OpenMP 2

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.
What is OpenMP?

- What does OpenMP stand for?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia

- Application Programming Interface (API) for multi-threaded parallelization consisting of
  - Source code directives
  - Functions
  - Environment variables

- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors
How to compile and run OpenMP programs?

- gcc 4.2 and above supports OpenMP 3.0
  - gcc –fopenmp a.c
  - g++ -fopenmp a.cpp

- To run: ‘a.out’
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh)
    - export OMP_NUM_THREADS=4 (bash)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Options</th>
<th>Default behavior for # of threads (OMP_NUM_THREADS not set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU (gcc, g++, gfortran)</td>
<td>-fopenmp</td>
<td>as many threads as available cores</td>
</tr>
<tr>
<td>Intel (icc ifort)</td>
<td>-openmp</td>
<td>as many threads as available cores</td>
</tr>
<tr>
<td>Portland Group (pgcc,pgCC,pgf77,pgf90)</td>
<td>-mp</td>
<td>one thread</td>
</tr>
</tbody>
</table>
Running OpenMP Programs

**Code:**
Delete `num_threads(8)` in the code and set it in the environment.

**Run:**
[ahnt@hopper:~/Course/csci4850/2018S/Lectures/codes/week2]$ export OMP_NUM_THREADS=4
[ahnt@hopper:~/Course/csci4850/2018S/Lectures/codes/week2]$ .
hello_openmp
Hello from thread 0 of 4 with printf()
Hello from thread 2 of 4 with printf()
Hello from thread 3 of 4 with printf()
Hello from thread 1 of 4 with printf()
OpenMP Constructs

OpenMP’s constructs:

- Parallel Regions
- Worksharing (for/DO, sections, …)
- Data Environment (shared, private, …)
- Synchronization (barrier, flush, …)
- Runtime functions/environment variables (omp_get_num_threads(), …)
OpenMP API Overview

Three Components:

● The OpenMP API is comprised of three distinct components. As of version 4.0:
  § Compiler Directives (44)
  § Runtime Library Routines (35)
  § Environment Variables (13)

● The application developer decides how to employ these components. In the simplest case, only a few of them are needed.

● Implementations differ in their support of all API components.
Compiler Directives

- Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.

- OpenMP compiler directives are used for various purposes:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads

- For example:

<table>
<thead>
<tr>
<th>Fortran</th>
<th>!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>#pragma omp parallel default(shared) private(beta,pi)</td>
</tr>
</tbody>
</table>
Compiler Directives

- private (list), shared (list)
- firstprivate (list), lastprivate (list)
- reduction (operator: list)
- schedule (method [, chunk_size])
- nowait
- if (scalar_expression)
- num_thread (num)
- threadprivate(list), copyin (list)
- ordered
- collapse (n)
- tie, untie
- And more …
Run-time Library Routines

- These routines are used for a variety of purposes:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature
  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

- For example:

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>INTEGER FUNCTION OMP_GET_NUM_THREADS()</td>
</tr>
<tr>
<td>C/C++</td>
<td>#include &lt;omp.h&gt;</td>
</tr>
<tr>
<td></td>
<td>int omp_get_num_num_threads(void)</td>
</tr>
</tbody>
</table>
Run-time Library Routines

- Number of threads: `omp_{set,get}_num_threads`
- Thread ID: `omp_get_thread_num`
- Scheduling: `omp_{set,get}_dynamic`
- Nested parallelism: `omp_in_parallel`
- Locking: `omp_{init,set,unset}_lock`
- Active levels: `omp_get_thread_limit`
- Wallclock Timer: `omp_get_wtime`
- thread private
- call function twice, use difference between end time and start time
- And more ...
Environment Variables

- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

- These environment variables can be used to control such things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

- For example,

<table>
<thead>
<tr>
<th>csh/tcsh</th>
<th><code>setenv OMP_NUM_THREADS 8</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>sh/bash</td>
<td><code>export OMP_NUM_THREADS=8</code></td>
</tr>
</tbody>
</table>
Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_STACKSIZE
- OMP_DYNAMIC
- OMP_NESTED
- OMP_WAIT_POLICY
- OMP_ACTIVE_LEVELS
- OMP_THREAD_LIMIT
- And more …
Create threads

