Multithreading in C with OpenMP

ICS432 - Spring 2015
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Pthreads are good and bad

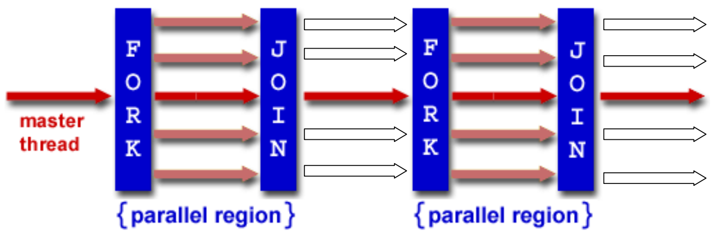
- Multi-threaded programming in C with Pthreads is flexible
  - You can have threads create threads recursively
  - You can stop threads individually
  - You can pretty much do anything you can think of
- But in many cases, you don't need all that flexibility
- Example:
  - You want to compute the sum 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 Pthreads)
- How?
  - A library with some simple functions
  - The definition of a few C pragmas
    - pragmas are a way to "extend" C
    - provide an easy way to give hints/information to a compiler
  - A compiler
    - Since gcc version 4.2 OpenMP is supported
    - Compile with the -fopenmp flag
  - A limited (but useful!) programming model

Fork-Join Programming 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

“Hello World” Example

```
#include <omp.h>
int main(){
printf(“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
{ // note the ‘{‘
   printf(“Hello World\n”);
} // note the ‘}’
/* Resume Serial Code */
 printf(“Done\n”);
}
```

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

- How many threads?

How Many Threads?

- Set via the OpenMP API

```
void omp_set_num_threads(int number);
int omp_get_num_threads();
```

- Typically, you set it to be exactly the number of cores on your machine

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);
  }
}

% export 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 in 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

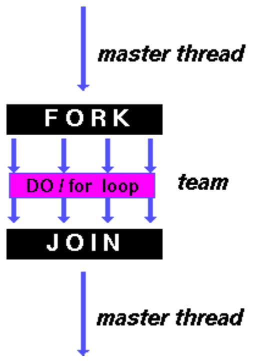
- Shared: all threads access the **single copy** of the variable, created in the master thread
  - It is the responsibility of the programmer to ensure that it is shared appropriately (no race conditions)
- Private: a **short-lived** copy of the variable is created for **each thread**, and discarded at the end of the parallel region (but for the master)
- There are other useful variations
  - firstprivate: initialization from the master's copy
  - lastprivate: the master gets the last value updated by the last thread to do an update
  - etc.

  (Look in the on-line material for all details)

Work Sharing directives

- We have seen the concept of a parallel region to run multiple threads
- 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
- To implement 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
  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 nowait
    for (i=0; i < N; i++) {
      // do some work
    }
    // Threads get here at different times
  }
```

Loop Parallelizations

- Not all loops can be safely parallelized
  - Not specific to OpenMP, but OpenMP makes loop parallelization so easy that one must be careful not to break the code
- If there are inter-iteration dependencies, we can have race conditions
- Simple example:

```
for (i=1; i < N; i++) {
        a[i] += a[i-1] * a[i-1]
}
```

  - We have a “data dependency”: iteration i needs data produced by previous iteration i-1

Three Kinds of Dependencies

- Consider a for loop: for (i=0; i < N; i++)
- Data dependency:
  - Example: a[i] += a[i-1] * a[i-1];
  - Iteration i needs data produced by the previous iteration
- Anti-dependency:
  - Example: a[i] = a[i+1] * 3;
  - Iteration i must use data before it is updated by the next iteration
- Output dependency
  - Example: a[i] = 2*a[i+1]; a[i+2] = 3 * a[i]
  - Different iterations write to the same addresses

Dealing with Dependencies

- With these dependencies, one simple cannot parallelize the loop
- Some of these loops are inherently sequential and simply cannot be parallelized
- Some have race conditions that we could fix with locks (will see how OpenMP does locks in a few slides)
  - But in this case, the loop likely becomes sequential
  - In fact, slower than sequential due to locking overhead
- Bottom-line: inter-iteration dependencies are bad

Dealing with Dependencies

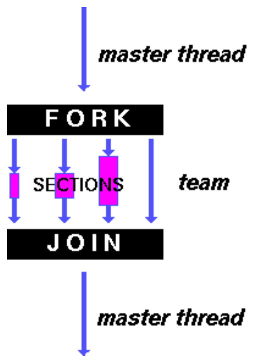
- Sometimes, one can rewrite the loop to remove dependencies
- Example:

```
for (i=1; i < N; i++) {
        a[i] = a[i-1] + 1;
}
```

⇩

```
for (i=1; i < N; i++) {
        a[i] = a[0] + i;
}
```

Sections



- Breaks work into separate sections
- Each section is executed by a thread
- To implement Task-parallelism
  - do different things on different data
- If more threads than sections, then some threads remain idle
- If fewer threads than sections, then some sections are serialized

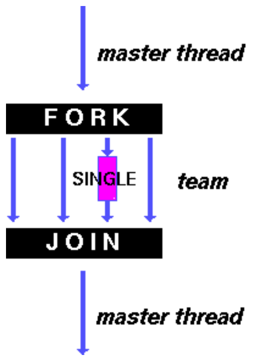
Section Example

