**OpenMP**

Overview

Commands

Compiling on the IBM systems

---

**Reading**

- Chapter 8 of the Wilkinson and Allen book
  - chapter is on shared memory programming
  - section 8.5 describes OpenMP
- Tutorials available via [http://www.openmp.org](http://www.openmp.org)
  - look in the “resources” section

---

**History**

- In the early days of parallel computing each company had its own compiler support for parallel programming
  - Sequent’s “parallel processing macros”, 1985
  - IBM, Cray, others soon followed
- Software developers urged standardization
  - early efforts: Parallel Computing Forum, ANSI
- OpenMP started in the late 1990’s
- Now a “de facto” standard
  - not endorsed (yet) by ANSI or any other organization
  - widely adopted by many companies, including IBM, SGI

---

**Shared Memory Model**

- One of the main classes of parallel processors is the shared memory multiprocessor (SMP)
  - see lecture notes (ppintro) from the first week
- From a programmer’s perspective, any process can access any variable
  - one large address space

```
lw $t5, X
sw $s0, X
```
Advantages of SMP

- More efficient communication
  - processes can share large data structures
  - process P sends the address of structure to process Q
  - reduces copying, a major source of program overhead

- Allows for incremental development
  - start with working, optimized sequential program
  - parallelize those parts that need to be improved
    - e.g. inner loop in Mandelbrot program

- One piece of code for parallel and sequential machines

Disadvantages of SMP

- Requires explicit synchronization
  - overhead from synchronization may limit scalability
  - MPI messages often do implicit synchronization

- Synchronization errors can be subtle
  - difficult to reproduce
  - difficult to locate

- Suitable for small to medium scale parallelism
  - need for cache coherence, other hardware mechanisms have prevented massive parallelism (1000s of processors)
    - expense (half the hardware cost devoted to memory management?)
    - performance overhead

Fork/Join Parallelism

- Application begins in sequential mode
- At some point (a fork) the program creates several threads
  - threads share the same address space and other O/S environment (e.g. I/O streams)
- When all threads are complete the program returns to sequential mode
- Usually threads implement a form of SPMD parallelism

Master and Team Members

- In OpenMP terminology, the original program is a single thread known as the master
- When the master encounters a construct that defines a parallel region, a team of new threads is created
  - the master continues as thread 0
  - team members are threads 1...n-1
- After all threads finish execution of the parallel region the master resumes execution in the sequential region
Pragmas

- OpenMP can be used to parallelize Fortran, C, or C++ programs
- Constructs are written as *pragmas*
  - *pragma* -- from “pragmatic” -- is an existing part of these languages
  - used to provide hints to the compiler about how to optimize a program
- In C/C++, pragmas are preprocessor directives -- lines that start with a “pound sign”
  ```cpp
  #include <...>
  ```
- An OpenMP command:
  ```cpp
  #pragma omp [commands]
  ```

OpenMP Pragma Syntax

- OpenMP pragmas are of the form:
  ```cpp
  #pragma omp directive [clause...]
  ```
- *omp* tells the compiler this is an OpenMP command
- *directive* is the command name, *clauses* are optional arguments
- the command must fit on a single line
- white space is allowed for readability
- commands are case-sensitive
- Note: only one directive is allowed per pragma (with one exception, covered later)

Parallel Region

- The command to start a parallel region is *parallel*
  ```cpp
  #pragma omp parallel [options...]
  ```
- The compiler will generate code that causes the next C++ statement to be executed in parallel by each thread
  - may be an assignment, for or while loop, function call, etc
  - a set of statements can be grouped into a single statement with braces
  ```cpp
  #pragma omp parallel numthreads(4)
  {
    ...
    ...
  }
  ```

Number of Threads

- The number of threads created can be defined three ways:
  - a *num_threads* clause in the *parallel* command
  - a call to *omp_set_num_threads()* from a sequential region before the parallel region is started
    - defines a default number of threads for all subsequent parallel regions, in case they don’t specify a *num_threads* clause
  - the *OMP_NUM_THREADS* environment variable
    - sets a default that can be over-ridden by the other two methods
Hello, World

- We’re finally ready to see “Hello, World” with OpenMP:
  ```cpp
  #include <omp.h>
  int main(int argc, char *argv[]) {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
      cout << "hello, world" << endl;
    }
    return 0;
  }
  ```

Compiling and Running Hello, World

- Settings for your Makefile (assuming gcc or g++)
  ```makefile
  CXXFLAGS = -fopenmp
  ```

- The resulting binary is a regular Unix application:
  ```bash
  % ./hello
  ```

- The output:
  ```bash
  hellhhhelleolelollo,, wwooorld, wo,rl dworldrlld
  ```

What Went Wrong?

