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 (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?

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

We will get to “false sharing” in the cache notes

Memory Allocation in a Multithreaded Program

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!
" );
exit( 0 );
#endif

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!
" );
exit( 0 );
#endif

Using OpenMP on Linux
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();

Asking which thread this one is:
me = omp_get_thread_num();

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 ! 
", omp_get_thread_num() );
}
return 0;
}

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 !

Uh-oh…

First Run
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

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

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

Fourth Run
Hello, World, from thread #1
Hello, World, from thread #3
Hello, World, from thread #5
Hello, World, from thread #2
Hello, World, from thread #4
Hello, World, from thread #6
Hello, World, from thread #0

There is no guarantee of thread execution order!

Creating OpenMP Team of Threads

#include <stdio.h>
#include <omp.h>

int main( )
{
omp_set_num_threads(8);
#pragma omp parallel default(none)
{
... }
return 0;
}

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

Creating OpenMP threads in Loops

#include <stdio.h>
#include <omp.h>

for ( int index = start ; index terminate condition; index changed )
{
... }

This tells the compiler to parallelize the for-loop into multiple threads. Each thread automatically gets its own personal copy of the variable 'index' 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:
  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

- `for( index = start; index < end; index <= end; index > end; index >= end; index++)`  
- `++index`  
- `index++`  
- `index-= incr`  
- `index -= decr`  
- `index = index + incr`  
- `index = incr + index`  
- `index = index - decr`  
- `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];
...
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"

### Arithmetic Operations Among Threads – A Problem

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

### OpenMP Scheduling

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

### OpenMP Allocation of Work to Threads

<table>
<thead>
<tr>
<th>Static 1</th>
<th>Static 2</th>
<th>Static 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0.3, 6, 9 1 1.4, 7, 10 1 4.5, 6, 7</td>
<td>0 0.1, 6.7 1 2.3, 8.9</td>
<td>0 0.1, 2.3 1 4.5, 6.7 1 8.9, 10.11</td>
</tr>
</tbody>
</table>

- 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
Here’s a trapezoid integration example (covered in another note set).\textsuperscript{19}
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.
And, not only are the answers bad, they are not even consistently bad!

\begin{verbatim}
0.469635 0.398893
0.51704 0.426519
0.383868 0.431204
0.437553 0.501783
0.39761 0.339996
0.50656 0.484124
0.483211 0.505362
0.54610 0.44826
0.47567 0.43737
0.530668 0.44919
0.500062 0.442342
0.672953 0.48537
0.411158 0.363092
0.408718 0.544778
0.523448 0.356299
\end{verbatim}

Here’s a trapezoid integration example (covered in another note set).\textsuperscript{20}
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!

Here’s a trapezoid integration example (covered in another note set).\textsuperscript{21}
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!

---

**Arithmetic Operations Among Threads – Three Solutions**\textsuperscript{22}

\begin{verbatim}
#pragma omp atomic
sum = sum + myPartialSum;
\end{verbatim}

• Fixes the non-deterministic problem
• But, serializes the code
• Operators include +, -, *, /, ++, --, >>, <<, ^, |, etc.

\begin{verbatim}
#pragma omp critical
sum = sum + myPartialSum;
\end{verbatim}

• Also fixes it
• But, serializes the code

\begin{verbatim}
#pragma omp parallel for reduction(+:sum),private(myPartialSum)
\end{verbatim}

• Performs (sum,product,and,...) in O(log_2N) time instead of O(N)
• Operators include +, -, *, /, ++, --, ^=, |=, &=

---

**If You Understand NCAA Basketball Brackets, You Understand Reduction**\textsuperscript{23}

Source: ESPN

---

**Reduction vs. Atomic vs. Critical**\textsuperscript{24}

---

**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; }

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

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( );
}
```

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

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 private(p)
Process( p );
```

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 );
    }
  }
}