```
#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

- 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 opm parallel sections
- These are the versions you should use by default

Combined Parallel/For Directive

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

- Note the **private(i)**
  - Each thread should have its own copy of the loop index (e.g., while Thread 0 is at i=4, Thread 1 could be at i=6)
- Putting a private(a) would create a copy of array a
  - At least in principle

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++; }
  #pragma omp section
  { x--; }
  }
```

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

Barrier

```
#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 hardware instructions that work atomically

OpenMP and Scheduling

- When you do parallelize a loop with OpenMP, one issue is: which thread does what?
  - This question is termed “scheduling”
- There are several options, but two are really common
- Static: each of p thread performs 1/p of the iterations

```
omp_set_num_threads(4);
#pragma omp parallel for private(i) schedule(static)
for (i=0; i < 1000; i++) {
  // some computation
}
```

- Thread 0 will do iterations 0-249
- Thread 1 will do iterations 250-499
- etc.

OpenMP and Scheduling

- Dynamic: each thread “grabs” work when done with one iteration: useful for irregular iterations

```
omp_set_num_threads(2);
#pragma omp parallel for private(i) schedule(dynamic)
for (i=0; i < 1000; i++) {
  // some computation
}
```

- Thread 0 will do iteration 0
- Thread 1 will do iteration 1
- Thread 1 may do iteration 2
- Thread 0 may do iteration 3
- Thread 1 may do iteration 4
- Thread 1 may do iteration 5
- ...

OpenMP Dynamic Scheduling

- How does OpenMP implement dynamic scheduling?
- With a lock!!
- There is a global variable under the cover that keeps track of the next iteration that should be done, say next_it
- Each thread does:

```
while (next_it != loop_bound)
    lock
    next_it ++
    unlock
    do iteration next_it
```

Dynamic scheduling, chunk size

- For dynamic scheduling, one can specify a "chunk size", e.g., "schedule(dynamic, 10)"

-
  ```
  while (next_it != loop_bound)
      lock
      next_it = min(loop_bound,
                        next_it+chunk_size)
      unlock
      do iteration next_it
  ```
- The idea is to reduce overhead
- But we also might increase load imbalance
- Picking the best chunk size is difficult

Static vs. Dynamic Scheduling

- Static Scheduling
  - Very low overhead (each thread just does its own loop on its own loop index range)
  - Poor load balancing if iterations are not of the same duration
- Dynamic Scheduling
  - Goo load balancing if iterations are not of the same duration
  - But some overhead due to the shared loop index and the used of a lock, and picking the best chunk size is really hard
- So if your iterations are all the same, definitely use static, otherwise probably use dynamic

Guided Scheduling

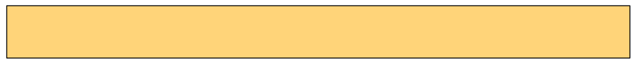
- Say you have thousands of iterations and they are not all the same
  - some are slow, some are fast
- Dynamic is overkill at the beginning of the execution load balancing is not important
  - But Dynamic is key at the end because we don't want to "wait for the last thread"
- So at the beginning of the execution we don't want to suffer from the overhead of dynamic scheduling, but at the end we don't want to suffer from the load imbalance of static scheduling
- What do we do? We strike a compromise!

Guided Scheduling

- At the beginning of the execution we give out large "chunks of iterations" to threads
- As execution progresses, we give them smaller chunks
- Say we have N iterations and p threads/cores
- We first give out the first N/2 iterations to the p threads (each thread gets a chunk of (N/2)/p iterations)
- We then give out the first N/4 iterations to the p threads (each thread gets a chunk of (N/4)/p iterations)
- Let's see it on a picture...

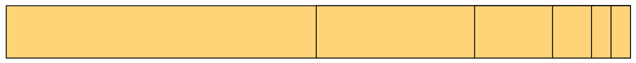
Guided Scheduling

- Say we have 4 threads and this many iterations:



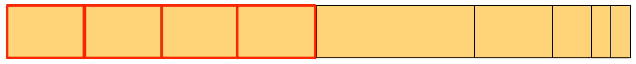
Guided Scheduling

- We split the iterations in halves...



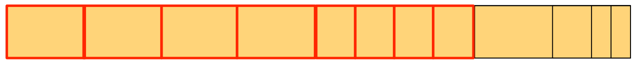
Guided Scheduling

- Then we split each piece into 4 chunks



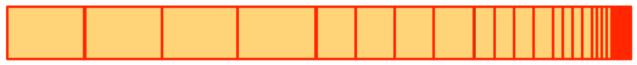
Guided Scheduling

- Then we split each piece into 4 chunks



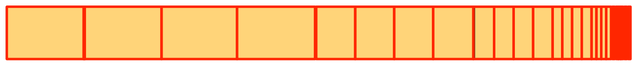
Guided Scheduling

- Then we split each piece into 4 chunks



Guided Scheduling

- Chunks are assigned to threads *dynamically* from left to right order



- So essentially, guided scheduling is like dynamic scheduling, but with a decreasing chunk size

Conclusion

- If you have an application that is "regular" in terms of concurrency
  - All threads sort of do the same thing
  - Data sharing patterns are simple and well understood
  - The number of threads doesn't need to change dynamically
- Then 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 OpenMP
- More info in this OpenMP tutorial