- Compiler Directives
  
  ```
  #pragma omp parallel num_threads(4)
  ```

- Run-time Library
  
  ```
  omp_set_num_threads(4);
  ```

- Environment Variables
  
  ```
  export OMP_NUM_THREADS=4
  ```
Count the primes

- 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.
- Check time using `omp_get_wtime()`. E.g.,
  ```
  double time1 = omp_get_wtime();
  ```
# include <cstdlib>
# include <iostream>
# include <iomanip>
# include <omp.h>

using namespace std;

int prime_static ( int n );
int prime_dynamic ( int n );

//****************************************************************************80
int main ( int argc, char *argv[] )
//****************************************************************************80
{
    int n;
    int primes;
    double time1;
    double time2;

    cout << "\n";
    cout << "SCHEDULE_OPENMP
";
    cout << "  C++/OpenMP version\n";
    cout << "  Count the primes from 1 to N.\n";
    cout << "  This is an unbalanced work load, particular for two threads.\n";
    cout << "Demonstrate static and dynamic scheduling.\n";
    cout << "\n";
    cout << "  Number of processors available = " << omp_get_num_procs ( ) << "\n";
    cout << "  Number of threads =              " << omp_get_max_threads ( ) << "\n";

    n = 100000;

    time1 = omp_get_wtime ( );
    primes = prime_static ( n );
    time1 = omp_get_wtime ( ) - time1;

    time2 = omp_get_wtime ( );
    primes = prime_dynamic ( n );
    time2 = omp_get_wtime ( ) - time2;

    cout << "\n"
    << " setw(8) << "n"
    << " setw(8) << "primes"
    << " setw(12) << "static time"
    << " setw(12) << "dynamic time" << "\n"
;
    cout << "\n"
    << " setw(8) << n"
    << " setw(8) << primes"
    << " setw(12) << time1"
    << " setw(12) << time2" << "\n";

    return 0;
}

//****************************************************************************80
int prime_static ( int n )
//****************************************************************************80
{
    int i;
    int j;
    int prime;
    int total = 0;

    #pragma omp for reduction ( + : total ) schedule ( static, 100 )
    #pragma omp parallel for reduction ( + : total )
    schedule ( static ) private ( i, j, prime )
    for ( i = 2; i < n; i++ )
    {
        prime = 1;
        for ( j = 2; j < i; j++ )
        {
            if ( i % j == 0 )
            {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}

//****************************************************************************80
int prime_dynamic ( int n )
//****************************************************************************80
{
    int i;
    int j;
    int prime;
    int total = 0;

    #pragma omp for reduction ( + : total )
    schedule ( dynamic ) private ( i, j, prime )
    for ( i = 2; i < n; i++ )
    {
        prime = 1;
        for ( j = 2; j < i; j++ )
        {
            if ( i % j == 0 )
            {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}
SCHEDULE clause

- Decide how the iterations are executed in parallel
  - schedule (static | dynamic | guided [, chunk] | runtime | auto)
  - default is static

- There is always a trade off between load balance and overhead

- Always start with static and go to more complex schemes as load balance requires.
STATIC schedule

- With no \textit{chunksize} specified
  - Iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread (\textit{block} schedule)

- If \textit{chunksize} is specified
  - Iteration space is divided into chunks, each of \textit{chunksize} iterations. The chunks are assigned cyclically to each thread (\textit{block cyclic} schedule)
DYNAMIC and GUIDED schedules

- **DYNAMIC**
  - Divides the iteration space up into chunks of size $chunksize$ and assigns them to threads on a **first-come first-served basis**.
  - i.e. as a thread finishes a chunk, it is assigned the next chunk in the list.
  - When no $chunksize$ is specified it defaults to 1.

- **GUIDED schedule**
  - Similar to DYNAMIC, but the **chunks start off large and get smaller exponentially**.
  - The size of the next chunk is (roughly) the number of remaining iterations divided by the number of threads.
  - The $chunksize$ specifies the minimum size of the chunks
  - When no $chunksize$ is specified it defaults to 1.
DYNAMIC and GUIDED schedules

\[
\text{SCHEDULE (DYNAMIC, 3)}
\]

\[
\text{SCHEDULE (GUIDED, 3)}
\]
Run Time Library & Environment Schedule

- Allows the choice of schedule to be deferred until runtime
  
  schedule (runtime)

- Set by environment variable `OMP_SCHEDULE`:
  - STATIC, no chunk size specified
  - Sets the run-time schedule type and an optional chunk size.

