Lecture 2

A “Hand-on” Introduction to OpenMP

James Lin
Center for HPC, SJTU
March 6th, 2017
Outline

• Introduction to OpenMP
• Creating Threads
• Synchronization between variables
• Parallel Loops
• Synchronize single masters and stuff
Ken Kennedy (1945-2007)
OpenMP* Overview:

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.
OpenMP BasicDefs: Solution Stack

User layer

User layer

Application

End User

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc1

Proc2

Proc3

ProcN

Shared Address Space
OpenMP Release History

- **1997**: OpenMP Fortran 1.0
- **1998**: OpenMP C/C++ 1.0
- **1999**: OpenMP Fortran 1.1
- **2000**: OpenMP Fortran 2.0
- **2002**: OpenMP C/C++ 2.0
- **2005**: OpenMP 2.5
- **2008**: OpenMP 3.0
- **2011**: OpenMP 3.1
- **2013**: OpenMP 4.0
- **2015**: OpenMP 4.5

- **1998** to **2002**: A single specification for Fortran, C and C++
- **2005** to **2008**: Tasking, other new features
- **2008** to **2011**: A few more features and bug fixes
- **2011** to **2013**: Device constructs, SIMD
- **2013** to **2015**: Taskloop construct
OpenMP core syntax

• Most of the constructs in OpenMP are compiler directives.

  #pragma omp construct [clause [clause]...]

  – Example

  #pragma omp parallel num_threads(4)

• Function prototypes and types in the file:

  #include <omp.h>

• Most OpenMP* constructs apply to a “structured block”.

  – It’s OK to have an exit() within the structured block.
Exercise 1, Part A: Hello world
Verify that your environment works

- Write a program that prints “hello world”.

```c
#include <stdio.h>

int main()
{
    int ID = 0;
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
}
```
Exercise 1, Part B: Hello world

Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

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

int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

Switches for compiling and linking

- gcc -fopenmp Linus
- gcc-6 -fopenmp OSX
- pgcc -mp pgi
- icl /Qopenmp intel (windows)
- icc -fopenmp intel (linux)
Exercise 1: Solution
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

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

int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d) ", ID);
        printf(" world(%d) \n", ID);
    }
}
```

Sample Output:
- hello(1) hello(0) world(1)
- world(0)
- hello (3) hello(2) world(3)
- world(2)

OpenMP include file
Parallel region with default number of threads
Runtime library function to return a thread ID.
OpenMP Overview: How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - race condition: when the program’s outcome changes as the threads are scheduled differently.
OpenMP Concepts

• Introduction to OpenMP
• Creating Threads
• Synchronization between variables
• Parallel Loops
• Synchronize single masters and stuff
OpenMP Programming Model:

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
Thread Creation: Parallel Regions

- You create threads in OpenMP* with the parallel construct.

- For example, To create a 4 thread Parallel region:

```c
double A[1000];
#pragma omp parallel num_threads(4)
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
```

- Each thread executes a copy of the code within the structured block

- Each thread calls `pooh(ID, A)` for $ID = 0$ to 3

* The name "OpenMP" is the property of the OpenMP Architecture Review Board
Thread Creation: Parallel Regions example

- Each thread executes the same code redundantly.

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a barrier)

* The name "OpenMP" is the property of the OpenMP Architecture Review Board
Exercises 2 to 4: Numerical Integration

Mathematically, we know that:

\[ \int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi \]

We can approximate the integral as a sum of rectangles:

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
Exercises 2 to 4: Serial PI Program

```c
static long num_steps = 1e9;
double step;
int main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/num_steps;

    for (i=0; i< num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```
# Exercise 2: A simple SPMD pi program

```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
int main ()
{
    int i, nthreads; double pi, sum[NUM_THREADS];
    step = 1.0 / num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        int i, id,nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id, sum[id]=0.0;i<num_steps; i=i+nthrds) {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<nthreads;i++)pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

Only one thread should copy the number of threads to the global value to make sure multiple threads writing to the same address don’t conflict.

This is a common trick in SPMD programs to create a cyclic distribution of loop iterations.
OpenMP Concepts

- Introduction to OpenMP
- Creating Threads
- Synchronization between variables
- Parallel Loops
- Synchronize single masters and stuff
High level synchronization:
- critical
- atomic
- barrier
- ordered

Low level synchronization
- flush
- locks (both simple and nested)
Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a critical region.

