Intro to OpenMP

http://www.oit.duke.edu/scsc
http://wiki.duke.edu/display/SCSC
scsc@duke.edu

John Pormann, Ph.D.
jbp1@duke.edu

Schedule

Wednesday

1:00
Intro to CPU Arch.

2:00
Intro to OpenMP

3:00
LAB

4:00
Intro to Parallelism

5:00
Outline

CPU Architecture
- "Shared Memory" vs. "Distributed Memory"

What is OpenMP?
- Operational Model for OpenMP

OpenMP Programming
- Loop Parallelism
- Private vs. Shared Variables
- Critical Regions
- Parallel Sections

Performance Issues/Hints

Basic CPU Architecture

![Diagram of CPU Architecture](attachment:image.png)
Outline

CPU Architecture
  ◆ “Shared Memory” vs. “Distributed Memory”

What is OpenMP?
  ◆ Operational Model for OpenMP

OpenMP Programming
  ◆ Loop Parallelism
  ◆ Private vs. Shared Variables
  ◆ Critical Regions
  ◆ Parallel Sections

Performance Issues/Hints

What is OpenMP?

◆ OpenMP is an industry-standard combination of compiler directives and library routines, for shared-memory computers, that allow programmers to specify parallelism in their code without getting too caught up in the details of parallel programming

◆ OpenMP is a response to increasing hardware-parallelism in most (all) HPC platforms
  ◆ Sort of a “bolt-on” to the language, not a new language unto itself

  ◆ In Fortran, it hides many parallel constructs within comment lines (so your code still compiles/runs just fine on 1 CPU)
  ◆ In C, it uses “pragmas” (older compilers will just ignore the parallel constructs)
“Hello World” Program in OpenMP

```c
program hello
  c$omp parallel
   print *, 'Hello world!'
   print *, omp_get_thread_num()
  c$omp end parallel
  print *, 'Good-bye!'
end

int main() {
  printf("Hello World!\n");
  #pragma omp parallel
   printf("%i\n",omp_get_thread_num());
   printf("Good-bye!\n");
  return( 0 );
}
```

Possible Outputs:

```
Hello world!
0
1
2
3
Good-bye!
```

Compiling OpenMP Programs

- OpenMP *REQUIRES* an OpenMP-compliant compiler
  - gcc 4.0 has some support
    - gcc 2.x, 3.x can NOT handle OpenMP!
  - Generally, you’ll need a “professional” compiler suite
    - Portland Group, Pathscale, Intel, Sun

- All you have to do is “turn on” the OpenMP support:
  - There is also a library that is required, but the compiler generally knows to include this automatically

```
% gcc -fopenmp
% cc -xopenmp=parallel
% icc -openmp
% pgcc -mp
% xlc -qomp=omp
```

Note that some compilers require ‘#pragma’ to appear in the first column of the line
Operational Model for OpenMP

```c
print *, 'Hello world!'
c$omp parallel
print *, omp_get_thread_num()
c$omp end parallel
print *, 'Good-bye!'
```

Master Thread performs the serial work

Master Thread encounters 'parallel region'
- creates N-1 additional Threads

All Threads (including Master) perform the parallel work

Master Thread waits for all Threads to finish
- implied Barrier at end of every parallel region

Every Parallel Region has a "cost"
(i.e. can slow down your code)

What is a Thread?

- A “Thread” is an operating system construct that contains all of the
  CPU-state associated with a piece of the parallel program
  - A set of floating-point and integer registers, a set of memory
    pointers, a stack pointer, a program counter, etc.
    - Sort of a “Virtual CPU” which competes for access to a real CPU in
      order to do real work
  - Generally, you want to use N Threads on a N CPU machine
    - But you can use any number of Threads on any number of CPUs
    - OS will time-share the system in (we hope) a fair manner

- For comparison: a Unix/Linux “Process” is a similar OS
  construct, except that 2 Processes cannot directly share memory
OpenMP Basics

How does OpenMP know how many threads to use?