  
  $ export OMP_SCHEDULE="guided,4"$
AUTO schedule

- (OpenMP 3.0) gives implementation freedom to choose best mapping of iterations to threads
Choosing a schedule

- STATIC is best for balanced loops – least overhead.
- STATIC is good for loops with mild or smooth load imbalance – but can introduce “false sharing” (see later).
- DYNAMIC is useful if iterations have widely varying loads, but ruins data locality (can get cache hit).
- GUIDED is often less expensive than DYNAMIC, but beware of loops where first iterations are the most expensive!
- Use RUNTIME for convenient experimentation
Data-sharing Attributes

- All threads have access to the same, *globally shared*, memory
- Data can be shared or private
  - **Shared** data is accessible by all threads
  - **Private** data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- *Synchronization* takes place, but it is *mostly implicit*
The PRIVATE and SHARED clauses

- **Shared** - There is only one instance of the data
  - All threads can read and write the data simultaneously, unless protected through a specific OpenMP construct
  - All changes made are visible to all threads

- **Private** - Each thread has a copy of the data
  - No other thread can access this data
  - Changes only visible to the thread owning the data
Combined parallel/worksharing construct

- OpenMP shortcut: Put the “parallel” and the worksharing directive on the same line

```c
#pragma omp parallel
{
#pragma omp for
    for (ii=0; ii<N; ii++)
        a[ii] = ...;
}
```

```c
#pragma omp parallel for
for (ii=0; ii<N; ii++)
    a[ii] = ...;
```
Shared Memory programming model

Most variables (including locals) are shared by default!

```c
{  
  int sum = 0;
  #pragma omp parallel for
  for (int i=0; i<N; i++) sum += i;
}
```

Global variables are shared

Some variables can be private

- Automatic variables inside the statement block
- Automatic variables in the called functions
- Variables can be explicitly declared as private.
  In that case, a local copy is created for each thread
Overriding storage attributes

- **private:**
  - A copy of the variable is created for each thread
  - There is no connection between the original variable and the private copies
    - private variables are not initialized
  - Can achieve the same using variables inside `{ }`

- **firstprivate:**
  - Same, but the initial value of the variable is copied from the main copy
  - All variables in the list are initialized with the value the original object had before entering the parallel construct

- **lastprivate:**
  - Same, but the last value of the variable is copied to the main copy
  - The thread that executes the sequentially last iteration or section updates the value of the objects in the list
```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

//****************************************************************************80
int main ( int argc, char *argv[] )
//****************************************************************************80
{
    int i = 10;

    #pragma omp parallel firstprivate(i)
    {
        printf("thread %d: i = %d\n", omp_get_thread_num(), i);
        i = 1000 + omp_get_thread_num();
    }

    printf("i = %d\n", i);

    return 0;
}
```
```cpp
#include <iostream>
#include <stdio.h>
#include <omp.h>

using namespace std;

//****************************************************************************80
//int main ( int argc, char *argv[] )
//****************************************************************************80
{
    int i;
    int x;

    x=44;

    #pragma omp parallel for lastprivate(x)
    for(i=0;i<=10;i++){
        x=i;
        printf("Thread number: %d     x: %d\n",omp_get_thread_num(),x);
    }

    printf("x is %d\n", x);

    return 0;
}
```
What is a Data Race?

- Two different threads in a multi-threaded shared memory program
- Loosely described, a data race means that the update of a shared variable is not well protected
- Access the same (=shared) memory location
  - Asynchronously
  - Without holding any common exclusive locks
  - At least one of the accesses is a write/store
Example of a Data Race

```c
#pragma omp parallel shared(x)

{x = x + 1;}
```
How to control race conditions

- Use **synchronization** to protect data conflicts
Implicit Barrier

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

```c
#pragma omp parallel [clause...]
structured_block

#pragma omp parallel
{
    printf("Hello!\n");
} // implicit barrier
```

