OpenMP Multithreaded Programming

- OpenMP stands for “Open Multi-Processing”
- OpenMP is a multi-vendor (see next page) standard to perform shared-memory multithreading
- OpenMP uses the fork-join model
- OpenMP is both directive- and library-based
- OpenMP threads share a single executable, global memory, and heap (malloc, new)
- Each OpenMP thread has its own stack (function arguments, function return address, local variables)
- Using OpenMP requires no dramatic code changes
- OpenMP probably gives you the biggest multithread benefit per amount of work you have to put in to using it

Much of your use of OpenMP will be accomplished by issuing C/C++ “pragmas” to tell the compiler how to build the threads into the executable

```
#pragma omp directive [clause]
```

Who is in the OpenMP Consortium?

- AMD
- ARM
- Fujitsu
- Intel
- NEC
- Oracle
- Red Hat
- IBM
- EPCC
- University of Illinois at Urbana-Champaign
- University of Minnesota
- University of Texas at Austin
- UC Santa Barbara
- University of Wisconsin-Madison
- University of Washington
- University of Texas at Austin
- Berkeley Computer Associates
- Argonne National Laboratory
- Craftsman
- Compaq

What OpenMP Isn’t:

- OpenMP doesn’t check for data dependencies, data conflicts, deadlocks, or race conditions. You are responsible for avoiding those yourself
- OpenMP doesn’t check for non-conforming code sequences
- OpenMP doesn’t guarantee identical behavior across vendors or hardware, or even between multiple runs on the same vendor’s hardware
- OpenMP doesn’t guarantee the order in which threads execute, just that they do execute
- OpenMP is not overhead-free
- OpenMP does not prevent you from writing code that triggers cache performance problems (such as in false-sharing), in fact, it makes it really easy

We will get to “false sharing” in the cache notes
Memory Allocation in a Multithreaded Program

One-thread

Multiple-threads

Program Executable

Stack

Stack

Globals

Common Program Executable

Common Globals

Heap

Common Heap

Don’t take this completely literally. The exact arrangement depends on the operating system and the compiler. For example, sometimes the stack and heap are arranged so that they grow towards each other.

Number of OpenMP threads

Two ways to specify how many OpenMP threads you want to have available:

1. Set the OMP_NUM_THREADS environment variable
2. Call omp_set_num_threads( num );

Asking how many cores this program has access to:

num = omp_get_num_procs( );

Setting the number of threads to the exact number of cores available:

num = omp_set_num_threads( omp_get_num_procs( ) );

Asking how many OpenMP threads this program is using right now:

num = omp_get_num_threads( );

Asking which thread this one is:

me = omp_get_thread_num( );

Creating an OpenMP Team of Threads

#pragma omp parallel default(none)
{

...
}

This creates a team of threads. Each thread then executes all lines of code in this block.

Think of it this way:

#pragma omp parallel default(none)
Creating an OpenMP Team of Threads

```c
#include <stdio.h>
#include <omp.h>
int main( )
{
  omp_set_num_threads( 8 );
  #pragma omp parallel default(none)
  {
    printf( "Hello, World, from thread #%d ! \n" , omp_get_thread_num( )  );
  }
  return 0;
}
```

Hint: run it several times in a row. What do you see? Why?

There is no guarantee of thread execution order!

Creating OpenMP threads in Loops

```c
#include <omp.h>
omp_set_num_threads( NUMT );
#pragma omp parallel default(none)
for( int i = 0; i < arraySize; i++ )
{
  ... 
  This tells the compiler to parallelize the for-loop into multiple threads. Each thread automatically gets its own personal copy of the variable i because it is defined within the for-loop body.
}
```

OpenMP for-Loop Rules

```
#pragma omp parallel for default(none), shared(...), private(…)
for( int index = start ; index terminate condition; index changed )
```

- The index must be an int or a pointer
- The start and terminate conditions must have compatible types
- Neither the start nor the terminate conditions can be changed during the execution of the loop
- The index can only be modified by the changed expression (i.e., not modified inside the loop itself)
- There can be no inter-loop data dependencies such as:
  - `a[ i ] = a[ i-1 ] + 1 ;`
  - because what if these two lines end up being given to two different threads
  - `a[101] = a[100] + 1 ;  a[102] = a[101] + 1 ;`
OpenMP For-Loop Rules

```c
for( index = start; index < end; index <= end; index > end; index >= end )
```

- `index++`
- `++index`
- `index--`
- `--index`
- `index += incr`
- `index = index + incr`
- `index -= decr`
- `index = index - decr`

OpenMP Directive Data Types

I recommend that you use:

- `default(none)`

in all your OpenMP directives. This will force you to explicitly flag all of your inside variables as shared or private. This will help prevent mistakes.

- `private(x)`

Means that each thread will have its own copy of the variable `x`

- `shared(x)`

Means that all threads will share a common `x`. This is potentially dangerous.

