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 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

```c
#pragma omp directive [clause]
```
Who is in the OpenMP Consortium?

[Logos of various companies and organizations associated with the OpenMP Consortium]
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.
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
icpc -o proj proj.cpp  -lm  -openmp  -align  -qopt-report=3 -qopt-report-phase=vec
```

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" );
    exit( 0 );
#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:

```c
num = omp_get_num_procs();
```

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

```c
num = omp_set_num_threads(omp_get_num_procs());
```

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

```c
num = omp_get_num_threads();
```

Asking which thread this one is:

```c
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:
#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 for 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.

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.

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 sets how many threads will be in the thread pool. It doesn’t create them yet, it just says how many will be used the next time you ask for them.

This creates a team of threads from the thread pool and divides the for-loop passes up among those threads.

The code starts out executing in a single thread.
OpenMP for-Loop Rules

```c
#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:
  ```c
  a[ i ] = a[ i-1 ] + 1.;
  ```
  because what if these two lines end up being given to two different threads:
  ```c
  a[101] = a[100] + 1.;
  a[102] = a[101] + 1.;
  ```
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 all threads will share a common x. This is potentially dangerous.

Example:

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

Chunksize = 1

<table>
<thead>
<tr>
<th>Static,1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,3,6,9</td>
<td>chunksize = 1</td>
</tr>
<tr>
<td>1</td>
<td>1,4,7,10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2,5,8,11</td>
<td></td>
</tr>
</tbody>
</table>

Each thread is assigned one iteration, then the assignments start over.

Chunksize = 2

<table>
<thead>
<tr>
<th>Static,2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,6,7</td>
<td>chunksize = 2</td>
</tr>
<tr>
<td>1</td>
<td>2,3,8,9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4,5,10,11</td>
<td></td>
</tr>
</tbody>
</table>

Each thread is assigned two iterations, then the assignments start over.

Chunksize = 4

<table>
<thead>
<tr>
<th>Static,4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,2,3</td>
<td>chunksize = 4</td>
</tr>
<tr>
<td>1</td>
<td>4,5,6,7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8,9,10,11</td>
<td></td>
</tr>
</tbody>
</table>

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!

**Assembly code:**

<table>
<thead>
<tr>
<th>Load sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add myPartialSum</td>
</tr>
<tr>
<td>Store sum</td>
</tr>
</tbody>
</table>

What if the scheduler decides to switch threads right here?

**Conclusion: Don’t do it this way!**
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. 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**!

<table>
<thead>
<tr>
<th>Answer 1</th>
<th>Answer 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.469635</td>
<td>0.398893</td>
</tr>
<tr>
<td>0.517984</td>
<td>0.446419</td>
</tr>
<tr>
<td>0.438868</td>
<td>0.431204</td>
</tr>
<tr>
<td>0.437553</td>
<td>0.501783</td>
</tr>
<tr>
<td>0.398761</td>
<td>0.334996</td>
</tr>
<tr>
<td>0.506564</td>
<td>0.484124</td>
</tr>
<tr>
<td>0.489211</td>
<td>0.506362</td>
</tr>
<tr>
<td>0.584810</td>
<td>0.448226</td>
</tr>
<tr>
<td>0.476670</td>
<td>0.434737</td>
</tr>
<tr>
<td>0.530668</td>
<td>0.444919</td>
</tr>
<tr>
<td>0.500062</td>
<td>0.442432</td>
</tr>
<tr>
<td>0.672593</td>
<td>0.548837</td>
</tr>
<tr>
<td>0.411158</td>
<td>0.363092</td>
</tr>
<tr>
<td>0.408718</td>
<td>0.544778</td>
</tr>
<tr>
<td>0.523448</td>
<td>0.356299</td>
</tr>
</tbody>
</table>
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.
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!

\[
\text{sum} = \begin{array}{c|c}
\text{Trial #} & 0 & 5 & 10 & 15 & 20 & 25 & 30 \\
\hline
0 & & & & & & & \\
0.1 & & & & & & & \\
0.2 & & & & & & & \\
0.3 & & & & & & & \\
0.4 & & & & & & & \\
0.5 & & & & & & & \\
0.6 & & & & & & & \\
0.7 & & & & & & & \\
0.8 & & & & & & & \\
\end{array}
\]

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 +, -, *, /, ++, --, >>, <<, ^, |
   - 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)
   ... sum = sum + myPartialSum;
   - Performs (sum, product, and, or, ...) in O(log₂N) time instead of O(N)
   - Operators include +, -, *, /, ++, --
   - Operators include +=, -=, *=, /=
   - Operators include ^=, |=, &=

Secretly creates a `sum` private variable for each thread, fills them all separately, then adds them together on O(log₂N) time.
If You Understand NCAA Basketball Brackets, You Understand Reduction

Source: ESPN

[Diagram of NCAA Tournament brackets with seedings and matches listed.]
Reduction vs. Atomic vs. Critical

Evaluations per Second

- Reduction
- Atomic
- Critical

MegaFunction Evaluations per Second
Why Not Do Reduction by Creating Your Own *sums* Array, one for each Thread?

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

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

- Blocks if the lock is not available
- Then sets it and returns when it is available
- If the lock is not available, returns 0
- If the lock is available, sets it and returns 1

**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

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
    }
}
```

There is an **implied barrier** at the end.
What do OpenMP Sections do for You?

omp_set_num_threads(1);

omp_set_num_threads(2);

omp_set_num_threads(3);
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;
    }
}
}
```