Parallel Programming using OpenMP

Oregon State University

Mike Bailey
mjb@cs.oregonstate.edu

OpenMP Multithreaded Programming

- OpenMP stands for “Open Multi-Processing”
- OpenMP is a multi-vendor (see next page) standard to perform shared-memory multitreading
- 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
- ARM Holdings
- Intel
- NEC
- Fujitsu
- HP
- National Semiconductor
- Sun Microsystems
- Tandem Computers
- Wang Laboratories
- Silicon Graphics
- VMEbus
- NEC
- Hitachi

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

Memory Allocation in a Multithreaded Program

- 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.

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)"

Using OpenMP on Linux

g++ -o proj proj.cpp -lm -fopenmp
icpc -o proj proj.cpp -lm -openmp -align -qopt-report=3 -qopt-report-phase=vec

Seeing if OpenMP is Supported on Your System

#include "omp.h"

If OPENMP is defined, the following should compile:

if (omp_get_num_procs() > 0) {                  
    printf(“Number of processors is %d\n”,
            omp_get_num_procs() );
} else {                                        
    printf(“OpenMP is not supported – sorry for inconvenience!\n”);
    exit(1);
}

endif
If you are using Visual Studio 2019 and get a compile message that looks like this:
1>c1xx: error C2338: two-phase name lookup is not supported for C++/CLI, C++/CX, or OpenMP. use /Zc:twoPhase-
then do this:
1. Go to "Project Properties" → "C/C++" → "Command Line"
2. Add /Zc:twoPhase- in "Additional Options" in the bottom section
3. Press OK

No, I don't know what this means either …

Creating an OpenMP Team of Threads

#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?

Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #7 !
Hello, World, from thread #5 !
Hello, World, from thread #4 !
Hello, World, from thread #3 !
Hello, World, from thread #2 !
Hello, World, from thread #0 !

There is no guarantee of thread execution order!

Creating OpenMP threads in Loops

The code starts out executing in a single thread
The code asks how many threads will be in the thread pool. It doesn't create them yet. It just asks how many will be used the next time you ask for them.

#pragma omp parallel for default(none)
for( int i = 0; i < arraySize; i++ )
{
    // Do something
}

There is an "implied barrier" at the end where each thread waits until all threads are done, then the code continues in a single thread
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.

The default(none) directive forces you to explicitly declare all variables declared outside the parallel region to be either private or shared while they are in the parallel region. Variables declared within the for-loop are automatically private.
OpenMP for-Loop Rules

• 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)
• You cannot use a break or a goto to get out of the loop
• There can be no inter-loop data dependencies such as:
  
  ```c
  a[ i ] = a[ i-1 ] + 1.;
  ```

```c
#pragma omp parallel for default(none), shared(…), private(…)
for( int index = start ; index <   end
index <= end
index >   end
index >= end
; )
```

```c
a[101] = a[100] + 1.;    // what if this is the last line of thread #0's work?
a[102] = a[101] + 1.; // what if this is the first line of thread #1's work?
```

OpenMP For-Loop Rules

• For-loop "Fission"

Because of the loop dependency, this whole thing is not parallelizable:

```c
x[ 0 ] = 0.;
y[ 0 ] *= 2.;
for( int i = 1; i < N; i++ )
{
x[ i ] = x[ i-1 ] + 1.;
y[ i ] *= 2.;
}
```

But, it can be broken into one loop that is not parallelizable, plus one that is:

```c
x[ 0 ] = 0.;
for( int i = 1; i < N; i++ )
{
x[ i ] = x[ i-1 ] + 1.;
}
#pragma omp parallel for shared(y)
for( int i = 0; i < N; i++ )
{
y[ i ] *= 2.;
}
```

For-loop "Collapsing"

Uh-oh, which for-loop do you put the #pragma on?

```c
for( int i = 1; i < N; i++ )
{
  for( int j = 0; j < M; j++ )
  {
    ...  
  }
} 
```

Ah-ha – trick question. You put it on both!

```c
#pragma omp parallel for collapse(2)
for( int i = 1; i < N; i++ )
{
  for( int j = 0; j < M; j++ )
  {
    ...  
  }
} 
```

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 ];
  }
```
Static Threads
- All work is allocated and assigned at runtime

Dynamic Threads
- The pool is statically assigned some of the work at runtime, but not all of it
- "Round-robin assignments"

OpenMP Scheduling
schedule(static [,chunksize])
schedule(dynamic [,chunksize])
Defaults to static
chunksize defaults to 1

OpenMP Allocation of Work to Threads

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

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

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

- 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 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!

Assembly code:
Add myPartialSum
Store sum

Conclusion: Don’t do it this way!

Here’s a trapezoid integration example.
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.
And, not only are the answers bad, they are not even consistently bad!

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 * );
(omp_lock_t is really an array of 4 unsigned chars)
```

```
Critical sections
#pragma omp critical
    Restricted execution to one thread at a time
#pragma omp single
    Restricted 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

```c
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();
}
```

Single-thread-execution Synchronization

```c
#pragma omp single
Restricts execution to a single thread ever. This is used when an operation only
makes sense for one thread to do. Reading data from a file is a good example.
```

Creating Sections of OpenMP Code

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

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

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

What do OpenMP Sections do for You?
They decrease your overall execution time.

```c
omp_set_num_threads(1);
omp_set_num_threads(2);
omp_set_num_threads(3);
```

A Functional Decomposition Sections Example

```c
omp_set_num_threads(3);
#pragma omp parallel sections
{
  #pragma omp section
  {
    Watcher();
  }
  #pragma omp section
  {
    Animals();
  }
  #pragma omp section
  {
    Plants();
  }
  // implied barrier – all functions must return to get past here
```