

OpenMP for Multi-Threading

ICS432 - Fall 2008
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Pthread is good and bad

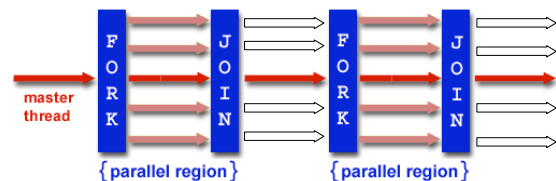
- Multi-threaded programming in C with Pthreads is very flexible
 - You can have threads create threads recursively
 - You can stop threads individually
 - You can pretty much do anything you can think off
- But in many cases, you don't need that flexibility
- Example:
 - You want to compute the sum of an array
 - You just want to "split it up" among threads
 - No recursion, threads killing other threads, etc.
- It seems painful to deal with the Pthread API "just" to achieve this, over and over
 - Creating the do_work() functions
 - Pack the arguments into structures
 - etc.

OpenMP to the Rescue

- Goal: make shared memory programming easy (or at least easier than with pthread)
- How?
 - A library with some simple functions
 - The definition of a few C pragmas
 - pragmas are a way to make the language extensible
 - provide an easy way to give hints/information to a compiler
 - A compiler
 - which is more like a preprocessor really

Fork-Join Model

- Program begins with a **Master thread**
- **Fork:** **Teams of threads** created at times during execution
- **Join:** Threads in the team synchronize (barrier) and only the master thread continues execution



OpenMP and #pragma

- One needs to specify blocks of code that are executed in parallel
- For example, a *parallel section*:
`#pragma omp parallel [clauses]`
 - Defines a section of the code that will be executed in parallel
 - The "clauses" specify many things including what happens to variables
 - All threads in the section execute the same code

Compiling with gcc

- Since version 4.2 OpenMP is supported
- Compile with the `-fopenmp` flag
- On our cluster gcc 4.2 is installed in
 - `/home/casanova/public/bin/gcc`
- You must set environment variable `LD_LIBRARY_PATH` to `/home/casanova/public/lib`

“Hello World” Example

```
#include <omp.h>
int main(){
print("Start\n");
#pragma omp parallel
{ // note the `{`
printf("Hello World\n");
} // note the `}`

/* Resume Serial Code */
printf("Done\n");
}

% my_program
Start
Hello World
Hello World
Hello World
Done
```

“Hello World” Example

```
#include <omp.h>
int main(){
print("Start\n");
#pragma omp parallel
{
printf("Hello World\n");
}
printf("Done\n");
}

% my_program
Start
Hello World
Hello World
Hello World
Done
```

- Questions
 - How many threads?
 - This is not useful of course because all threads do exactly the same thing

How Many Threads?

- Set via an environment variable
 - e.g., in tcsh: `setenv OMP_NUM_THREADS 8`
- Set via the OpenMP API

```
void omp_set_num_threads(int number);
int omp_get_num_threads();
```
- Typically, a function of the number of processors/cores available
 - We often take the number of threads identical to the number of processors/cores
 - But we've seen that sometimes more is better

Threads Doing Different Things

```
#include <omp.h>
int main() {
int iam = 0, np = 1;
#pragma omp parallel private(iam, np)
{
np = omp_get_num_threads();
iam = omp_get_thread_num();
printf("Hello from thread %d out of %d threads\n",
iam, np);
}
}

% setenv OMP_NUM_THREADS 3
% my_program
Hello from thread 0 out of 3
Hello from thread 1 out of 3
Hello from thread 2 out of 3
```

What about Memory

- This is good so far, but what we need now is a way to figure out how threads share or don't share memory
- When writing Pthread code, you know exactly which variables are shared
 - global variables
 - pointers passed to `do_work()` function
- and which variables are not shared
 - local variables to the `do_work()` function
 - variables passed to the `do_work()` function
- With OpenMP, you need to specify whether a variable is shared or private

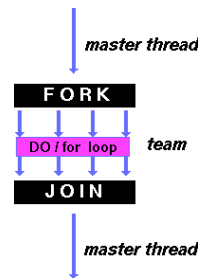
Data Scoping and Clauses

- **Shared**: all threads access the **single copy** of the variable, created in the master thread
 - it is the **responsibility** of the programmer to ensure that it is shared appropriately
- **Private**: a **short-lived** copy of the variable is created for **each thread**, and discarded at the end of the parallel region (but for the master)
- There are other variations
 - `firstprivate`: initialization from the master's copy
 - `lastprivate`: the master gets the last value updated by the last thread to do an update
 - and several others(Look in the on-line material if you're interested)

Work Sharing directives

- We have seen the concept of a **parallel region**
- Work Sharing directives make it possible to have threads “share work” *within a parallel region*.
 - For Loop
 - Sections
 - Single

For Loops



- Share iterations of the loop across threads
- Represents a type of “data parallelism”
 - do the same operation on pieces of the same big piece of data
- Program correctness must NOT depend on which thread executes which iteration
 - No ordering!

