Thread 0: d[15]= 840.374939
Thread 1: c[33]= 104.849998
Thread 0: d[41]= 3896.024902
Thread 1: c[34]= 107.349998
Thread 0: d[42]= 4054.049805
Thread 1: c[35]= 109.849998
Thread 0: d[43]= 4215.074707
Thread 1: c[36]= 112.349998
Thread 0: d[44]= 4379.100098
Thread 1: c[37]= 114.849998
Thread 1: c[38]= 117.349998
Thread 1: c[39]= 119.849998
Thread 1: c[40]= 122.349998
Thread 1: c[41]= 124.849998
Thread 1: c[42]= 127.349998
Thread 1: c[43]= 129.850006
Thread 1: c[44]= 132.350006
Thread 0: d[45]= 4546.125000
Thread 1: c[45]= 134.850006
Thread 1: c[46]= 137.350006
Thread 0: d[47]= 4889.174805
Thread 1: c[47]= 139.850006
Thread 0: d[48]= 5065.199707
Thread 0: d[49]= 5244.225098
Thread 0 done.
Thread 1: c[48]= 142.350006
Thread 1: c[49]= 144.850006
Thread 1 done.

Nowait in sections
Calculating $\pi$ using Numerical Integration

$$\pi = \int_0^1 \frac{4}{1 + x^2} \, dx$$

Riemann Sum Approximation:

$$\int_a^b f(x) \, dx = \lim_{||P|| \to 0} \sum_{i=1}^{n} f(x_i^*) \Delta x_i$$

$P$ 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 $x_i$. $x_i^*$ is in the interval $[x_{i-1}, x_i]$ for each $i$. As $||P|| \to 0$, $n \to \infty$ and $x \to 0$.

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

http://www.youtube.com/watch?v=H20cKjz-bjw
Sequential Calculation of $\pi$ in C

```c
#include <stdio.h>
/* Serial Code */
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;
    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 = %.12f\n", pi);
}
```

pi = 3.14159265
#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 = 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\n", pi / num_steps);
}
OpenMP Version #1 (with bug)

pi = 2.187926626214
pi = 3.062113243183
pi = 2.093066397959
pi = 3.141592653598
pi = 2.199725334399
pi = 2.129254580053
pi = 3.298710582772
pi = 2.358075931672
pi = 1.979735213760
pi = 3.134406923694
pi = 3.141592653598
pi = 2.273284475646

pi = 3.141592653598
pi = 1.999849495043
pi = 2.348751552706
pi = 1.858418846828
pi = 3.141592653598
pi = 3.141592653598
pi = 2.425973292566
pi = 3.141592653598
pi = 2.024963879529
pi = 2.357897349032
pi = 2.325699828349
pi = 1.825693174923
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 = 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\n", pi / num_steps);
}
OpenMP Critical Section

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

Only one thread executes the following code block at a time

Compiler generates necessary lock/unlock code around the increment of x
OpenMP Version #2

```c
#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, 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 += NUM_THREADS){
            x = (i + 0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum;
    }
    printf ("pi = %6.12f,\n", pi/num_steps);
}
```
OpenMP Version #2 (with bug)

\[
\begin{align*}
\pi &= 2.193065170635 & \pi &= 3.141592653598 \\
\pi &= 2.094493774172 & \pi &= 2.446564596856 \\
\pi &= 2.941181318377 & \pi &= 2.292319872127 \\
\pi &= 3.706889160821 & \pi &= 3.430000515865 \\
\pi &= 3.146809442180 & \pi &= 3.650290478614 \\
\pi &= 2.416534126971 & \pi &= 1.988308636488 \\
\pi &= 3.141592653598 & \pi &= 3.141592653598 \\
\pi &= 2.880126390443 & \pi &= 2.516381208613 \\
\pi &= 2.012071161080 & \pi &= 3.141592653598 \\
\pi &= 2.294675684941 & \pi &= 2.331351828709 \\
\pi &= 2.769967988388 & \pi &= 2.245872892167 \\
\pi &= 2.373218039482 & \pi &= 1.928775146184 
\end{align*}
\]
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:
  
  ```c
  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;
  ```
```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 = sum + 4.0/(1.0+x*x);
    }
    pi = sum / num_steps;
    printf ("pi = %6.8f\n", 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
  ```c
  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.
OpenMP Version #3

\[ \pi = 3.14159265 \text{ in 0.176 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.190 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.178 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.177 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.182 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.181 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.177 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.178 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.177 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.179 seconds} \]
\[ \pi = 3.14159265 \text{ in 0.180 seconds} \]
Agenda

• OpenMP Introduction
• Administrivia
• OpenMP Examples
• Break
• Common Pitfalls
• Summary
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;
        }
    }
run_time = omp_get_wtime() - start_time;
```

Note: Outer loop spread across N threads; inner loops inside a thread
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:
  ```c
  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:
  
  ```
  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:
  
  ```
  #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 #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!
OpenMP Pitfall #4: Parallel Overhead

```c
start_time = omp_get_wtime();
for (i=0; i<Ndim; i++){
    for (j=0; j<Mdim; j++){
        tmp = 0.0;
        #pragma omp parallel for reduction(+:tmp)
        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;
    }
}
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, ...

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