- It builds in the capability to use any number of threads
- What happens at run-time? It depends …
  - Sometimes it’s a system-default
  - It may be assigned to you by a queuing system
  - Sometimes it can be overridden by:

```
setenv OMP_NUM_THREADS 4
```

- This can be a problem!
  - It may not be efficient to run a small loop with lots of threads
    - If there isn’t much work to do, the overhead of the parallel region may be greater than the cost of doing the work on 1 thread

OpenMP Basics, cont’d

How does the compiler know where to end a parallel region?

- Fortran: you use ‘c$omp end parallel’
- C: the compiler parallelizes the NEXT STATEMENT only
  - Often, the next statement is a loop
    - Then the whole loop is parallelized
  - You can always make the next line be an opening curly-brace ‘{
    - Then everything within the braces will be parallelized

```c
int main() {
    #pragma omp parallel
    {
        printf("Hello World!\n");
        printf("\n", omp_get_thread_num());
        printf("Good-bye\n");
    }
    return 0;
}
```

These dangling braces can be confusing, use indentation and comments to improve readability
OpenMP Basics, cont’d

Why isn’t the output consistent?

- CPUs are inherently asynchronous … even though they are all running at the same speed and executing the exact same stream of computer instructions, someone “gets there first”
  - Note that there are operating system processes that pop up and execute now and then on different CPUs
- This is one of the most prevalent bugs in parallel programs!
  - Called a “Race Condition”

Outline

- CPU Architecture
  - “Shared Memory” vs. “Distributed Memory”
- What is OpenMP?
  - Operational Model for OpenMP
- OpenMP Programming
  - Loop Parallelism
  - Private vs. Shared Variables
  - Critical Regions
  - Parallel Sections
- Performance Issues/Hints
Loop Parallelism

- Loop-level parallelism is the most common approach to extracting parallel work from a computer program
  - E.g. you probably have lots of loops in your program that iterate over very large arrays:
    ```
    do i = 1, 1,000,000
    y(i) = alpha*x(i) + y(i)
    enddo
    ```
  - Instead of 1 CPU handling all 1,000,000 entries, we use N threads which each handle (1,000,000/N) entries
    ```
    c$omp parallel do
do i = 1, 1,000,000
    y(i) = alpha*x(i) + y(i)
    enddo
    c$omp end parallel
    ```
  - This is what OpenMP is really meant for!

Loop Parallelism, cont’d

- The compiler inserts code to compute the loop bounds for each thread
  - For 10 threads, thread 0 works on i = 1 to 100,000; thread 1 works on 100,001 to 200,000, etc.
  - Since this is a shared memory environment, every thread can “see” all data items in all arrays … so we don’t have to coordinate each thread’s loop bounds with it’s portion of each array
    - This often makes it easier to program in OpenMP than MPI
Data Sharing Examples

```
c$omp parallel do
do i = 1, 1000000
   a(i) = b(i+1) + c(i+5)*d(i-3)
enddo
c$omp end parallel
```

This is Ok since each thread is only READING different array locations and WRITEs do not conflict with each other (i.e. only one write to a(i))

```
c$omp parallel do
do i = 1, 1000000
   a(i) = 0.5*( a(i) + b(i-1) )
   b(i) = 2.0*c(i)*d(i)
enddo
c$omp end parallel
```

PROBLEM! — How do we know if b(i-1) is being computed right now by another thread?

Data Sharing Examples, cont’d

```
c$omp parallel do
   do i = 1, 1000000
      do j = 1, 1000000
         a(i,j-1) = b(i+1,j) + c(i)*d(j-1)
      enddo
   enddo
c$omp end parallel
```

This is Ok since each thread is only READING different array locations and WRITEs do not conflict (i.e. only one thread writes to a(i,j-1))

```
c$omp parallel do
   do i = 1, 1000000
      b(i) = 2.0*c(i) + d(i)
   enddo
c$omp end parallel
c$omp parallel do
   do i = 1, 1000000
      a(i) = 0.5*( a(i) + b(i-1) )
   enddo
c$omp end parallel
```

