Embedded computing for scientific and industrial imaging applications

Lecture 14 - OpenMP, critical sections, parallel for loops

OpenMP

“Open Specifications for MultiProcessing”

Standard for shared memory parallel programming.
For shared memory computers, such as multi-core.

Can be used with Fortran (77/90/95/2003), C and C++.

Complete specifications at http://www.openmp.org

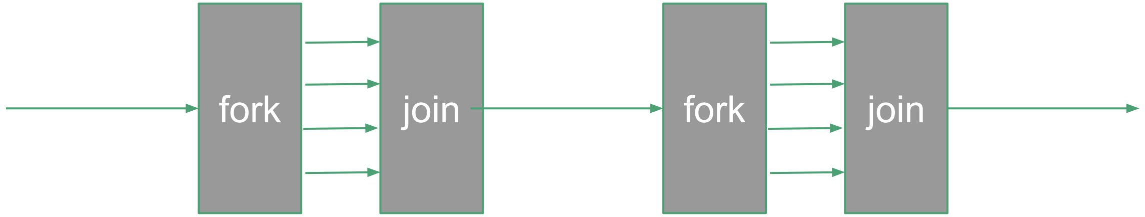
References

- http://www.openmp.org
- http://www.openmp.org/wp/resources/
- B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, MIT Press, 2007.
- R. Chandra, L. Dagum, et. al., Parallel Programming in OpenMP, Academic Press, 2001.
- https://github.com/OpenMP

OpenMP — Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread.



OpenMP — Basic Idea

- Rule of thumb: One thread per processor (or core),
- User inserts compiler directives telling compiler how statements are to be executed
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
- Compiler generates explicit threaded code
- Dependencies in parallel parts require synchronization between threads
- User's job to remove dependencies in parallel parts or use synchronization. (Tools exist to look for race conditions.)

OpenMP - compilers

http://openmp.org/wp/openmp-compilers/

- MS - OpenMP in Visual C++
  /openmp (Enable OpenMP 2.0 Support)
  - Project-> Properties -> C/C++ -> Language
  - Change OpenMP Support to Yes(/openmp)
- GNU - gcc
  Free and open source
  From GCC 6.1, OpenMP 4.5 is fully supported in C and C++.
  Compile with -fopenmp to enable OpenMP.

OpenMP compiler directives

Uses compiler directives that start with #pragma (!$ in fortran.)
These look like comments but are recognized when compiled with the flag -fopenmp(g++), /openmp (VS).

OpenMP statements:

OpenMP compiler directives, e.g.

```
    #pragma omp parallel do
```

Calls to OpenMP library routines:

```
    #include <omp.h>       //need this header
    omp_set_num_threads(2)
```

OpenMP directives

```
#pragma omp directive  [clause ...]
                 if (scalar_expression)
                 private (list)
                 shared (list)
                 default (shared | none)
                 firstprivate (list)
                 reduction (operator: list)
                 copyin (list)
                 num_threads (integer-expression)
```

A few OpenMP directives

```
#pragma omp parallel [clause]
{    // block of code
}
#pragma omp parallel do [clause]
{    // do loop
}
#pragma omp barrier
// wait until all threads arrive
```

Several others we'll see later…

OpenMP

API also provides for (but implementation may not support):

- Nested parallelism (parallel constructs inside other parallel constructs)
- Dynamically altering number of threads in different parallel regions

The standard says nothing about parallel I/O.

OpenMP provides "relaxed-consistency" view of memory.

Threads can cache their data and are not required to maintain exact consistency with real memory all the time.

```
#pragma omp flush
```

can be used as a memory fence at a point where all threads must have consistent view of memory

OpenMP test code

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

int main()
{
    int thread_num;

    omp_set_num_threads(2);
    printf("Testing openmp ...\n");
    #pragma omp parallel
    {
        #pragma omp critical
        {
            thread_num = omp_get_thread_num();
            printf("This thread = %d\n", thread_num);
        }
    }
    return 0;
}
```

OpenMP test code output

Compiled with OpenMP:

```
    $gcc -fopenmp test.c
    $./a.out

    Testing openmp …
    This thread = 0
    This thread = 1
```

(or threads might print in the other order!)

OpenMP test code

```
//Specify number of threads to use:
omp_set_num_threads(2)
```

Can specify more threads than processors, but they won’t execute in parallel.

The number of threads is determined by (in order):

- Evaluation of if clause of a directive
  (if evaluates to zero or false ⇒ serial execution)
- Setting the num_threads clause
- the omp_set_num_threads() library function
- the OMP_NUM_THREADS environment variable
- Implementation default

OpenMP test code