- Our first encounter with synchronization problems

- All threads share the same Unix environment
  - inherited from the master thread
  - may or may not have the same process ID (O/S decision, not specified by OpenMP)
  - may show up as a separate process in top, etc

- Part of the shared environment is the stdout stream inherited from the shell

- All threads are writing to the same stream

Synchronization

- What we need is a **critical region**
  - a section of code that is executed by just one process at a time
  - old terminology: a concept from operating systems and concurrent programming

- In OpenMP:
  ```cpp
  #pragma critical
  ```

- The critical region is the next C++ statement
  - may be a complex statement surrounded by brackets
  - but for obvious performance reasons, keep critical regions small
Synchronization (cont’d)

- OpenMP makes sure only one thread is in the region at any time
  - note a thread can continue after it leaves the critical region

Hello World, Take 2

- A better version:
  ```c
  #include <omp.h>
  int main(int argc, char *argv[]) {
    int main(int argc, char *argv[]) {
      omp_set_num_threads(4);
      #pragma omp parallel
      { #pragma omp parallel
        #pragma omp critical
          cout << "hello, world" << endl;
        #pragma omp critical
          cout << "hello, world" << endl;
      }
      return 0;
    }
  }
  ```

Runtime Environment

- We’ve already seen how to control the number of threads
  - `omp_set_num_threads()`
    - called from serial code
    - can also use an environment variable, or specify as part of `parallel`
- To find out how many threads were created:
  - `omp_get_num_threads()`
    - call from parallel code
- Other useful functions:
  - `omp_get_thread_num()`
  - `omp_get_max_threads()`
    - host-specified maximum
    - may be more than number of processors

Parallel for Loops

- The `for` pragma “unrolls” bodies of for loops
  - instances of the loop body are executed in parallel by team members
- Example: sum all the elements in the Array object A
  ```c
  #pragma omp for
  for (j = 0; j < nc; j++) {
    csum = 0;
    for (i = 0; i < nr; i++)
      csum += A(i,j);
    #pragma omp critical
    x += csum;
  }
  ```
Parallel for Loops (cont’d)

- For this construct to work, the loop has to be in **canonical form**
  
  ```c
  for (i = 0; i < n; i++) ...
  ```

  - the iterator has to be an integer
  - limited choice of test, increment expressions
  - for details: search for “canonical” in OpenMP specs (PDF on-line)

- A shortcut allows the `for` command on the same line as `parallel`:
  
  ```c
  #pragma omp parallel for [options...]
  ```

  - equivalent to `parallel` followed immediately by `for`

---

Shared vs Private Variables

- For `parallel for` to work each loop iteration must have its own copy of the loop index variable
  
  - `i = 0` on one thread, `i = 1` on the next thread, etc

- The rules:
  
  - by default all variables are **shared**
  - index variables are **private**
  - other variables can be declared private by specifying an option to the `parallel` command
  
  ```c
  #pragma omp parallel private(x,y)
  ```

  - variables can also be explicitly specified as shared, e.g.
    
    ```c
    #pragma omp parallel shared(A) private(n)
    ```

---

Shared vs Private Variables (cont’d)

- Any new variables declared within a parallel region are private
  
  ```c
  #pragma omp parallel
  {
  int i;
  ...
  }
  ```

- Advice:
  
  - examine the body of each parallel region
  - identify each variable used
  - declare every variable as shared or private

---

Scheduling

- In the simplest parallel `for` loop there will be one thread per iteration
  
  ```c
  #pragma omp parallel for numthreads(5)
  for (j = 0; j < 5; j++) {
    ...
  }
  ```

- In general there will be many more iterations than threads
  
  - Example: outer loop of Mandelbrot iterates over all columns
  - A runtime **scheduler** allocates iterations to threads

- Two types of schedules:
  
  - static
  - dynamic
Scheduling (cont’d)

- In a static schedule, the system decides before the loop is executed which threads will execute which iterations
- A fixed number of iterations is assigned to each thread
  - example: 1000 iterations, 4 threads
    - thread 0: \(i = 0..249\)
    - thread 1: \(i = 250..499\), etc
- In a dynamic schedule, a thread is given a “chunk” of iterations
- When a thread finishes a chunk, it is given the next available chunk
  - example: 1000 iterations, 4 threads, chunk = 10
    - thread 0 might do \(i = 0..9, 40..49, 60..69\), etc
    - chunk size can be specified as a parameter of parallel for

Synchronization

- We’ve already seen the critical statement as one form of synchronization
- There are several others
  - #pragma omp barrier
    - all threads wait at this statement before proceeding
    - example: make sure all threads have had a chance to read boundary values from neighboring threads
- See the manual for more statements
  - parallel regions can be ordered
  - a lock statement is a building block for more complex synchronization

Reduction

- A common operation in parallel programs is reduction
- Reduction is a concept from functional programming
  - example: computing the sum of a variable across all threads
  - in MPI: \(\text{MPI\_reduce}(\ldots)\)
  - can be applied to any binary associative operator
- In OpenMP, reduction is an alternative to using a critical region for this common operation
  - can be much more efficient, since reduction can be done in parallel
  - tree at right is for 8 threads

Reduction Example

- From the “Red Book”:
  ```c
  #pragma omp parallel for private(i) reduction(+:result)
  for (i = 0; i < 3; i++) {
    for(j = i + 1; j < 4; j++) {
      printf("Hello.\n");
      result = result + 1;
    }
  }
  ```
  (they didn’t ask for my advice, but if they had, I would have suggested “private j” or “for int j...”)
Other OpenMP Constructs

- There are many more OpenMP constructs
- Some examples:
  - sections
    - more general method than loop unrolling for sharing work
  - master
    - single
      - specify a region that is to be executed by just one thread (e.g. for I/O)
  - if
    - clause for parallel command
    - conditional parallelism (e.g. if problem big enough)

To Do

- To test your understanding of OpenMP:
  - write a "Hello, World" program for OpenMP
  - create a Makefile with CXX, CXXFLAGS for compiling the program on the ACISS
  - compile and test the program
  - explore critical regions, thread IDs, other constructs