This is now Ok!
A More Complex Loop

```
do i = 1, 100
  do j = 1, 100
    x = (i-1)/100
    y = (j-1)/100
    d(i,j) = mandelbrot(x,y)
  enddo
enddo
```

```
for(i=0;i<100;i++) {
  for(j=0;j<100;j++) {
    x = i/100;
    y = j/100;
    d[i][j] = mandelbrot(x,y);
  }
}
```

Now, there is a function call inside the inner-most loop

```
c$omp parallel do
  do i = 1, 100
    do j = 1, 100
      x = (i-1)/100
      y = (j-1)/100
      d(i,j) = mandelbrot(x,y)
    enddo
  enddo
c$omp end parallel
```

```
#pragma omp parallel for
  for(i=0;i<100;i++) {
    for(j=0;j<100;j++) {
      x = i/100;
      y = j/100;
      d[i][j] = mandelbrot(x,y);
    }
  }
```

Any problems? (what is the value of 'x'?)

A More Complex Loop, cont’d

- Recall that all variables are shared across all threads
  - EXCEPT the loop-index/loop-bounds
  - So in this example, x and y are written and over-written by ALL threads for EVERY iteration in the loop
    - This example will compile, and even run, without any errors
    - But it will produce very strange results … since the value that a thread computes for x may be overwritten just before that thread calls the mandelbrot() function
      - e.g. Thread-0 sets x=0.0, y=0.0 and prepares to call the function
      - but just before the function call, Thread-1 writes x=0.5, y=0.5
      - so Thread-0 actually calls mandelbrot(0.5,0.5) instead of (0,0)
      - and then, because i and j are not shared (i=1, j=1), Thread-0 stores the result into d(1,1)
Private vs. Shared Variables

Single-thread portion of the program

“c$omp parallel” starts new parallel region

2 Threads are created, each with a different loop-range

“x=(i-1)/100” uses private-i and writes to global-x

What values of (x,y) are used depends on the actual system timing ... which can change each time the program is run - could change each iteration!

Private vs. Shared Variables, cont’d

OpenMP allows the programmer to specify if additional variables need to be Private to each Thread

◆ This is specified as a “clause” to the c$omp or #pragma line:

```c
#include <omp.h>

int main() {
    double x, y, d[100][100];
    int i, j;

    c$omp parallel do private(j,x,y) #pragma omp parallel for 
    do i = 1, 100
        do j = 1, 100
            x = (i-1)/100
            y = (j-1)/100
            d(i,j) = mandelbrot(x,y)
        enddo
        c$omp end parallel
    } for(i=0;i<100;i++) {
    for(j=0;j<100;j++) {
        x = i/100;
        y = j/100;
        d[i][j] = mandelbrot(x,y);
    }
}

return 0;
```

Private Variable Examples

```fortran
Private Variable Examples

    c$omp parallel do private(tmp1,tmp2)
    do i = 1, 1000000
        tmp1 = ( vm(i) - 75.0 )/alpha
        tmp2 = ( 33.0 - vm(i) )*beta
        a(i) = exp(tmp1)/(1.0-exp(tmp2))
    enddo
    c$omp end parallel

    struct xyz point1,point2;
    #pragma omp parallel for private(point1,d)
    for(i=0;i<1000000;i++) {
        point1.x = i+1;
        d = distance(point1,point2);
        if( d > 100.0 ) {
            printf("Error: distance too great");
        }
    }
```

Mixing Shared and Private Variables

What if we want to accumulate the total count from all of the mandelbrot() calculations?

```fortran
Mixing Shared and Private Variables

    total = 0;
    c$omp parallel do private(j,x,y)
    do i = 1, 100
        do j = 1, 100
            x = (i-1)/100
            y = (j-1)/100
            d(i,j) = mandelbrot(x,y)
            total = total + d(i,j)
        enddo
    enddo
    c$omp end parallel

    total = 0;
    #pragma omp parallel for \\ private(j,x,y)
    for(i=0;i<100;i++) {
        for(j=0;j<100;j++) {
            x = i/100;
            y = j/100;
            d[i][j] = mandelbrot(x,y);
            total += d[i][j];
        }
    }
```