```
#pragma omp parallel
{
    #pragma omp critical
    {
        thread_num = omp_get_thread_num();
        printf("This thread = %d\n", thread_num);
    } //end of omp critical
} //end of omp parallel
```

The `#pragma omp parallel` block spawns two threads and each one works independently, doing all instructions in block. Threads are destroyed at `} //end of omp parallel`
However, the statements are also in a `#pragma omp critical` block, which indicates that this section of the code can be executed by only one thread at a time, so in fact they are not done in parallel.

So why do this? The function `omp_get_thread_num()` returns a unique number for each thread and we want to print both of these.

OpenMP test code

Incorrect code without critical section:

```
#pragma omp parallel
{
    thread_num = omp_get_thread_num();
    printf("This thread = %d\n", thread_num);
}
```

Why not do these in parallel?

1. If the prints are done simultaneously they may come out garbled (characters of one interspersed in the other).
2. thread_num is a shared variable. If this were not in a critical section, the following would be possible:

```
Thread 0 executes function, sets thread_num=0
Thread 1 executes function, sets thread_num=1
Thread 0 executes print statement: "This thread = 1"
Thread 1 executes print statement: "This thread = 1"
```

There is a data race or race condition.

OpenMP test code

Could change to add a private clause:

```
#pragma omp parallel
{
    #pragma omp critical
    {
        thread_num = omp_get_thread_num();
        printf("This thread = %d\n", thread_num);
    } //end of omp critical
} //end of omp parallel
```

Then each thread has it’s own version of the `thread_num` variable.

OpenMP parallel for loops

```
#pragma omp parallel for
for(i=0; i<N; i++)
{
     //do stuff for each i
}
```

indicates that the for loop can be done in parallel.

Requires:

what’s done for each value of i is independent of others

Different values of i can be done in any order.

The iteration variable i is private to the thread: each thread has its own version.

By default, all other variables are shared between threads unless specified otherwise.

OpenMP parallel for loops

This code fills a vector `y` with function values that take a bit of time to compute:

```
dx = 1.0 / (n+1.0);

#pragma omp parallel for private(x)
for(i = 1; i <= n; i++)
{
  x = i*dx;
  y[i-1] = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.0);
}
```

Elapsed time for n = 10^8, without OpenMP: about 3.3 sec.

Elapsed time using OpenMP on 2 processors: about 1.9 sec.

OpenMP parallel for loops

This code is not correct:

```
#pragma omp parallel for
for(i = 1; i <= n; i++)
{
  x = i*dx;
  y[i-1] = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.0);
}
```

By default, x is a shared variable.

Might happen that:

Processor 0 sets x properly for one value of i,

Processor 1 sets x properly for another value of i,

Processor 0 uses x but is now incorrect.

OpenMP parallel for loops

Correct version:

```
#pragma omp parallel for private(x)
for(i = 1; i <= n; i++)
{
  x = i*dx;
  y[i-1] = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.0);
}
```

Now each thread has its own version of x.
Iteration counter i is private by default.
Note that `dx, n, y` are shared by default. OK because:

`dx, n` are used but not changed,
`y` is changed, but independently for each `i`

OpenMP parallel for loops

Incorrect code:

```
dx = 1.0 / (n + 1.0)
#pragma omp parallel for private(x, dx)
for(i = 1; i <= n; i++)
{
  x = i*dx;
  y[i-1] = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.0);
}
```

Specifying dx private won't work here.

This will create a private variable dx for each thread but it will be uninitialized.

Will run but give garbage.

OpenMP parallel for loops

Could fix with:

```
dx = 1.0 / (n + 1.0)
#pragma omp parallel for firstprivate(dx)
for(i = 1; i <= n; i++)
{
  x = i*dx;
  y[i-1] = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.0);
}
```

The firstprivate clause creates private variables and initializes to the value from the master thread prior to the loop.

There is also a lastprivate clause to indicate that the last value computed by a thread (for i = n) should be copied to the master thread’s copy for continued execution.

OpenMP parallel for loops

! from $CSE6000/codes/OpenMP/private1.c

```
n = 7;
y = 2.0;
#pragma omp parallel for firstprivate(y) lastprivate(y)
for(i = 1; i <= n; i++)
{
    y = y + 10.0;
    x[i - 1] = y;
    #pragma omp critical
    {
        printf("i = %d, x[i - 1] = %f\n", i, x[i - 1]);
    }
}
printf("At end y = %f\n", y);
```

OpenMP parallel for loops

Run with 2 threads: The 7 values of i will be split up, perhaps

i = 1, 2, 3, 4 executed by thread 0,

i = 5, 6, 7 executed by thread 1.

Thread 0’s private y will be updated 4 times, 2 → 12 → 22 → 32 → 42

Thread 1’s private y will be updated 3 times, 2 → 12 → 22 → 32

might produce:

```
i = 1, x[i - 1] = 12.000000
i = 5, x[i - 1] = 12.000000
i = 2, x[i - 1] = 22.000000
i = 6, x[i - 1] = 22.000000
i = 3, x[i - 1] = 32.000000
i = 7, x[i - 1] = 32.000000
i = 4, x[i - 1] = 42.000000
At end y = 32.000000
```

Order might be different but final y will be from i = 7.

OpenMP synchronization

```
    #pragma omp parallel for
    for(i = 1; i <= n; i++)
    {
        /do stuff for each i
    }
```

There is an implicit barrier at the end of the loop.

The master thread will not continue until all threads have finished with their subset of `1, 2, ..., n.`

Except if :

```
    #pragma omp parallel for nowait
```

Conditional clause

Loop overhead may not be worthwhile for short loops.

(Multi-thread version may run slower than sequential)

Can use conditional clause:

```
#pragma omp parallel for if (n > 1000)
for(i = 1; i <= n; i++)
{
    // do stuff
}
```

If `n ≤ 1000` then no threads are created,

master thread executes loop sequentially

Nested loops

```
#pragma omp parallel for private(i)
for(j = 0; j < m; j++)
{
    for(i = 0; i < n; i++)
    {
      a[j * n + i] = 0.0;
    }
}
```

The loop on `j` is split up between threads.

The thread handling `j=0` does the entire loop on i,

sets a[0 * n + 0], a[0 * n + 1], ..., a[0 * n + n].

Note: The loop iterator `i` must be declared private!

`j` is private by default, `i` is shared by default.

Nested loops - Which is better? (assume m ≈ n)

```
#pragma omp parallel for private(i)
for(j = 0; j < m; j++)
{
    for(i = 0; i < n; i++)
    {
        a[j * n + i] = 0.0;
    }
}
```

```
for(j = 0; j < m; j++)
{
    #pragma omp parallel for
    for(i = 0; i < n; i++)
    {
        a[j * n + i] = 0.0;
    }
}
```

Nested loops - Which is better? (assume $m \approx n$)

```
#pragma omp parallel for private(i)
for(j = 0; j < m; j++)
{
    for(i = 0; i < n; i++)
    {
        a[j * n + i] = 0.0;
    }
}
```

```
for(j = 0; j < m; j++)
{
    #pragma omp parallel for
    for(i = 0; i < n; i++)
    {
        a[j * n + i] = 0.0;
    }
}
```

The first has less overhead: Thread created only once.
The second has more overhead: Thread created `m` times.

Nested loops

Incorrect code for replicating first column:

```
#pragma omp parallel for private(j)
for(i = 1; i < n; i++)
{
    for(j = 0; j < m; j++)
    {
        a[j * n + i] = a[j * n + i - 1];
    }
}
```

Corrected: (`j`'s can be done in any order, `i`'s cannot)

```
#pragma omp parallel for private(i)
for(j = 0; j < m; j++)
{
    for (i = 1; i < n; i++)
    {
        a[j * n + i] = a[j * n + i - 1];
    }
}
```

Reductions

Incorrect code for computing $\|x\|_1 = \sum_i |x_i|$

```
norm = 0.0;
#pragma omp parallel for
for (i = 0; i < n; i++)
{
    norm = norm + fabs(x[i]);
}
```

There is a race condition: each thread is updating same shared variable norm.

Correct code:

```
#pragma omp parallel for reduction(+ : norm)
for (i = 0; i < n; i++)
{
    norm = norm + fabs(x[i]);
}
```

A reduction reduces an array of numbers to a single value.

Reductions

A more complicated way to do this:

```
#pragma omp parallel private(mysum) shared(norm)
{
    mysum = 0;
    #pragma omp for
    for(i = 0; i < n; i++)
    {
        mysum = mysum + fabs(x[i]);
    }
    #pragma omp critical
    {
        norm = norm + mysum;
    }
}
```

Some other reductions

Can do reductions using +, −, $*$, min, max, .and., .or., some others

General form:

```
#pragma omp parallel for reduction(operator : list)
```

Example with max:

```
double y = -1.0e3; //very negative value
#pragma omp parallel for reduction(max: y)
for(i = 1; i < n; i++)
{
  y = fmax(y, x[i]);
}
printf("max of x = %f\n", y);
```