Lecture 17: OpenMP Basics

William Gropp

www.cs.illinois.edu/~wgropp
Model of Computation

• Fork/join model

• Note difference between abstract model and implementation
  ✷ Fork/join model does not require that threads are created each time
OpenMP Syntax

- Mostly directives
  - `#pragma omp construct [ clause ... ]`
- Some functions and types
  - `#include <omp.h>`
- Most apply to a block of code
  - Specifically, a “structured block”
  - Enter at top, exit at bottom only*
    - `exit()`, `abort()` permitted
Different OpenMP styles of Parallelism

- OpenMP supports several different ways to specify thread parallelism
  - General parallel regions
    - All threads execute the code, roughly as if you made a routine of that region and created a thread to run that code
  - Parallel loops
    - Special case for loops; simplifies data parallel code
  - Task parallelism
    - New(ish) in OpenMP 3

- Several ways to manage thread coordination, including
  - Master regions
  - Locks

- Memory model for shared data
  - “flush”
Parallel Region

• `#pragma omp parallel`
  ```
  {
    ... code executed by each thread
  }
  ```
• Effectively a single thread runs before:
  ◆ “fork” at the beginning
  ◆ “join” at the end
• Single thread runs after
Hello World in OpenMP: The Serial Version

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    {
        int id = 0;
        int np = 1;
        printf( "Hello world %d of %d\n", id, np );
    }

    return 0;
}
```
Hello World in OpenMP: The Parallel Version

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

int main(int argc, char *argv[]) {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf( "Hello world %d of %d\n", id, np );
    }
    return 0;
}
```
Hello World in OpenMP: The Parallel Version

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

int main(int argc, char *argv[]) {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf("Hello world %d of %d\n", id, np);
    }
    return 0;
}
```
Hello World in OpenMP: The Parallel Version

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

int main(int argc, char *argv[])
{
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        int np = omp_get_num_threads();
        printf( "Hello world %d of %d\n", id, np );
    }
    return 0;
}
Notes on Hello World

• Variables declared outside of the parallel region are *shared by all* threads
  ♦ If id declared outside of the `#pragma omp parallel`, it would have been shared by the threads, *possibly* causing erroneous output

• Why? What would go wrong? Why is it only “possibly”?  
• Take a few minutes to see why – just use two threads but remember that if “int id;” is outside of the parallel region, id is in a single memory location that both threads access.
Private Variables

- Private clause can be used to make thread-private versions of such variables:
  ```c
  #pragma omp parallel private(id)
  {
    id = omp_get_thread_num();
    printf("My thread num = %d\n",id);
  }
  ```

- More details
  - What is their value on entry? Exit?
  - OpenMP provides ways to control that
  - Can use default(none) to require the sharing of each variable to be described (a sort of "implicit none" for OpenMP)
Master Region

• It is often useful to have only one thread execute some of the code in a parallel region. I/O statements are a common example
Example of OMP Master

```c
#pragma omp parallel
{
    #pragma omp master
    {
        int k = omp_get_num_threads();
        printf("Number of Threads requested = %i\n", k);
    }
}
```
Data Parallel Computation and Loops

• OpenMP provides an easy way to parallelize a loop:
  
  ```c
  #pragma omp parallel for
  for (i=0; i<n; i++) c[i] = a[i];
  ```

• OpenMP handles index variable (no need to declare in for loop or make private)

• Which thread does which values?
Scheduling of Loop Computation

- Let the OpenMP runtime decide
- The decision is about how the loop iterates are *scheduled*
- OpenMP defines three choices of loop scheduling:
  - Static – Predefined at compile time. Lowest overhead, predictable
  - Dynamic – Selection made at runtime
  - Guided – Special case of dynamic; attempts to reduce overhead
Example of parallel for: STREAM

- Using OpenMP in STREAM COPY
  
  ```
  #pragma omp parallel for
  for (j=0; j<STREAM_ARRAY_SIZE; j++)
      c[j] = a[j];
  ```

- Running STREAM
  
  ```
  export OMP_NUM_NUM_THREADS=4
  ./stream
  ```
STREAM Performance on Blue Waters
Comparison With Performance Model

- **Good**: Performance increases linearly to 6 cores
- **Bad**: Odd dips from 8 to 12
- **Unsurprising**: Dip at 16
  - Possible contention with OS
- **Many open questions here**
  - What are some of them?
  - Stop here and write some down, then go on to see a few possibilities
Possible Issues

- How are threads in STREAM assigned to cores in the node?
- There are two processor chips in the node. The simple performance model assumes a single memory pathway
  - Each chip introduces a separate limit
  - How are threads distributed across cores?
- Are these measurements repeatable?
  - STREAM code makes no effort to get repeatable result
Questions

• Find out how to use OpenMP on your platform of choice. Recent versions of gcc, for example, support OpenMP with the option \texttt{-fopenmp}
  ♦ Clang compiler adding openmp support now, so make sure your “gcc” is a real gcc

• Test that your option works by writing and running a program that prints the number of threads available (and more than 1!)
Loop Scheduling

- static, dynamic, guided
  - Plus auto (let compiler choose) and runtime (set with environment variable)
- Syntax is
  ```c
  #pragma omp parallel for \n  schedule(kind[,chunksize])
  ```
- E.g.,
  ```c
  #pragma omp parallel for \n  schedule(guided,100)
  for (i=0; i<n; i++) c[i]=a[i];
  ```
STREAM and Loop Schedule

• STREAM as distributed uses the default (static) schedule
  ♦ Best when loop limits known, work per iteration constant, cores only used by the application

• Question: Are all of those assumptions correct?
STREAM and Loop Schedule

- Question: Are all of those assumptions correct?
  - That last one (cores only used be application) is the most suspect
  - Try running STREAM with one thread per available core and:
    - Static
    - Dynamic
    - Guided
  - How do they perform?
More on Loops: Reductions

- What happens with code like this:
  
  ```c
  #pragma omp parallel for
  For (i=0; i<n; i++)
      sum += a[i];
  ```

- Like all variables, there is one “sum” variable; **all** threads access it.

- But addition is not atomic:
  ```c
  ld sum, r1
  ld a[i], r2
  fadd r1, r2, r3
  st r3, sum
  ```
### Race Conditions

<table>
<thead>
<tr>
<th>Thread 0 (core 0)</th>
<th>Thread 1 (core 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld sum, r1</td>
<td>Ld sum, r1</td>
</tr>
<tr>
<td>Ld a[i], r2</td>
<td>Ld a[j], r2</td>
</tr>
<tr>
<td>Fadd r1, r2, r3</td>
<td>Fadd r1, r2, r3</td>
</tr>
<tr>
<td>St r3, sum</td>
<td>St r3, sum</td>
</tr>
</tbody>
</table>

- In this order, the contribution from thread 0 \( (a[i]) \) is lost – thread 0 has lost a race with thread 1 to read sum, add \( a[i] \) to it, and store it back before thread 1 accesses sum.
Reductions in OpenMP

- Reductions are both common and important for performance.
- OpenMP lets the programmer indicate that a variable is used for a reduction with a particular operator.

```c
sum = 0;
#pragma omp parallel for reduction(+,sum)
for (i=0; i<n; i++) sum += a[i]*b[i];
```
More Reading

• *Using OpenMP*, B. Chapman, G. Jost, A. van der Pas
  [http://mitpress.mit.edu/books/using-openmp](http://mitpress.mit.edu/books/using-openmp)

• Many tutorials online

• OpenMP official site:
  [www.openmp.org](http://www.openmp.org)
Questions

• What are the pros and cons of block scheduling for parallelizing a loop?