We want ‘total’ to be shared (not private) … which is the default
Synchronization for Shared Variables

- This is really several discrete steps:
  - Read ‘total’ from global/shared memory into a CPU-register
  - Read ‘d(i,j)’ into a CPU-register
  - Add the two registers together
  - Store the sum back to the ‘total’ global/shared memory location

- There is a possible race condition: another Thread could have incremented ‘total’ just after this Thread read the value
  - So our new ‘total’ will be incorrect … but will overwrite any previous (correct) value

Synchronization … Critical Sections

- OpenMP allows user-defined Critical Sections
  - Only ONE THREAD at a time is allowed in a Critical Section

```c
total = 0;
c$omp parallel do private(j,x,y)
do i = 1, 100
do j = 1, 100
  x = (i-1)/100
  y = (j-1)/100
  d(i,j) = mandelbrot(x,y)
c$omp critical
  total = total + d(i,j)
c$omp end critical
d$omp end parallel
```

- Note that Critical Sections imply SERIAL computation
  - PARALLEL performance will likely be reduced!
“Named” Critical Regions

What if there are different regions, each of which needs a single-Thread restriction, but multiple such regions could execute simultaneously?

- E.g. compute min and max of a list of items
  - Updating of global-min must be single-Thread-only
  - Updating of global-max must be single-Thread-only
  - But one Thread can update min while another updates max

“Named” Critical Sections

- You can include a (Programmer-defined) name for each Critical Section
  - c$omp critical (MINLOCK)
  - c$omp critical (MAXLOCK)

Named Critical Region Example

```c
!$omp parallel do
do i = 1,1000000
  if( a(i) < cur_min ) then
    !$omp critical (MINLOCK)
    cur_min = a(i)
  !$omp end critical
  endif
  if( a(i) > cur_max ) then
    !$omp critical (MAXLOCK)
    cur_max = a(i)
  !$omp end critical
  endif
enddo
!$omp end parallel
```

Is this all we need?
Named Critical Region Example, cont’d

c$omp parallel do
do i = 1,1000000
  if( a(i) < cur_min ) then
    c$omp critical (MINLOCK)
      if( a(i) < cur_min ) then
        cur_min = a(i)
      endif
    c$omp end critical
  endif
  if( a(i) > cur_max ) then
    c$omp critical (MAXLOCK)
      if( a(i) > cur_max ) then
        cur_max = a(i)
      endif
    c$omp end critical
  endif
endo
c$omp end parallel

“Very Small” Critical Sections

- In some cases, we may use critical reasons to provide locks on small increments to a global counter, or other “small” computation
  - We don’t really care what the value is, we just want to increment it
  - What we really want is an ‘atomic’ update of that global variable
  - This uses unique hardware features on the CPU, so it is very fast
    - But there is a limit to what can be done

c$omp parallel do
do i = 1, 1000000
  c$omp atomic
    sum = sum + a(i)
endo
c$omp end parallel

Why bother?

Fortran:
+ - * /
& .AND. .OR. etc.
MIN() MAX()

C/C++:

#pragma omp parallel for
for(i=0;i<1000000;i++) {
  #pragma omp atomic
    sum += a[i];
}

#pragma omp parallel for
for(i=0;i<1000000;i++) {
  #pragma omp atomic
    sum += a[i];
}
Synchronization … Reduction Operations

Where Critical Sections can be any arbitrary piece of code, OpenMP also allows for optimized routines for certain standard operations:

- Accumulate across all Threads, Min across all Threads, etc.

- Called a “Reduction” Operation

```
total = 0
omp parallel do private(j,x,y)
omp reduction(+:total)
do i = 1, 100
   do j = 1, 100
      x = (i-1)/100
      y = (j-1)/100
      d(i,j) = mandelbrot(x,y)
      total = total + d(i,j)
   enddo
enddo
omp end parallel
```

Reduction Operations

- Fortran:
  - +, -, *, .AND., .OR., .EQV., .NEQV.
  - MIN, MAX
  - IAND, IOR, IEOR

- C:
  - +, -, *, &&, ||, &

Suitable initial values are chosen for the reduction variable:
- E.g. ‘reduction(+:total)’ initializes ‘total=0’ in each Thread
Initializing Private Variables

What if we want each Thread’s private variable to start off at a specific value?

