Parallel Programming using OpenMP

Mike Bailey
mjb@cs.oregonstate.edu

Oregon State University

OpenMP Multithreaded Programming

- OpenMP stands for "Open Multi-Processing"
- OpenMP is a multi-vendor* 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, 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

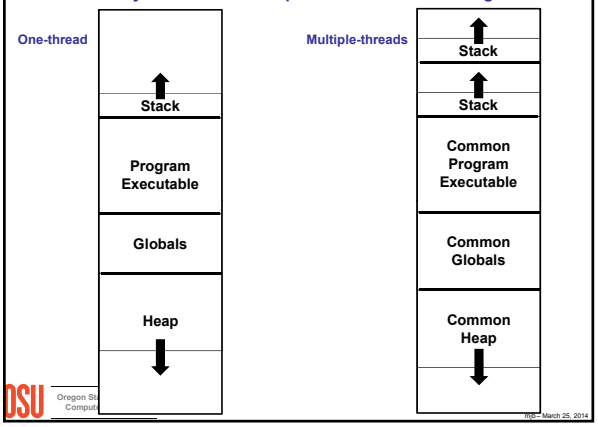
```
#pragma omp directive [clause]
```

* **AMD, Fujitsu, HP, IBM, Intel, Microsoft, NEC, NVIDIA, Oracle, Texas Instruments, VMWare, …**

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
- 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 false-sharing code (in fact, it makes it really easy)

Memory Allocation in an OpenMP Multithreaded Program



Using OpenMP in Linux

g++ -o program program.cpp -lm **-fopenmp**

Using OpenMP in Microsoft Visual Studio

1. Go to the Project menu ⟶ Project Properties
2. Change the setting Configuration Properties ⟶ C/C++ ⟶ Language ⟶ OpenMP Support to **"Yes (/openmp)"**

Seeing if OpenMP is Supported on Your System

```
#ifndef _OPENMP
fprintf( stderr, "OpenMP is not supported – sorry!\n" );
#endif
```


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( );
```

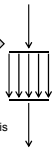
Asking which thread this one is:

```
me = omp_get_thread_num( );
```

Creating an OpenMP Team of Threads

```
#pragma omp parallel default(none)
{
        . . .
}
```

This creates a team of threads



Each thread would then execute all lines of code in this block.

Try this, just for fun:

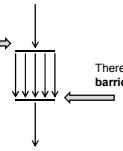
```
omp_set_num_threads( 4 );
#pragma omp parallel default(none)
printf( "Hello, World, from thread #%d ! \n" , omp_get_thread_num( ) );
```

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

Creating OpenMP threads in Loops

```
#include <omp.h>
int i;
#pragma omp parallel for default(none),private(i)
for( i = 0; i < num; i++ )
{
        . . .
}
```

This creates a team of threads



There is an **"implied barrier"** at the end

This tells the compiler to parallelize the for loop into multiple threads, and to give each thread its own personal copy of the variable *i*. But, you don't have to do this for variables defined in the loop body:

```
#pragma omp parallel for default(none)
for( int i = 0; i < num; i++ )
{
        . . .
}
```


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 *end* conditions must have compatible types
- Neither the *start* nor the *end* 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 between-loop data dependencies

This is the probably the biggest parallel benefit per programming effort !

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 = incr + index / 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 each thread will share a common x. This is potentially dangerous.

Example:

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

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

#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

```
#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

• There is even no guarantee that each thread will finish this line before some other thread interrupts it

• This is non-deterministic !

Don't do it this way!

Here's a trapezoid integration example (covered in another section).
The partial sums are added up, as shown on the previous page.
The integration was done 30 times.
The answer is supposed to be exactly 2.
None of the 30 answers is even close.
Nand, nt only are the answers bad, they are not even consistently bad!

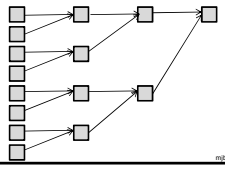
Don't do it this way!

Arithmetic Operations Among Threads – Three Solutions

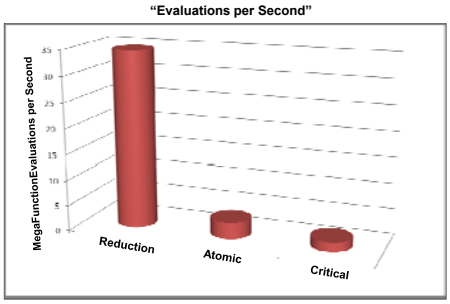
#pragma omp atomic
sum = sum + myPartialSum;
- Fixes the non-deterministic problem
- But, serializes the code
- Operators include +, -, *, /, ++, --, >>, <<, ^, |
- Operators include +=, -=, *=, /=, etc.

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

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



Reduction vs. Atomic vs. Critical



Synchronization

Mutual Exclusion Locks (Mutexes)

*omp_init_lock(omp_lock_t *);*
*omp_set_lock(omp_lock_t *);* ← Blocks if the lock is not available. Then sets it and returns when it is available
*omp_unset_lock(omp_lock_t *);*
*omp_test_lock(omp_lock_t *);* ← If the lock is not available, returns 0. If the lock is available, sets it and returns !0

(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 all threads 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

Sections are consecutive, independent blocks of code

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

Each section is executed by *some* thread (not necessarily its *own* thread)

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 forked off into one thread *in the current thread team*
- There has to be an existing parallel thread team for this to be effective
- One of the best uses of this is to make a function call.
- That function then runs concurrently until it completes.

```
#pragma omp task
    WatchForInternetInput( );
```

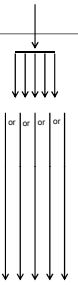
You can create a task barrier with:

#pragma omp taskwait

These are very much like OpenMP Sections, but Sections are more static, that is, they are setup when you write the code, whereas Tasks can be created anytime under control of your program's logic.

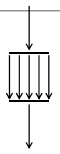
OpenMP Task Example: Processing each element of a linked list

```
#pragma omp parallel
{
    #pragma omp single default(none)
    {
        element *p = listHead;
        while( p != NULL )
        {
            #pragma omp task
            Process( p );

            p = p->next;
        }
    }
}
```



Single Program Multiple Data (SPMD) in OpenMP

```
#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 );
#pragma omp end parallel
```



```
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];
        }
}
```