Threaded Programming

Lecture 2: OpenMP fundamentals

Overview

- Basic Concepts in OpenMP
- History of OpenMP
- Compiling and running OpenMP programs

What is OpenMP?

- OpenMP is an API designed for programming shared memory parallel computers.

- OpenMP uses the concepts of *threads* and *tasks*

- OpenMP is a set of extensions to Fortran, C and C++

- The extensions consist of:
  - Compiler directives
  - Runtime library routines
  - Environment variables

Directives and sentinels

- A directive is a special line of source code with meaning only to certain compilers.
- A directive is distinguished by a sentinel at the start of the line.
- OpenMP sentinels are:
  - Fortran: `!$OMP`
  - C/C++: `#pragma omp`
- This means that OpenMP directives are ignored if the code is compiled as regular sequential Fortran/C/C++.

Parallel region

- The *parallel region* is the basic parallel construct in OpenMP.
- A parallel region defines a section of a program.
- Program begins execution on a single thread (the master thread).
- When the first parallel region is encountered, the master thread creates a team of threads (fork/join model).
- Every thread executes the statements which are inside the parallel region
- At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements

Parallel region

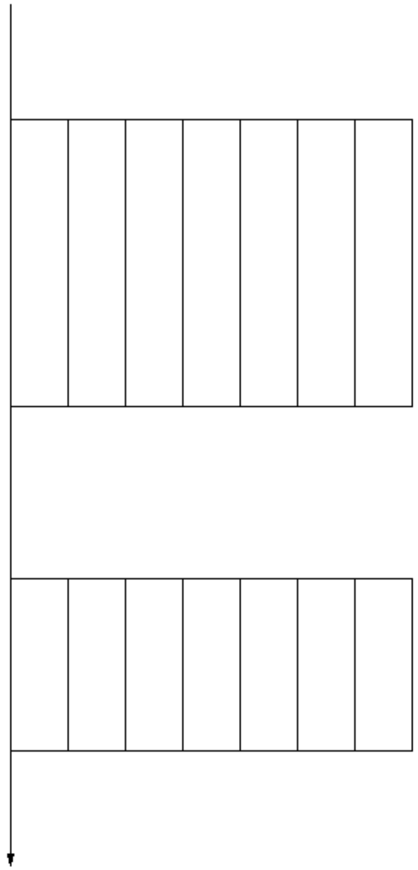
Sequential part

Parallel region

Sequential part

Parallel region

Sequential part



```
      PROGRAM FRED
          .
          .
!$OMP PARALLEL
          .
          .
          .
          .
          .
          .
          .
          .
          .
!$OMP END PARALLEL
          .
          .
          .
          .
          .
!$OMP PARALLEL
          .
          .
          .
          .
          .
!$OMP END PARALLEL
          .
          .
          .
```

```
int main(){
.
.
#pragma omp parallel
{
.
.
.
.
.
.
.
}
.
.
.
.
#pragma omp parallel
{
.
.
.
}
.
.
.
```

Shared and private data

- Inside a parallel region, variables can either be *shared* or *private.*
- All threads see the same copy of shared variables.
- All threads can read or write shared variables.
- Each thread has its own copy of private variables: these are invisible to other threads.
- A private variable can only be read or written by its own thread.
    - May be possible to access another thread's private data, but behaviour is unspecified, and very bad coding style!

Parallel loops

- In a parallel region, all threads execute the same code
- OpenMP also has directives which indicate that work should be divided up between threads, not replicated.
  - this is called worksharing
- Since loops are the main source of parallelism in many applications, OpenMP has extensive support for parallelising loops.
- The are a number of options to control which loop iterations are executed by which threads.
- It is up to the programmer to ensure that the iterations of a parallel loop are *independent*.
- Only loops where the iteration count can be computed before the execution of the loop begins can be parallelised in this way.

Synchronisation

- The main synchronisation concepts used in OpenMP are:
- Barrier
  - all threads must arrive at a barrier before any thread can proceed past it
  - e.g. end of parallel region, end of parallel loop
- Critical region
  - a section of code which only one thread at a time can enter
  - e.g. modification of shared variables
- Atomic accesses
  - an update/read/write of a variable which can be performed only by one thread at a time
  - e.g. modification of shared variables (special case)

Brief history of OpenMP

- Historical lack of standardisation in shared memory directives.
  - each hardware vendor provided a different API
  - mainly directive based
  - almost all for Fortran
  - hard to write portable code

- OpenMP forum set up by Digital, IBM, Intel, KAI and SGI. Now includes most major vendors (and some academic organisations, including EPCC).

- OpenMP Fortran standard released October 1997, minor revision (1.1) in November 1999. Major revision (2.0) in November 2000.
- OpenMP C/C++ standard released October 1998. Major revision (2.0) in March 2002.

History (cont.)

- Combined OpenMP Fortran/C/C++ standard (2.5) released in May 2005.
  - no new features, but extensive rewriting and clarification
- Version 3.0 released in May 2008
  - new features, including tasks, better support for loop parallelism and nested parallelism
- Version 4.0 released in July 2013
  - accelerator offloading, thread affinity, more task support,...
- Version 4.5 released November 2015
  - corrections and a few new features
- Version 5.0 released November 2018
  - some extra features, extensions to offloading

- Most current implementations comply with 4.5, except for offloading which has much less widespread support

OpenMP resources

- Web site:

  `www.openmp.org`

  - Official web site: language specifications, links to compilers and tools, mailing lists

- Books:
  - “Using OpenMP: Portable Shared Memory Parallel Programming”, Chapman, Jost and Van der Pas, MIT Press, ISBN: 0262533022
    - covers up to Version 2.5
  - “Using OpenMP—The Next Step”,
    Van der Pas, Stotzer and Terboven, MIT Press,
    ISBN: 9780262534789
    - covers Affinity, Accelerators, Tasking, and SIMD

Compiling and running OpenMP programs

- OpenMP is built-in to most of the compilers you are likely to use.
- To compile an OpenMP program you need to add a (compiler-specific) flag to your compile and link commands.
  - `-qopenmp` for Intel compilers
  - `-fopenmp` for gcc/gfortran
  - on by default in Cray compilers
- The number of threads which will be used is determined at runtime by the `OMP_NUM_THREADS` environment variable
  - set this before you run the program
  - e.g. `export OMP_NUM_THREADS=4`
- Run in the same way you would a sequential program
  - type the name of the executable

Exercise

Hello World

- Aim: to compile and run a trivial program.
- Vary the number of threads using the **`OMP_NUM_THREADS`** environment variable.
- Run the code several times - is the output always the same?

Reusing this material