- Reduction operations automatically initialize the values to something meaningful (0 or 1)

- Other major option is to copy the global value into each Thread’s private variable
  
  Called “firstprivate(...)” clause
  
  ```
  var1 = 0
  c$omp parallel do private(x) firstprivate(var1)
  do i = 1, 100
    x = (i-1)/100
    var1 = var1 + f(x)
  enddo
  print *, 'partial sum', var1
  c$omp end parallel
  ```

At the end of the loop, (global) var1 is UNCHANGED!

```
var1 == 0
```

Private vs. Firstprivate vs. Copyin vs …

<table>
<thead>
<tr>
<th></th>
<th>Initial Value of the Local Var</th>
<th>Final Value of the Global Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>Undefined</td>
<td>Undefined</td>
</tr>
<tr>
<td>Firstprivate</td>
<td>Global is copied to each Thread’s Local</td>
<td>Whatever value was there prior to entering the parallel region</td>
</tr>
<tr>
<td>Lastprivate</td>
<td>Undefined</td>
<td>Master’s Value is copied to Global</td>
</tr>
<tr>
<td>Reduction</td>
<td>0 or 1 (depends on operand)</td>
<td>Global is sum/min/max of Threads’ Locals</td>
</tr>
<tr>
<td>Copyin</td>
<td>Master’s Local is copied to Thread’s Local</td>
<td>Undefined</td>
</tr>
</tbody>
</table>
OpenMP and Function Calls

What happens inside the mandelbrot() function?
- Are variables defined inside mandelbrot() private or shared?

Function Calls, cont’d

In C and Fortran, variables defined inside a function are placed on the stack, NOT in a “global” memory area
- Each Thread has its own stack
- …so each Thread will have its own set of function-variables on its own stack
  - i.e. variables defined in a function are PRIVATE
OpenMP and Function Calls, Take-II

However, if you have a parallel section that is defined wholly inside a function, then the “usual” rules apply:

- E.g. Master thread, running by itself, calls a function which breaks into a parallel region … then all function-local variables CAN be seen (shared) by all threads.

```fortran
subroutine mandelbrot(x,y)    subroutine mandelbrot( float x, float y ) { 
  real x, y             float temp1, temp2; 
  real temp1, temp2     int it_ctr; 
  integer it_ctr

  !$omp parallel do
  do i = 1, N
    temp1, temp2, it_ctr are SHARED
    . . .
  enddo
  mandelbrot = it_ctr
end subroutine
```

Parallel Sections

- Loop-level parallelism tends to imply a Domain Decomposition or Iterative Parallelism approach to splitting a problem up.

- Parallel Sections can be used if there are fundamentally different algorithms that do not need to interact:
  - One Thread needs to act as Server while the others act as Clients
  - One Thread handles all I/O while the others “do the work”

- Imagine 2 loops which compute the sum and sum-squared of a set of data items:
  - You could split up each loop with its own ‘c$omp parallel do’
  - Or you could split the whole program up so that one thread computes the sum and another thread computes sum-squared
    - What if we have more than 2 CPUs?
Parallel Sections, cont’d

```c
!$omp parallel do
  do i = 1,num
    val = val + data(i)
  enddo
$omp parallel do
  do i = 1,num
    val2 = val2 + data(i)*data(i)
  enddo
$$omp end parallel sections
```

A Few More Tidbits

- **omp_get_wtime()**
  - Returns the wall-clock time (in seconds, double-precision value)
  - **omp_get_wtick()** returns time-resolution of `wtime`
  - Note that other system timers return “User Time” which is double
    the wall-clock time with 2 threads, etc.