Hello!
Hello!
Hello!
Hello!
The if/private/shared clauses

- **if (scalar expression)**
  - Only execute in parallel if expression evaluates to true
  - Otherwise, execute serially

- **private (list)**
  - No storage association with original object
  - All references are to the local object
  - Values are undefined on entry and exit

- **shared (list)**
  - Data is accessible by all threads in the team
  - All threads access the same address space

```c
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        x[i] = y[i] + 10;
} /*-- End of parallel region --*/
```
The if clause

- Only execute in parallel if expression evaluates to true
- Otherwise, execute serially

```c
#pragma omp parallel if (n > some_threshold) \
    shared(n,x,y) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        x[i] += y[i];
} /*-- End of parallel region --*/
```
Barrier

- 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?
Barrier

- 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]
```

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

**Barrier Region**

idle

idle

idle

 Barrier syntax in OpenMP:

```plaintext
#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
{
    : 
}
```

```fortran
!$omp do 
    : 
!$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 --*/
```
SECTIONS directive

- Several blocks are executed in parallel

```c
#pragma omp parallel
{
#pragma omp sections
{
    {
        ... 
    }
#pragma omp section
{
    ... 
}
#pragma omp section
{
    ... 
}
... 

#pragma omp sections
{

}
```
Suppose we only want to see the output once or read once.

The **single** pragma directs compiler that only a single thread should execute the block of code the pragma precedes.

This construct is ideally suited for I/O or initializations

**Original Code**

```
......
"read a[0..N-1];"
......
```

```
"declare A to be be shared"
```

```
#pragma omp parallel
{
    ......
    one volunteer requested
    "read a[0..N-1];"
    thanks, we're done
    ......
}
```

May have to insert a barrier here
SINGLE directive

- Suppose we only want to see the output once or read once.
- The `single` pragma directs compiler that only a single thread should execute the block of code the pragma precedes.

```
#pragma omp single [private][firstprivate] \ [copyprivate][nowait]
{
    <code-block>
}
```

*Only one thread in the team executes the code enclosed*
MASTER directive

- 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.

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

!$omp master
    <code-block>
!$omp end master
```
Count the primes

```c
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <omp.h>
using namespace std;

int prime_static ( int n );
int prime_dynamic ( int n );

//****************************************************************************80
int main ( int argc, char *argv[] )
//****************************************************************************80
{
    int n;
    int primes;
    double time1;
    double time2;
    cout << "SCHEDULE_OPENMP"
    cout << "  C++/OpenMP version"
    cout << "  Count the primes from 1 to N."
    cout << "  This is an unbalanced work load, particular for two threads."
    cout << "  Demonstrate static and dynamic scheduling."
    cout << "Number of processors available = " << omp_get_num_procs () << "n"
    cout << "Number of threads = " << omp_get_max_threads () << "n"
    n = 100000;
    time1 = omp_get_wtime ( );
    primes = prime_static ( n );
    time1 = omp_get_wtime ( ) - time1;
    time2 = omp_get_wtime ( );
    primes = prime_dynamic ( n );
    time2 = omp_get_wtime ( ) - time2;
    cout << "n" << setw(8) << "n"
    << "n" << setw(8) << "primes"
    << "n" << setw(12) << "static time"
    << "n" << setw(12) << "dynamic time" << "n"
    cout << n << n << primes
    << n << time1 << primes
    << n << time2 << "n"
    return 0;

//****************************************************************************80
int prime_static ( int n )
//****************************************************************************80
{
    int i;
    int j;
    int prime;
    int total = 0;
    #pragma omp for reduction ( + : total ) schedule ( static, 100 )
    #pragma omp parallel for reduction ( + : total )
    schedule ( static ) private ( i, j, prime )
    for ( i = 2; i <= n; i++ )
    {
        prime = 1;
        for ( j = 2; j < i; j++ )
        {
            if ( i % j == 0 )
            {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}

//****************************************************************************80
int prime_dynamic ( int n )
//****************************************************************************80
{
    int i;
    int j;
    int prime;
    int total = 0;
    #pragma omp parallel for reduction ( + : total )
    schedule ( dynamic ) private ( i, j, prime )
    for ( i = 2; i <= n; i++ )
    {
        prime = 1;
        for ( j = 2; j < i; j++ )
        {
            if ( i % j == 0 )
            {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}
```