OpenMP 4

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

● Our plan for the week... Active learning!

  ▪ We will mix short lectures and short exercises to learn about various OpenMP topics.
  ▪ Do the exercise we assign and then change things around and experiment.

  ➔ Embrace Active Learning
Synchronization Constructs

Consider a simple example where two threads on two different processors are both trying to increment a variable \( x \) at the same time (assume \( x \) is initially 0):

**THREAD 1:**
```
increment(x)
{
    x = x + 1;
}
```

**THREAD 2:**
```
increment(x)
{
    x = x + 1;
}
```

10 LOAD A, (x address)
20 ADD A, 1
30 STORE A, (x address)

What’s the value of \( x \)?
Synchronization Constructs

- One possible execution sequence:
  1. Thread 1 loads the value of $x$ into register A.
  2. Thread 2 loads the value of $x$ into register A.
  3. Thread 1 adds 1 to register A.
  4. Thread 2 adds 1 to register A.
  5. Thread 1 stores register A at location $x$.
  6. Thread 2 stores register A at location $x$.

- The resultant value of $x$ will be 1, not 2 as it should be.

- To avoid a situation like this, the incrementing of $x$ must be synchronized between the two threads to ensure that the correct result is produced.

- OpenMP provides a variety of Synchronization Constructs that control how the execution of each thread proceeds relative to other team threads.
Synchronization Constructs

- Remember these four synchronization constructs

```c
#pragma omp master
#pragma omp critical
#pragma omp atomic
#pragma omp barrier
```
The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.

There is no implied barrier associated with this directive.

```
#pragma omp master
{<code-block>}

!$omp master
  <code-block>
!$omp end master
```
CRITICAL directive

If sum is a shared variable, this loop can not run in parallel

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

We can use a critical region for this:

```
for (i=0; i < n; i++){
    ....
    // critical region
    sum += a[i];
    // critical region
    ....
}
```

one at a time can proceed
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

#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>}

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

!$omp atomic
 <statement>
```

This is a lightweight, special form of a critical section

```c
#pragma omp atomic
 a[indx[i]] += b[i];
```
Lab Again

- Count the primes from 1 to 100000.
- This is an unbalanced work load, particular for two threads.
- Demonstrate static and dynamic scheduling (test with different chunks, e.g., 100, 500).
- Try to define shared and private variables correctly.
- Compare “reduction” to “shared” and “critical”
- Compare “reduction” to “shared” and “atomic”
single and critical are two very different things.

- single specifies that a section of code should be executed by single thread (not necessarily the master thread)
- critical specifies that code is executed by one thread at a time

```c
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical
    b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

- will print
  single: 1 -- critical: 4
**omp critical and omp single**

- **single** and **critical** belong to two completely different classes of OpenMP constructs.
- **single** is a worksharing construct, alongside for and sections.
- Worksharing constructs are used to distribute a certain amount of work among the threads.
- Such constructs are "collective" in the sense that in correct OpenMP programs all threads must encounter them while executing and moreover in the same sequential order, also including the barrier constructs.
omp critical and omp single

- The three worksharing constructs cover three different general cases:
  - **for** (a.k.a. loop construct): distributes automatically the iterations of a loop among the threads - in most cases all threads get work to do;
  - **sections**: distributes a sequence of independent blocks of code among the threads - some threads get work to do. This is a generalization of the for construct as a loop with 100 iterations could be expressed as e.g. 10 sections of loops with 10 iterations each.
  - **single**: singles out a block of code for execution by one thread only, often the first one to encounter it (an implementation detail) - only one thread gets work. single is to a great extent equivalent to sections with a single section only.
omp critical and omp single

- A common trait of all worksharing constructs is the presence of an implicit barrier at their end.
- The implicit barrier might be turned off by adding the nowait clause to the corresponding OpenMP construct.
- But the standard does not require such behavior and with some OpenMP runtimes the barrier might continue to be there despite the presence of nowait.
- Incorrectly ordered (i.e. out of sequence in some of the threads) worksharing constructs might therefore lead to deadlocks.
- A correct OpenMP program will never deadlock when the barriers are present.
critical is a synchronization construct, alongside master, atomic, and others. Synchronization constructs are used to prevent race conditions and to bring order in the execution of things.

- critical prevents race conditions by preventing the simultaneous execution of code among the threads in the so-called contention group. This means all threads from all parallel regions encountering similarly named critical constructs get serialized;

- atomic turns certain simple memory operations into atomic ones, usually by utilizing special assembly instructions. Atomics complete at once as a single non-breakable unit. For example, an atomic read from some location by one thread, which happens concurrently with an atomic write to the same location by another thread, will either return the old value or the updated value, but never some kind of an intermediate mash-up of bits from both the old and the new values;
omp master vs omp single

- **master** singles out a block of code for execution by the master thread (thread with ID of 0) only.

