OpenMP 3

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- Design, implement, and analyze parallel programming using OpenMP for shared-memory system.
If sum is a shared variable, this loop cannot run in parallel

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

    
}
```

We can use a critical region for this:

```c
for (i=0; i < n; i++){
    
    #one at a time can proceed

    sum += a[i];

    
    #next in line, please

}
```
CRITICAL region

- Useful to avoid a race condition, or to perform I/O (but that still has random order)
- Be aware that there is a cost associated with a critical region
CRITICAL example

```c
#pragma omp critical

#pragma omp parallel shared(x)
{
    #pragma omp critical
    {
        // only one thread in here
    }
} // implicit barrier
```
CRITICAL and ATOMIC Constructs

**Critical:** All threads execute the code, but only one at a time:

```c
#pragma omp critical [(name)]
{<code-block>}
```

```c
!$omp critical [(name)]
    <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit!

**Atomic:** Only the loads and store are atomic ....

```c
#pragma omp atomic
    <statement>
```

```c
!$omp atomic
    <statement>
```

This is a lightweight, special form of a critical section

```c
#pragma omp atomic
    a[indx[i]] += b[i];
```
Atomic vs Critical

- Syntax

```c
#pragma omp atomic
expression
```

```c
#pragma omp critical [(name)]
{
    code_block
}
```
Lab: MAX without Critical

```c++
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <omp.h>

using namespace std;
#define SIZE 10000

int main ( int argc, char *argv[] )
{
    int i, p, max_val;
    int a[SIZE];

    // randomly initialize the a[i] array
    srand(time(NULL));
    for (i = 0; i < SIZE; i++)
    {
        a[i] = rand() % 10000;
        cout << a[i] << " ";
    }
    cout << endl;

    omp_set_num_threads(40);
    p = omp_get_num_threads();
    cout << "Number of threads = " << p << endl;

    // serial max find
    max_val = a[0];
    for (i = 1; i < SIZE; i++)
    {
        if (a[i] > max_val)
        {
            max_val = a[i];
        }
    }
    cout << "max = " << max_val << endl;

    // parallel max find
    max_val = a[0];
    #pragma omp parallel
    {
        #pragma omp master
        {
            p = omp_get_num_threads();
            cout << "Number of threads = " << p << endl;
        }
        #pragma omp for
        for (i = 1; i < SIZE; i++)
        {
            // #pragma omp critical
            if (a[i] > max_val)
            {
                max_val = a[i];
            }
        }
    }
    cout << "max = " << max_val << endl;
}
```
Lab: MAX without Critical

39 3226 5554 9564 5939 3641 9677 6087 9594 5705 3706 8511 6290 780 5099 5379 5810 7898 4743 7155 1620 6067 5217 4053 2148 2233 4456 8426 2226 4133 4514 1820 6191 4572 332 2402 5866 158

** Number of threads = 1
max = 9998

** Number of threads = 40
max = 9997

[ahnt@hopper:~/Course/csci4850/2018S/Lectures/codes/week4]$
Lab: MAX with Critical

79093 380173 584545 754459 728299 83580 37023
Number of threads = 1
max = 999971

Number of threads = 40
max = 999971

[ahnt@hopper:~/Course/csci4850/2018S/Lectures/codes/week4]$
Locks

- Locks are provided through `omp.h` library calls
  - `omp_init_lock()`
  - `omp_destroy_lock()`
  - `omp_test_lock()`
  - `omp_set_lock()`
  - `omp_unset_lock()`
Reduction?

- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:
REDUCTION Clause

Purpose:
- The REDUCTION clause performs a reduction operation on the variables that appear in its list.
- A private copy for each list variable is created and initialized for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Format:

<table>
<thead>
<tr>
<th>Fortran</th>
<th>REDUCTION (operator: list)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>reduction (operator: list)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fortran</th>
<th>C/C++</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>Logical AND</td>
<td>.and.</td>
<td>&amp;&amp;</td>
<td>0</td>
</tr>
<tr>
<td>Logical OR</td>
<td>.or.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND bitwise</td>
<td>iand</td>
<td>&amp;</td>
<td>all bits on / 1</td>
</tr>
<tr>
<td>OR bitwise</td>
<td>ior</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusive OR bitwise</td>
<td>ieor</td>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>Equivalent</td>
<td>.equiv.</td>
<td></td>
<td>.true.</td>
</tr>
<tr>
<td>Not Equivalent</td>
<td>.neqv.</td>
<td></td>
<td>.false.</td>
</tr>
<tr>
<td>Maximum</td>
<td>max</td>
<td>max</td>
<td>Most negative #</td>
</tr>
<tr>
<td>Minimum</td>
<td>min</td>
<td>min</td>
<td>Largest positive #</td>
</tr>
</tbody>
</table>
OpenMP MIN MAX Reduction

- OpenMP 3.1 added predefined min and max reduction operators for C and C++

- How to check the version of OpenMP on Linux
  - It appears that the C/C++ specification for OpenMP provides no direct way of doing this programmatically. So you have to check the docs for your compiler version.
  - $ gcc --version ## get compiler version
  - As of GCC 4.2, the compiler implements version 2.5 of the OpenMP standard and as of 4.4 it implements version 3.0 of the OpenMP standard. The OpenMP 3.1 is supported since GCC 4.7.
// parallel max find
max_val = a[0];
#pragma omp parallel
{
    #pragma omp master
    {
        p = omp_get_num_threads();
        cout << "Number of threads = " << p << endl;
    }
    #pragma omp for reduction(max: max_val)
    for (i = 1; i < SIZE; i++) {
        if (a[i] > max_val) {
            max_val = a[i];
        }
    }
}
cout << "max = " << max_val << endl;
False Sharing

- False sharing occurs when threads on different processors modify variables that reside on the same cache line. This invalidates the cache line and forces a memory update to maintain cache coherency.

- Memory addresses are grouped into cache lines
- If one element of the cache line is changed, the whole line is invalidated
False Sharing

- False sharing degrades performance when all of the following conditions occur.
  - Shared data is modified by multiple processors.
  - Multiple processors update data within the same cache line.
  - This updating occurs very frequently (for example, in a tight loop).
  - Note that shared data that is read-only in a loop does not lead to false sharing.

- In general, false sharing can be reduced by
  - making use of private data as much as possible;
  - utilizing the compiler’s optimization features to eliminate memory loads and stores.
void nesting(int n) {
    int i, j;
    #pragma omp parallel
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}
void nesting(int n) {
    int i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            innerloop(i,n);
        }
    }
}

void innerloop(int i, int n) {
    int j;
    #pragma omp parallel
    {
        #pragma omp for
        for (j=0; j < n; j++)
            work(i, j);
    }
}
Nested Loop

- A PARALLEL section within a PARALLEL section
- Nested parallelism is off by default:
  - the inner PARALLEL section gets only 1 thread
- Turn on nested parallelism by:
  - using `omp_set_nested(1)`
  - setting the OMP_NESTED environment variable to “TRUE”
Calculate Pi (Integration)

\[ y = \sqrt{1-x^2} \]

Removed to create a well-defined function

\[ \sqrt{1-x_i^2} \cdot h \]

\[ x = -1 + (i + 0.5) \cdot h; \]

\[ \text{sum} += \text{sqrt}(1 - x \cdot x) \cdot h; \]

\[ \pi = \text{sum} \cdot 2.0; \]
HW3: Calculate Pi (Monte Carlo Method) using OpenMP

\[ \rho = \frac{\text{Area of Circle}}{\text{Area of Square}} = \frac{\pi}{4} = 0.7853981633974483 \]