Example:

```c
#pragma omp parallel for default(none),private(i,j),shared(x)
```

OpenMP For-Loop Rules

`for( index = start; index < end; index <= end; index > end; index >= end )`

- `index++`
- `++index`
- `index--`
- `--index`
- `index += incr`
- `index = index + incr`
- `index -= decr`
- `index = index - decr`

OpenMP Directive Data Types

I recommend that you use:

- `default(none)`

in all your OpenMP directives. This will force you to explicitly flag all of your inside variables as shared or private. This will help prevent mistakes.

- `private(x)`

Means that each thread will have its own copy of the variable `x`

- `shared(x)`

Means that all threads will share a common `x`. This is potentially dangerous.

Example:

```c
#pragma omp parallel for default(none),private(i,j),shared(x)
```

OpenMP Single Program Multiple Data (SPMD) in OpenMP

```c
#define NUM 1000000
float A[NUM], B[NUM], C[NUM];
...
total =omp_get_num_threads( );
#pragma omp parallel default(none),private(me),shared(total)
{    
    me = omp_get_thread_num( );
    DoWork( me, total );
}
void  DoWork( int me, int total )
{    
    int first = NUM * me / total;
    int last = NUM * (me+1)/total - 1;
    for( int i = first; i <= last; i++)
    {        
        C[i] = A[i] * B[i];
    }
}
```

OpenMP Allocation of Work to Threads

**Static Threads**
- All work is allocated and assigned at runtime

**Dynamic Threads**
- Consists of one Master and a pool of threads
  - The pool is assigned some of the work at runtime, but not all of it
  - When a thread from the pool becomes idle, the Master gives it a new assignment
  - “Round-robin assignments”

**OpenMP Scheduling**
- `schedule(static [,chunksize])`
- `schedule(dynamic [,chunksize])`
- Defaults to static
- chunksize defaults to 1
  - In static, the iterations are assigned to threads before the loop starts
OpenMP Allocation of Work to Threads

```c
#pragma omp parallel for default(none),schedule(static,chunksize)
for( int index = 0 ; index < 12 ; index++ )
```

Static, 1
0 0,3,6,9
1 1,4,7,10
2 2,5,8,11

chunksize = 1
Each thread is assigned one iteration, then
the assignments start over

Static, 2
0 0,1,6,7
1 2,3,8,9
2 4,5,10,11

chunksize = 2
Each thread is assigned two iterations, then
the assignments start over

Static, 4
0 0,1,2,3
1 4,5,6,7
2 8,9,10,11

chunksize = 4
Each thread is assigned four iterations, then
the assignments start over

Arithmetic Operations Among Threads – A Problem

```c
#pragma omp parallel for private(myPartialSum),shared(sum)
for( int i = 0; i < N; i++ )
{
    float myPartialSum = ...
    sum = sum + myPartialSum;
}
```

• There is no guarantee when each thread will execute this line correctly
• There is not even a guarantee that each thread will finish this line before some
  other thread interrupts it. (Remember that each line of code usually generates
  multiple lines of assembly.)
• This is non-deterministic!

Here’s a trapezoid integration example (covered in another note set).

The partial sums are added up, as shown on the previous page.
The integration was done 30 times.
None of the 30 answers is even close.
And, not only are the answers bad, they are not even consistently bad!

<table>
<thead>
<tr>
<th>Trial #</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.469635</td>
</tr>
<tr>
<td>2</td>
<td>0.517984</td>
</tr>
<tr>
<td>3</td>
<td>0.438868</td>
</tr>
<tr>
<td>4</td>
<td>0.437553</td>
</tr>
<tr>
<td>5</td>
<td>0.398761</td>
</tr>
<tr>
<td>6</td>
<td>0.506564</td>
</tr>
<tr>
<td>7</td>
<td>0.482211</td>
</tr>
<tr>
<td>8</td>
<td>0.506362</td>
</tr>
<tr>
<td>9</td>
<td>0.504610</td>
</tr>
<tr>
<td>10</td>
<td>0.486226</td>
</tr>
<tr>
<td>11</td>
<td>0.476870</td>
</tr>
<tr>
<td>12</td>
<td>0.434737</td>
</tr>
<tr>
<td>13</td>
<td>0.530668</td>
</tr>
<tr>
<td>14</td>
<td>0.444919</td>
</tr>
<tr>
<td>15</td>
<td>0.500062</td>
</tr>
<tr>
<td>16</td>
<td>0.442432</td>
</tr>
<tr>
<td>17</td>
<td>0.672593</td>
</tr>
<tr>
<td>18</td>
<td>0.548837</td>
</tr>
<tr>
<td>19</td>
<td>0.334996</td>
</tr>
<tr>
<td>20</td>
<td>0.484124</td>
</tr>
<tr>
<td>21</td>
<td>0.489211</td>
</tr>
<tr>
<td>22</td>
<td>0.506362</td>
</tr>
<tr>
<td>23</td>
<td>0.504610</td>
</tr>
<tr>
<td>24</td>
<td>0.486226</td>
</tr>
<tr>
<td>25</td>
<td>0.476870</td>
</tr>
<tr>
<td>26</td>
<td>0.434737</td>
</tr>
<tr>
<td>27</td>
<td>0.530668</td>
</tr>
<tr>
<td>28</td>
<td>0.444919</td>
</tr>
<tr>
<td>29</td>
<td>0.500062</td>
</tr>
<tr>
<td>30</td>
<td>0.442432</td>
</tr>
</tbody>
</table>