- **omp_init_lock( &lock )**
- **omp_set_lock( &lock )**
- **omp_test_lock( &lock )**
- **omp_unset_lock( &lock )**
- **omp_destroy_lock( &lock )**
  - User-defined locks for critical regions of code
  - Number of locks can be determined at run-time (CRITICAL
    sections are defined at compile-time)
The FLUSH Directive

Modern CPU architecture doesn’t entirely “play nice” with shared memory programming

- Cache memory is used to store recently used data items
- What happens if a cached memory item is needed by another Thread?
  - i.e. the data-item currently resides in cache on CPU#1 and is not visible to CPU#2, how do we force CPU#1 to write the data to main memory?

- #pragma omp flush (var1,var2,var3)
- c$omp flush

FLUSH Directive, cont’d

```c
while( (volatile)(GlobalFlag) == 0 ) {
  ...  
  /* on some other thread */
  GlobalFlag = 1;
}
```

(1) Without ‘volatile’ compiler may remove the loop
(2) First time through loop, GlobalFlag is read into cache... and may never be re-read

```c
while( (volatile)(GlobalFlag) == 0 ) {
  #pragma omp flush
  ...  
  /* on some other thread */
  GlobalFlag = 1;
  #pragma omp flush
```

Make sure we read new value

Fortran: may need to tag GlobalFlag as ‘static’ or ‘save’
Outline

- CPU Architecture
  - “Shared Memory” vs. “Distributed Memory”

- What is OpenMP?
  - Operational Model for OpenMP

- OpenMP Programming
  - Loop Parallelism
  - Private vs. Shared Variables
  - Critical Regions
  - Parallel Sections

Performance Issues/Hints

Basic Tips

- OpenMP is built for large-loop parallelism with little synchronization
  - If your loops require tight synchronization, you may have a hard time programming it in OpenMP

This kind of loop can be pipelined

```c
for (i=0; i<N; i++) {
    x[i] = 0.5*(x[i-1] + x[i+1]);
}
```

- It is sometimes easier to have a single (outer) OpenMP parallel-for loop which makes function calls to a “worker” routine

```c
c$omp parallel do private(x)
    do i=1,N
        x = i*0.1
        call DoWork(x)
    enddo
```
One “Big” Parallel Region vs. Several Smaller Ones

Each parallel region has an implied barrier (high cost)

One “Big” Parallel Region, cont’d

This is still a parallel section

One (larger) parallel region has only one implied barrier (lower cost).

Sometimes, duplicate computation is cheaper than having a single thread compute a value and “broadcast” it to all the others.
Single-CPU Region within a Parallel Region

- Master -- only 1 Thread (and only the Master Thread) executes the code
- Single -- only 1 Thread (chosen at random) executes the code

```c
#pragma omp parallel
  #pragma omp for
  for (i = 0; i < N; i++)
  
    y[i] = alpha * x[i] + y[i]

#pragma omp end parallel
```

Performance Issue with Small Loops

- Each OpenMP parallel region (loop or section) has an overhead associated with it
  - Non-zero time to start up N other Threads
  - Non-zero time to synchronize at the end of the region
- If you have a very fast CPU and a very small loop, these overheads may be greater than the cost (time) to just do the loop on 1 CPU
  - For small loops, we can tell OpenMP to skip the parallelism by using an ‘if’ clause
    - i.e. only run the loop in parallel *IF* a certain condition holds

```c
#pragma omp parallel do if(N>1000)
  do i = 1, N
    if (N>1000)
      y(i) = alpha * x(i) + y(i)  for (i=0; i<N; i++)
    enddo
    y[i] = alpha * x[i] + y[i]
  enddo
```

Where you should put this cut-off is entirely machine-dependent
Performance of Critical Sections

- Critical Sections can cause huge slow-downs in parallel code
  - Recall, a Critical Section is serial (non-parallel) work, so it is not surprising that it would impact parallel performance

- Many times, you really want a ‘Reduction’ operation
  - ‘Reduction’ will be much faster than a critical section

- For “small” Critical Sections, use ‘Atomic’ updates
  - But check the compiler docs first …
    - The Sun compilers turn ‘atomic’ blocks into comparable ‘critical’ sections