- **Unlike single**, there is no implicit barrier at the end of the construct and also there is no requirement that all threads must encounter the master construct.

- Also, the lack of implicit barrier means that master does not flush the shared memory view of the threads.

- **master** is basically a shorthand for if (omp_get_thread_num() == 0) { ... }.
Implicit Barrier

– beginning and end of parallel constructs
– end of all other control constructs

```c
#pragma omp parallel
{
    printf("Hello!\n");
}
```
Suppose we run each of these two loops in parallel over i:

```c
for (i=0; i < N; i++)
    a[i] = b[i] + c[i]
```

```c
for (i=0; i < N; i++)
    d[i] = a[i] + b[i]
```

This may give us a wrong answer. Why?
Suppose we run each of these two loops in parallel over i:

```c
for (i=0; i < N; i++)
    a[i] = b[i] + c[i]
```

```
for (i=0; i < N; i++)
    d[i] = a[i] + b[i]
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point.
#pragma omp barrier

Barrier syntax in OpenMP:

```
#pragma omp barrier

!$omp barrier
```
#pragma omp for nowait

- **nowait** – Removes implicit barrier from end of block
  - To minimize synchronization, some OpenMP directives/pragmas support the optional `nowait` clause
  - If present, threads do not synchronize/wait at the end of that particular construct
  - In Fortran the `nowait` clause is appended at the closing part of the construct
  - In C, it is one of the clauses on the pragma

```c
#pragma omp for nowait
{
    :
}

!$omp do
    :
!$omp end do nowait
```
A more elaborate example

```c
#pragma omp parallel if (n>limit) default(none) \
    shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;

#pragma omp for nowait
    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];

#pragma omp for nowait
    for (i=0; i<n; i++)
        a[i] = b[i] + c[i];

#pragma omp barrier
    ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
    ....
} /*-- End of parallel region --*/
```
The MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code.

There is no implied barrier associated with this directive.

```
#pragma omp master
{<code-block>}

!$omp master
  <code-block>
!$omp end master
```
Time Logging

- OpenMP:
  
  ```c
  double time1 = omp_get_wtime();
  ```

- Linux “time” command:

  The **time** command runs the specified program `command` with the given arguments. When `command` finishes, **time** writes a message to standard output giving timing statistics about this program run.
time command

- **Real**: this is the wall clock time. If other processes are running at the same time, they will slow down your process and thus will increase "real".

- **User**: the time that CPU spent on your program in user mode. (Kernel mode does not counted in this. For example you requested a disk IO and your disk is very slow. Such a system call is invoked on kernel mode, therefore it will not be reflected to "user".)

- **Sys**: the time that CPU spent on kernel mode during the execution. Kernel mode contains operations like disk IO, network IO, devices, memory allocation etc. (Part of the mem. allocation is still in user space, though.)
[ahnt@turing:~/Courses/CSCI4850/Lab12]$ time ./lab12

SCHEDULE_OPENMP
C++/OpenMP version
Count the primes from 1 to N.
This is an unbalanced work load, particular for two threads.
Demonstrate static and dynamic scheduling.

Number of processors available = 8
Number of threads = 2

<table>
<thead>
<tr>
<th>n</th>
<th>primes</th>
<th>static time</th>
<th>dynamic time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>9592</td>
<td>1.44031</td>
<td>0.978357</td>
</tr>
</tbody>
</table>

real 0m2.422s
user 0m3.919s
sys 0m0.004s
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”
Lab: Nested loop test

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

#include <omp.h>

using namespace std;

void report_num_threads(int level);

void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level: %d Number of threads: %d Thread number: %d \n",
            level, omp_get_num_threads(), omp_get_thread_num());
    }
}

int main(int argc, char *argv[])
{
    omp_set_nested(1); // test w/wo omp_set_nested(1)
    #pragma omp parallel num_threads(4)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(4)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(4)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```
Tasking in OpenMP

- Tasking was introduced in OpenMP 3.0
- Until then it was impossible to efficiently and easily implement certain types of parallelism
- The initial functionality was very simple by design
  - The idea was (and still) is to augment tasking as we collectively gain more insight and experience
- Note that tasks can be nested
The Tasking Concept In OpenMP
Who Does What And When?

- **Developer**
  - Use a pragma to specify where the tasks are
    (The assumption is that all tasks can be executed independently)

- **OpenMP runtime system**
  - When a thread encounters a task construct, a new task is generated.
  - The moment of execution of the task is up to the runtime system.
  - Execution can either be immediate or delayed.
  - Completion of a task can be enforced through task synchronization.
The Tasking Construct

- C, C++
  
  \#pragma omp task

- Fortran
  
  !$omp task

Defines a task
Task Synchronization

There are two task synchronization constructs

- **Barrier**
  
  ```
  #pragma omp barrier
  !$omp barrier
  ```

- **Taskwait**
  
  ```
  #pragma omp taskwait
  !$omp taskwait
  ```
Task Completion

- Explicitly wait on the completion of child tasks:

```
#pragma omp flush taskwait
!$omp flush taskwait
```

- **Flush**
  - Different threads can have different values for the same shared variable
  - flush assures that the calling thread has a consistent view of memory
Your task for Today:

Write a program that prints either “A race car” or “A car race” and maximize the parallelism with 2 threads.
```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main (int argc, char *argv[]) {
    cout << "A ";
    cout << "race ";
    cout << "car ";
    cout << endl;
    cout << endl;
    return 0;
}
```

```
$g++ ex1.cpp -o ex1
$ ./ex1
A race car
```
# include <iostream>
# include <stdio.h>
# include <omp.h>

using namespace std;

int main (int argc, char *argv[]) {
    #pragma omp parallel
    {
        cout << "A ";
        cout << "race ";
        cout << "car ";
    } // end parallel region

cout << endl;

    return 0;
}

What will this program print using 2 threads?
Lab: OpenMP Task

$ g++ -fopenmp omp_task.cpp -o omp_task
$ export OMP_NUM_THREADS=2
$ ./omp_task
A race car A race car
$ ./omp_task
A A race race car car
```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main (int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            cout << "A ";
            cout << "race ";
            cout << "car ";
        } // end pragma omp single
    } // end pragma omp parallel

    cout << endl;

    return 0;
}
```

What will this program print using 2 threads?
Lab: OpenMP Task

```
$ g++ -fopenmp omp_task.cpp -o omp_task
$ export OMP_NUM_THREADS=2
$ ./omp_task
A race car
```

But of course now only 1 thread executes ........
# include <iostream>
# include <stdio.h>
# include <omp.h>

using namespace std;

int main (int argc, char *argv[])
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            cout << "A ";

            #pragma omp task
            {
                cout << "race ";
            } // end pragma omp task

            #pragma omp task
            {
                cout << "car ";
            } // end pragma omp task
        } // end pragma omp single
    } // end pragma omp parallel
    cout << endl;
    return 0;
}

What will this program print using 2 threads?
$ g++ -fopenmp omp_task.cpp -o omp_task
$ export OMP_NUM_THREADS=2
$ ./omp_task
A race car
$ ./omp_task
A car race
Lab: OpenMP Task

● Your task for Today:

Have the sentence end with "is fun to watch"
```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

int main (int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            cout << "A ";
            #pragma omp task
            {
                cout << "race ";
            } // end pragma
            #pragma omp task
            {
                cout << "car ";
            } // end pragma
cout << "is fun to watch " << endl;
        } // end pragma
    } // end pragma
    cout << endl;
    return 0;
}
```

What will this program print using 2 threads?
Lab: OpenMP Task

```
$ g++ -fopenmp omp_task.cpp -o omp_task
$ export OMP_NUM_THREADS=2
$ ./omp_task
A race is fun to watch car
$ ./omp_task
A is fun to watch race car
$ ./omp_task
A race car is fun to watch
```
# include <iostream>
# include <stdio.h>
# include <omp.h>

using namespace std;

int main (int argc, char *argv[])
{

    #pragma omp parallel
    {
        #pragma omp single
        {
            cout << "A ";
            #pragma omp task
            {
                cout << "race ";
            } // end pragma omp task
            #pragma omp task
            {
                cout << "car ";
            } // end pragma omp task
            #pragma omp taskwait
            cout << "is fun to watch " << endl;
        } // end pragma omp single
    } // end pragma omp parallel

    cout << endl;
    return 0;
}
Lab: OpenMP Task

```
$ g++ -fopenmp omp_task.cpp -o omp_task
$ export OMP_NUM_THREADS=2
$ ./omp_task
A car race is fun to watch
$ ./omp_task
A race car is fun to watch
```
OpenMP Resources

- https://computing.llnl.gov/tutorials/openMP/
- https://computing.llnl.gov/tutorials/openMP/exercise.html