```c
float res;
#pragma omp parallel
{
    float B; int i, id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for(i=id;i<niters;i+=nthrds){
        B = big_job(i);
        #pragma omp critical
            res += consume(B);
    }
}
```

Threads wait their turn – only one at a time calls consume()
Synchronization: Atomic

- **Atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```c
#pragma omp parallel
{
    double tmp, B;
    B = DOIT();
    tmp = big_ugly(B);
    #pragma omp atomic
    X += tmp;
}
```

Atomic only protects the read/update of X
OpenMP Concepts

• Introduction to OpenMP
• Creating Threads
• Synchronization between variables
• Parallel Loops
• Synchronize single masters and stuff
A parallel construct by itself creates an SPMD or “Single Program Multiple Data” program … i.e., each thread redundantly executes the same code.

How do you split up pathways through the code between threads within a team?

This is called worksharing

- Loop construct
- Sections/section constructs
- Single construct
- Task construct

Discussed later
The loop worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team.

#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++){
       NEAT_STUFF(I);
   }
}

Loop construct name:
- C/C++: for
- Fortran: do

The variable I is made “private” to each thread by default. You could do this explicitly with a “private(I)” clause.
Loop worksharing Constructs
A motivating example

Sequential code

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

#pragma omp parallel
```
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1) iend = N;
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}
}
```

#pragma omp parallel for
```
for(i=0;i<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP parallel region

OpenMP parallel region and a worksharing for construct
Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```c
int i, j, A[MAX];
j = 5;
for (i=0;i< MAX; i++) {
    j +=2;
    A[i] = big(j);
}
```

Note: loop index “i” is private by default

```c
int i, A[MAX];
#pragma omp parallel for
for (i=0;i< MAX; i++) {
    int j = 5 + 2*(i+1);
    A[i] = big(j);
}
```

Remove loop carried dependence
## loop work-sharing constructs: The schedule clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
<tr>
<td>AUTO</td>
<td>When the runtime can “learn” from previous executions of the same loop</td>
</tr>
</tbody>
</table>

- **Least work at runtime**: scheduling done at compile-time
- **Most work at runtime**: complex scheduling logic used at run-time
Loop Reduction

How do we handle this case?

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can’t be trivially removed.
- This is a very common situation … it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

```c
double ave=0.0, A[MAX]; int i;
for (i=0; i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```
Loop Reduction

- OpenMP reduction clause:
  
  ```
  reduction (op : list)
  ```

- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

- The variables in “list” must be shared in the enclosing parallel region.

```c
double ave=0.0, A[MAX]; int i;
#pragma omp parallel for reduction (+:ave)
for (i=0; i< MAX; i++) {
    ave += A[i];
}
ave = ave/MAX;
```
OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>min</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>max</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>
OpenMP Concepts

- Introduction to OpenMP
- Creating Threads
- Synchronization between variables
- Parallel Loops
- Synchronize single masters and stuff
**Synchronization: Barrier**

- **Barrier**: Each thread waits until all threads arrive.

```c
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp barrier
    #pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(i,A);}
    #pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
    A[id] = big_calc4(id);
}
```

- **Implicit barrier at the end of a parallel region**
- **Implicit barrier at the end of a for worksharing construct**
- **No implicit barrier due to nowait**
Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```c
#pragma omp parallel
{
    do_many_things();
#pragma omp master
    { exchange_boundaries(); }
#pragma omp barrier
    do_many_other_things();
}
```
Single worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a `nowait` clause).

```c
#pragma omp parallel
{
    do_many_things();
# pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```
Sections worksharing Construct

- The *Sections* worksharing construct gives a different structured block to each thread.

```cpp
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        X_calculation();
        #pragma omp section
        y_calculation();
        #pragma omp section
        z_calculation();
    }
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.
Books about OpenMP

A new book about OpenMP 2.5 by a team of authors at the forefront of OpenMP’s evolution.

A book about how to “think parallel” with examples in OpenMP, MPI and java.

OpenMP Tutorials from LLNL

https://computing.llnl.gov/tutorials/openMP