Alternatives to Critical Sections

```c
integer temp(MAX_NUM_THREADS), sum
integer self
c  assume temp(:) = 0 to start

c$omp parallel private(self)
  self = omp_get_thread_num()
c$omp do
do i = 1, N
  temp(self) = temp(self) + val
endo
c$omp enddo
c$omp end parallel
sum = 0
do i = 1, MAX_NUM_THREADS
  sum = sum + temp(i)
endo
```
“Dangerous” Performance Enhancement

◆ You can tell OpenMP to not insert barriers at the end of parallel regions
 ◆ This is “dangerous,” since with no barrier at the end, the Master Thread may begin running code that ASSUMES certain variables are stable

```c
omp parallel do nowait
   do i = 1, N
       y(i) = alpha*x(i) + y(i)
   enddo
omp end parallel
print *, 'y(100)=', y(100)
```

With no barrier at the end of the parallel region, we cannot guarantee that y(100) has been computed yet

Race Conditions

◆ A “Race” is a situation where multiple Threads are making progress toward some shared or common goal where timing is critical

◆ Generally, this means you’ve ASSUMED something is synchronized, but you haven’t FORCED it to be synchronized

◆ E.g. all Threads are trying to find a “hit” in a database
  ◆ When a hit is detected, the Thread increments a global counter
  ◆ Normally, hits are rare events and so incrementing the counter is “not a big deal”
  ◆ But if two hits occur simultaneously, then we have a race condition
    ● While one Thread is incrementing the counter, the other Thread may be reading it
    ● Hence it may read the old value or new value depending on the exact timing of the two Threads
Race Conditions, cont’d

Some symptoms:

- Running the code with 4 Threads works fine, running with 5 causes erroneous output
- Running the code with 4 Threads works fine, running it with 4 Threads again produces different results
- Sometimes the code works fine, sometimes it crashes after 4 hours, sometimes it crashes after 10 hours, …
- Program sometimes runs fine, sometimes hits an infinite loop
- Program output is sometimes jumbled (the usual line-3 appears before line-1)

- Any kind of indeterminacy in the running of the code, or seemingly random changes to its output, could indicate a race condition

Race Conditions, cont’d

Race Conditions can be EXTREMELY hard to debug

- Generally, race conditions don’t cause crashes or even logic-errors
  - E.g. a race condition leads two Threads to both think they are in charge of data-item-X … nothing crashes, they just keep writing and overwriting X (maybe even doing duplicate computations)
- Often, race conditions don’t cause crashes at the time they actually occur … the crash occurs much later in the execution and for a totally unrelated reason
  - E.g. a race condition leads one Thread to think that the solver converged but the other Thread thinks we need another iteration … crash occurs because one Thread tried to compute the global residual
- Sometimes race conditions can lead to deadlock
  - Again, this often shows up as seemingly random deadlock when the same code is run on the same input
Deadlock

A “Deadlock” condition occurs when all Threads are stopped at synchronization points and none of them are able to make progress

- Sometimes this can be something simple:
  - Thread-1 enters a Critical-Section and waits for Variable-X to be incremented, and then it will release the lock
  - Thread-2 wants to increment X but needs to enter the Critical-Section in order to do that

- Sometimes it can be more complex or cyclic:
  - Thread-1 enters Crit-Sec-1 and needs to enter Crit-Sec-2
  - Thread-2 enters Crit-Sec-2 and needs to enter Crit-Sec-3
  - Thread-3 enters Crit-Sec-3 and …

A related condition is “Starvation”: when one Thread needs a resource (Critical Section) that is never released by another Thread

Deadlock, cont’d

- Symptoms:
  - Program hangs (!)