For Loop Example

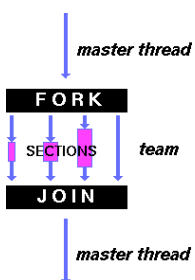
```
#include <omp.h>
#define N 1000
main () {
    int i, chunk; float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c) private(i)
    {
        #pragma omp for schedule(dynamic)
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section - threads are
        synchronized*/
}
```

For Loop and “nowait”

- With “nowait”, threads do not synchronize at the end of the loop
 - i.e., threads may exit the #pragma omp for at different times

```
#pragma omp parallel shared(a,b,c) private(i)
{
    #pragma omp for schedule(dynamic) nowait
    for (i=0; i < N; i++) {
        // do some work
    }
    // Threads get here at different times
}
```

Sections



- Breaks work into **separate** sections
- Each section is executed by a thread
- Can be used to implement “task parallelism”
 - do different things on different pieces of data
- If more threads than sections, then some are idle
- If fewer threads than sections, then some sections are serialized

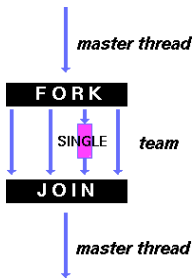
Section Example

```
#include <omp.h>
#define N 1000
main () {
    int i; float a[N], b[N], c[N];
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c) private(i)
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                for (i=0; i < N/2; i++)
                    c[i] = a[i] + b[i];
            }
            #pragma omp section
            {
                for (i=N/2; i < N; i++)
                    c[i] = a[i] + b[i];
            }
        } /* end of sections */
    } /* end of parallel section */
}
```

Section #1

Section #2

Single



- Serializes a section of code within a parallel region
- Sometimes more convenient than terminating a parallel region and starting it later
 - especially because variables are already shared/private, etc.
- Typically used to serialize a small section of the code that's not thread safe
 - e.g., I/O

Combined Directives

- Sometimes it is cumbersome to create a parallel region and *then* create a parallel for loop, or sections, just to terminate the parallel region
- Therefore OpenMP provides a way to do both at the same time
 - `#pragma omp parallel for`
 - `#pragma omp parallel sections`

Synchronization and Sharing

- When variables are shared among threads, OpenMP provides tools to make sure that the sharing is correct
- Typical race condition

```
int x = 0;
#pragma omp parallel sections shared(x)
{
    #pragma omp section
    { x = x + 1 }
    #pragma omp section
    { x = x + 2 }
}
```

Synchronization directive

- `#pragma omp master`
 - Creates a region that only the master executes
- `#pragma omp critical`
 - Creates a critical section
- `#pragma omp barrier`
 - Creates a "barrier"
- `#pragma omp atomic`
 - Create a "mini" critical section

Critical Section

```
#pragma omp parallel for shared(sum)
private(i)
for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical
    {
        sum = sum + value;
    }
}
```

Barrier

```
if (x == 2) {
    #pragma omp barrier
}
```

- All threads in the current parallel section will synchronize
 - they will all wait for each other at this instruction
- Must appear within a basic block

Atomic

```
#pragma omp atomic
    i++;
```

- Only for some expressions
 - `x = expr` (no mutual exclusion on expr evaluation)
 - `x++`
 - `++x`
 - `x--`
 - `--x`
- Is about atomic access to a memory location
- Some implementations will just replace `atomic` by `critical` and create a basic blocks
- But some may take advantage of the cool hardware instructions that work atomically

Conclusion

- If you have an application that is “regular” in terms of concurrency
 - All threads sort of do the same thing
 - They’re doing it more or less at the same time
 - Data sharing patterns are simple and well understood
 - The number of threads doesn’t need to change dynamically
- Then, IF there is a decent OpenMP compiler for your platform (which there should)
 - e.g., gcc 4.2 and later
- You should most likely use OpenMP
 - And you may look very smart by just adding a few pragmas to a piece of code and accelerating it
- Typically, “systems” people use pthreads, while “scientific computing” people use OpenPM, but it’s changing
- More info in a tutorial linked off the Web site