Parallel Programming using OpenMP

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 multithreading
- OpenMP is both compiler-directive- and library-based
- OpenMP threads share a single-executable, a single global memory, and a single heap (malloc, new)
- Each OpenMP thread has its own stack (function arguments, function return address, local variables)
- Using OpenMP usually 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 your executable, like this:

#pragma omp directive [clause]

Who is in the OpenMP Consortium?

Much of your use of OpenMP will be accomplished by issuing C/C++ "pragmas" to tell the compiler how to build the threads into your executable, like this:

#pragma omp directive [clause]

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 (we’ll talk about what this means)
- 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

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 on Linux

g++ -o proj proj.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)"

If you are using Visual Studio and get a compile message that looks like this:

Click here for a note on two-phase name lookup, not supported for C++ (C:O:\CC, or OpenMP: see [2: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
#ifdef _OPENMP
fprintf(stderr, "OpenMP release %d is supported here\n", _OPENMP);
#else
fprintf(stderr, "OpenMP is not supported here – sorry!\n\n\nreturn 1;
#endif

## Seeing if OpenMP is Supported on Your System

Printing `_OPENMP` gives you a year and month of the OpenMP release that you are using.

To get the OpenMP version number from the year and month, check here:

- OpenMP 5.0 – November 2018
- OpenMP 4.5 – November 2015
- OpenMP 4.0 – July 2013
- OpenMP 3.1 – July 2011
- OpenMP 2.0 – March 2002
- OpenMP 1.0 – October 1998

• By default, R1 uses g++ 11.4, which uses OpenMP version 4.5
• Visual Studio 2022 uses OpenMP 2.0

### Numbers of OpenMP threads

How to specify how many OpenMP threads you want to have available:

```
omp_set_num_threads( num );
```

Asking how many cores this program has access to:

```
num = omp_get_num_procs();  // Actually returns the number of hyperthreads, not the number of physical cores
```

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

```
omp_set_num_threads( omp_get_num_procs() );
```

Asking which thread number this one is:

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

### Creating an OpenMP Team of Threads

```
#include <omp.h>
#pragma omp parallel default(none)
{ .... }  // This creates a team of threads
```

Think of it this way:

```
#pragma omp parallel default(none)
```

```
   omp_set_num_threads( num );
   #pragma omp parallel default(none)
   for( int i = 0; i < arraySize; i++ )
   { .... }  // Here we ask for them. This creates a team of threads and divides the for-loop passes up among those threads
```

```
   #pragma omp parallel for default(none)
   for( int i = 0; i < arraySize; i++ )
   { .... }  // This sets how many threads will be created. It doesn't create them yet; it just says how many will be used the next time you ask for them.
```

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

```
   return 0;
```

```
   #pragma omp parallel default(none)
```

```
   #pragma omp parallel for default(none)
```

There is no guarantee of thread execution order!

### Creating OpenMP Threads to Process Loop Passes

The code starts executing in a single thread.

```
omp_set_num_threads( NUMT );
```

This sets how many threads will be created.

```
#pragma omp parallel for default(none)
```

Then the compiler will 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

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

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

What to do about Variables Declared Before the for-loop Starts?

private(x)
Means that each thread will get its own version of the variable.

shared(x)
Means that all threads will share a common version of the variable.

default(none)
I recommend that you include this in your OpenMP for-loop directive. This will force you to explicitly flag all of your externally-declared variables as shared or private. Don’t make a mistake by leaving it up to the default!

Example:
#pragma omp parallel for default(none), private(x)

For-loop “Fission”

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

x[ 0 ] = 0.;
y[ 0 ] = 2.;
for( int i = 1; i < N; i++ )
  { 
    x[ i ] = x[ i-1 ] + 1.;
    y[ i ] = y[ i-1 ] + 2.;
  } // what if this is the last line of thread #0’s work?

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

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 ] = y[ i-1 ] + 2.;

Single Program Multiple Data (SPMD) in OpenMP

#define NUM 1000000
float A[NUM], B[NUM], C[NUM];
...
int total = omp_get_num_threads(  );
#pragma omp parallel default(none),shared(total)
{ 
  int me = omp_get_thread_num(  );
  DoWork( me, total );
}
void  DoWork( int m, int t )
{ 
  int first = NUM * m / t;
  int last = ( NUM * (m+1)/t ) - 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**
- The pool is statically assigned some of the work at runtime, but not all of it
- When a thread from the pool becomes idle, it gets a new assignment
- “Round-robin assignments”

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

---

**Arithmetic Operations Among Threads – A Problem**

```c
float sum = 0.;
#pragma omp parallel for default(none), shared(sum)
for( int i = 0; i < N; i++ )
{
    float myPartialSum = ...  
    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!

---

**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`
- 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

---

**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

---

**OpenMP Allocation of Work to Threads**

```
#pragma omp parallel for default(none),schedule(static,chunksize)
```

<table>
<thead>
<tr>
<th>Thread</th>
<th>chunksize</th>
<th>Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,6,9</td>
<td>1,4,7,10</td>
</tr>
<tr>
<td>2</td>
<td>2,5,8,11</td>
<td></td>
</tr>
</tbody>
</table>

---

**Synchronization Example**

```
omp_lock_t Sync;
omp_init_lock( &Sync );
```

**Thread 0:**
```
omp_set_lock( &Sync );<br>
<< code that needs the mutual exclusion >>
omp_unset_lock( &Sync );
```

**Thread 1:**
```
omp_set_lock( &Sync );<br>
<< code that needs the mutual exclusion >>
omp_unset_lock( &Sync );
```
omp_lock_t Sync;
omp_init_lock(&Sync);
...
while(omp_test_lock(&Sync) == 0)
{
    DoSomeUsefulWork_0();
}
while(omp_test_lock(&Sync) == 0)
{
    DoSomeUsefulWork_1();
}

#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.

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

#pragma omp parallel sections
{
#pragma omp section
{
    Watcher();
}
#pragma omp section
{
    Animal();
}
#pragma omp section
{
    Plants();
}
} // implied barrier -- all functions must return to get past here

If you print to standard error (stderr) from inside a for-loop, like I do, then you think that you need to include stderr in the shared list because, well, the loops share it:

#pragma omp parallel for default(none) shared(a,b,stderr)

This turns out to be true for g++/gcc only.

If you are using Visual Studio, then do not include stderr in the list. If you do, you will get this error:

1>Y:\CS575\SQ22\RobertW-01\Project1\Project1.cpp(113,98): error C2059: syntax error: '('

A Potential OpenMP/Visual Studio Compiler Problem

This is because:

• In g++/gcc, stderr is a variable
• In Visual Studio, stderr is a defined macro