Conclusion: Don’t do it this way!
Arithmetic Operations Among Threads – Three Solutions

1.
#pragma omp atomic  
sum = sum + myPartialSum;  
• Fixes the non-deterministic problem  
• But, serializes the code  
• Operators include +, -, *, /, ++, --, >>, <<, ^, |, etc.

2.  
#pragma omp critical  
sum = sum + myPartialSum;  
• Also fixes it  
• But, serializes the code

3.  
#pragma omp parallel for reduction(+:sum),private(myPartialSum)  
• Performs (sum, product, and, or, ...) in $O(\log_2 N)$ time instead of $O(N)$  
• Operators include +, -, *, /, ++, --  
• Operators include +=, -=, *=, /=, etc.  
• Operators include ^=, |=, &=

Reduction vs. Atomic vs. Critical

Why Not Do Reduction by Creating Your Own sums Array, one for each Thread?

float *sums = new float [omp_get_num_threads()];
for (int i = 0; i < omp_get_num_threads(); i++)
    sums[i] = 0.0;

#pragma omp parallel for private(myPartialSum),shared(sums)
for (int i = 0; i < N; i++)
    {
        myPartialSum = ...  
        sums[omp_get_thread_num()] += myPartialSum;
    }

float sum = 0.0;
for (int i = 0; i < omp_get_num_threads(); i++)
    sum += sums[i];

delete [] sums;

• This seems perfectly reasonable, it works, and it gets rid of the problem of multiple threads trying to write into the same reduction variable.

• The reason we don’t do this is that this method provokes a problem called False Sharing. We will get to that when we discuss caching.
Synchronization

Mutual Exclusion Locks (Mutexes)

- omp_init_lock( omp_lock_t *);
- omp_set_lock(   omp_lock_t *);
- omp_unset_lock( omp_lock_t *);
- omp_test_lock( omp_lock_t *);

Note: omp_lock_t is really an array of 4 unsigned chars

Critical sections

- #pragma omp critical
  - Restricts execution to one thread at a time
- #pragma omp single
  - Restricts execution to a single thread ever

Barriers

- #pragma omp barrier
  - Forces each thread to wait here until all threads arrive

(Note: there is an implied barrier after parallel for loops and OpenMP sections, unless the nowait clause is used)

Synchronization Examples

```
omp_lock_t Sync;
...
omp_init_lock( &Sync );
...
omp_set_lock( &Sync );
    // code that needs the mutual exclusion
omp_unset_lock( &Sync );
...
while( omp_test_lock( &Sync ) == 0 )
{
    DoSomeUsefulWork( );
}
```

Creating Sections of OpenMP Code

```
#pragma omp parallel sections
{
    #pragma omp section
    { Task 1 }
    #pragma omp section
    { Task 2 }
}
```

Sections are independent blocks of code, able to be assigned to separate threads if they are available.

What do OpenMP Sections do for You?

- omp_set_num_threads( 1 );
- omp_set_num_threads( 2 );
- omp_set_num_threads( 3 );

There is an implied barrier at the end
OpenMP Tasks

- An OpenMP task is a single line of code or a structured block which is immediately assigned to one thread in the current thread team.
- The task can be executed immediately, or it can be placed on its thread’s list of things to do.
- If the if clause is used and the argument evaluates to 0, then the task is executed immediately, superseding whatever else that thread is doing.
- There has to be an existing parallel thread team for this to work. Otherwise, one thread ends up doing all tasks.
- One of the best uses of this is to make a function call. That function then runs concurrently until it completes.

```c
#pragma omp task
    Watch_For_Internet_Input();
```

You can create a task barrier with:

```c
#pragma omp taskwait
```

Tasks are very much like OpenMP Sections, but Sections are more static, that is, the number of sections is set when you write the code, whereas Tasks can be created anytime, and in any number, under control of your program’s logic.

OpenMP Task Example: Processing each element of a linked list

```c
#pragma omp parallel
{

    #pragma omp single default(none)
    { 
        element *p = listHead;
        while( p != NULL )
        {
            #pragma omp task private(p)
            Process( p );
            p = p->next;
        }
    }
}
```