- Debugging deadlock is often straightforward … unless there is also a race condition involved

  - Simple approach: put print statement in front of all Critical Section and Lock calls

```
Thread-1 acquired lock on region-A
Thread-1 acquired lock on region-C
Thread-2 acquired lock on region-B
Thread-1 release lock on region-C
Thread-3 acquired lock on region-C
Thread-1 waiting for lock on region-B
Thread-2 waiting for lock on region-A
```
Static Scheduling for Fixed Work-loads

- OpenMP allows dynamic scheduling of the “work-load” inside, e.g., a Parallel ‘Do’ Loop
  - If one CPU gets loaded down (by OS tasks), other CPUs will pick up additional iterations in the loop to offset that performance-loss
  - Generally, this works well -- you don’t have to worry about all the random things that might happen, OpenMP does the “smart” thing

- However, this dynamic scheduling implies some additional “cost”
  - If nothing else, the CPUs must check if other CPUs are loaded

- If your work-load inside a parallel region is fixed, then you may want a “static” work-assignment
  - E.g. the loop is exactly 1,000,000 iterations, so the CPUs should take 100,000 each

---

Static Scheduling, cont’d

```c
#pragma omp parallel do schedule(static)
  do i = 1, 1000000
    y(i) = alpha*x(i) + y(i)
  enddo
#pragma omp end parallel
```

```c
for(i=0;i<1000000;i++)
  y[i] = alpha*x[i] + y[i]
```

If the per-iteration work-load is variable, or you are sharing the CPUs with many other users, then a dynamic scheduling algorithm (the default) may be better.
Moving Towards Pthreads

Pthreads (POSIX Threads) is the other major shared-memory programming approach

- It is a little bit like the OpenMP ‘Section’ Directive
- E.g. you use pthread_create() to create a new Thread and assign it a “work-function”

You can use the OpenMP Library Functions to create your own splitting of the loops:

- self_thread = omp_get_thread_num()
- num_thr = omp_get_num_threads()
- num_proc = omp_get_num_procs()

- You may be able to avoid some of the overheads with multiple parallel regions by “forcing” a consistent splitting onto your data and loops

Alternative Approach to OpenMP

```
dataize = 1000000
#c omp parallel private(i,self_thr,num_thr,istart,ifinish)
self_thr = omp_get_thread_num()
num_thr = omp_get_num_threads()
istart = (self_thr-1)*(dataize/num_thr) + 1
ifinish = istart + (dataize/num_thr)
do i = istart, ifinish
    y(i) = alpha*x(i) + y(i)
endo
c omp end parallel
```

```
dataize = 1000000
#pragma omp parallel private(i,self_thr,num_thr,istart,ifinish)
{
    self_thr = omp_get_thread_num();
    num_thr = omp_get_num_threads();
    istart = (self_thr-1)*(dataize/num_thr);
    ifinish = istart + (dataize/num_thr);
    for(i=istart;i<ifinish;i++) {
        y[i] = alpha*x[i] + y[i];
    }
}
```
For More Info

- “Parallel Programming in OpenMP”
  - Chandra, Dagum, Kohr, Maydan, McDonald, Menon

- http://openmp.org
  - Good list of links to other presentations and tutorials

- Many vendors have OpenMP capability in their compilers
  - Often have good end-user documentation too
  - IBM Redbooks, tech-notes, “porting” guides

Where to go from here?

- http://www.csem.duke.edu . . . csem@duke.edu

- Get on our mailing lists:
  - Scientific Visualization seminars (Friday 12-1pm)
  - CSEM seminars
  - Workshop announcements

- Follow-up seminars on specific topics
  - Parallel programming methods (OpenMP, Pthreads, MPI)
  - Performance tuning methods (Vtune, Parallel tracing)
  - Visualization tools (Amira, AVS, OpenDX)

- Contacts:
  - Dr. Bill Rankin, wrankin@ee.duke.edu
  - Dr. Rachael Brady, rbrady@duke.edu
  - Dr. John Pormann, jbp1@duke.edu
Private Variables (Detail)

real x

c$omp parallel do
  do i = 1, 1000000
    x = (i)*dx
    a(i) = 3.0*x + 4.0*(x**2)
  enddo

This 'x' is global

real x

c$omp parallel do private(x)
  do i = 1, 1000000
    x = (i)*dx
    a(i) = 3.0*x + 4.0*(x**2)
  enddo
  print *, x

A new 'x' is created which is Private

This will print the global 'x'
(currently undefined)