OpenMP

CIS-4230, Fall 2012
Peter C. Chapin
OpenMP?

• Standard for parallel programming...
  – Compiler extensions + run time support library
    • Supports C and Fortran
    • Requires compiler support
      – gcc
      – Microsoft Visual C/C++
      – Intel C/C++
      – others...
      – not clang! (at least not yet)
  – Programmer adds # pragmas defining parallel code
Background

• Targets scientific and engineering apps
  – Large numeric computations
    • Floating point intensive
    • Big arrays
    • Loops that process big arrays

• Manages threads
  – Creates and manages a thread pool
  – Coordinates threads
  – Allows programmer to take control when needed
#pragma

• The #pragma directive...
  – Part of standard C. Used to control compiler
    • No #pragmas defined by standard
    • Unknown #pragmas are to be ignored
  – Many compilers use #pragmas for
    • Controlling warnings
    • Controlling listings
    • Controlling optimization and code generation
  – OpenMP uses #pragmas to control parallelization
The Basics

• Executing a for loop in parallel
  - #pragma omp parallel for
    for( i = 0; i < SIZE; ++i ) {
      array[i] *= 2.0;
    }
  - Compiler creates “team” of threads at #pragma
    • Splits the loop automatically
      – Each thread executes a subset of iterations in parallel.
    • Team joins together at the end of the construct.
Restrictions

• How many times does this loop execute?
  - `for ( i = 0; i < SIZE; i = f(i) ) {
      array[i] *= 2.0;
  }`

  - OpenMP compilers can’t tell either.

  - `#pragma omp parallel for requires...`
    • Only relational operations `<`, `<=`, `>`, `>=`
    • Increment expression involves integer operators `++`, `--`, `+=`, or `-=`
Even More Basic

- Executing arbitrary code in parallel
  - #pragma omp parallel sections
    
    #pragma omp parallel section
    f();
    #pragma omp parallel section
    g();
  
  - Only two threads used
    - One a single core sections executed serially.
More Primitive Directives

• The most basic directive...
  – #pragma omp parallel
    {
      f();
    }
  – A team of threads is created
    • All threads execute the same code
Can be Combined

• Example...
  
  ```
  #pragma omp parallel
  {
      #pragma omp for
      for( i = 0; i < SIZE; ++i )
      array[i] *= 2.0;
      #pragma omp sections
      {
          #pragma omp section
          f( );
          #pragma omp section
          g( );
      }
  }
  ```
#pragma omp single

• Special code executed by a single thread
  
  – #pragma omp parallel
     
     #pragma omp for
     for( i = 0; i < SIZE; ++i ) …
     #pragma omp single
     {
       printf("One thread!\n");
     } // Barrier inserted here.
     #pragma omp for
     for( i = 0; i < SIZE; ++i ) …
  
  

What About Sharing?

• Take a closer look...
  – #pragma omp parallel for
    for( i = 0; i < SIZE; ++i ) {
      array[i] *= 2.0;
    }
  – Each thread must have its own i
    • Loop control variables are “private” by default.
  – The threads must share array
    • Other variables are shared by default.
Making Sharing Explicit

• Same as previous example...
  
  – #pragma omp parallel for \
    private(i) shared(array) \
    for( i = 0; i < SIZE; ++i ) {
    array[i] *= 2.0;
  }

  – Can use “clauses” like private and shared to override defaults.

  – Several other clauses are defined
Private Variables

• Normally undefined on entry and exit

```c
int n = 0;
#pragma omp parallel sections private(n)
{
    #pragma omp section
    n = n + 1;  // n uninitialized!
    #pragma omp section
    n = 1;
}
printf("n = %d\n", n);  // n undefined!
```
First Private Variables

- **Initialized on entry**

  ```c
  int n = 0;
  #pragma omp parallel sections
     firstprivate(n)
  {
     #pragma omp section
     n = n + 1;  // n initially zero
     #pragma omp section
     n = n * 2;  // n initially zero
  }
  printf("n = %d\n", n);  // n undefined!
  ```
Last Private Variables

- Well defined on exit

```c
int n = 0;
#pragma omp parallel sections
    lastprivate(n)
{
    #pragma omp section
    n = 1;
    #pragma omp section
    n = 2;
}
printf("n = %d\n", n);  // n definitely 2
```
Synchronization, Part 1

• Barriers
  – #pragma omp parallel
    {
      #pragma omp for
      for( I = 0; I < SIZE; ++I ) …
      #pragma omp barrier
      #pragma omp for
      for( I = 0; I < SIZE; ++I ) …
    }
  – Threads in a team wait at the barrier until all arrive.
Synchronization, Part 2

• Critical sections
  
  ```c
  int n = 0;
  #pragma omp parallel shared(n)
  {
    #pragma omp critical
    {
      n++;
    }
  }
  
  – Only one thread at a time executes critical section.
  ```
Reduction

• Common operation...
  - `int sum = 0;`
    ```
    #pragma omp parallel for \
    reduction(+:sum)
    for( I = 0; I < SIZE; ++I )
        sum += array[i];
    ```
  • Each thread in the team computes a local value for sum
  • Those local values are combined using +
  • Value of sum after parallel loop is the overall sum
Reduction Operators

• Only certain operators are supported
  • + (addition)
  • * (multiplication)
  • – (subtraction)
  • & (bitwise AND)
  • | (bitwise OR)
  • ^ (bitwise XOR)
  • && (logical AND)
  • || (logical OR)
Be Careful!

• Can this loop be parallelized?
  – #pragma omp parallel for
    for( i = 0; i < SIZE - 1; ++i ) {
      array[i] += array[i + 1];
    }
  – Consider what happens at the team boundary
  – Up to you to get this right!
    • OpenMP compiler won’t help.
Alternative

• How about this loop?
  – #pragma omp parallel for
    for( i = 0; i < SIZE - 1; ++i )
      array_2[i] =
        array_1[i] + array_1[i + 1];
  – Notice array_1 not changed
    • Immutable data easier to handle
    • Does require more memory
More Complete Version

• Copy the result back in parallel
  
  - #pragma omp parallel
    
    { 
      #pragma omp for
      for( i = 0; i < SIZE - 1; ++i )
        array_2[i] =
        array_1[i] + array_1[i + 1];

      #pragma omp barrier
      
      #pragma omp for
      for( i = 0; i < SIZE - 1; ++i )
        array_1[i] = array_2[i];
    }