Shared Memory Programming

Introduction to OpenMP

Overview

- Shared memory systems
- Basic Concepts in OpenMP
- Brief history of OpenMP
- Compiling and running OpenMP programs

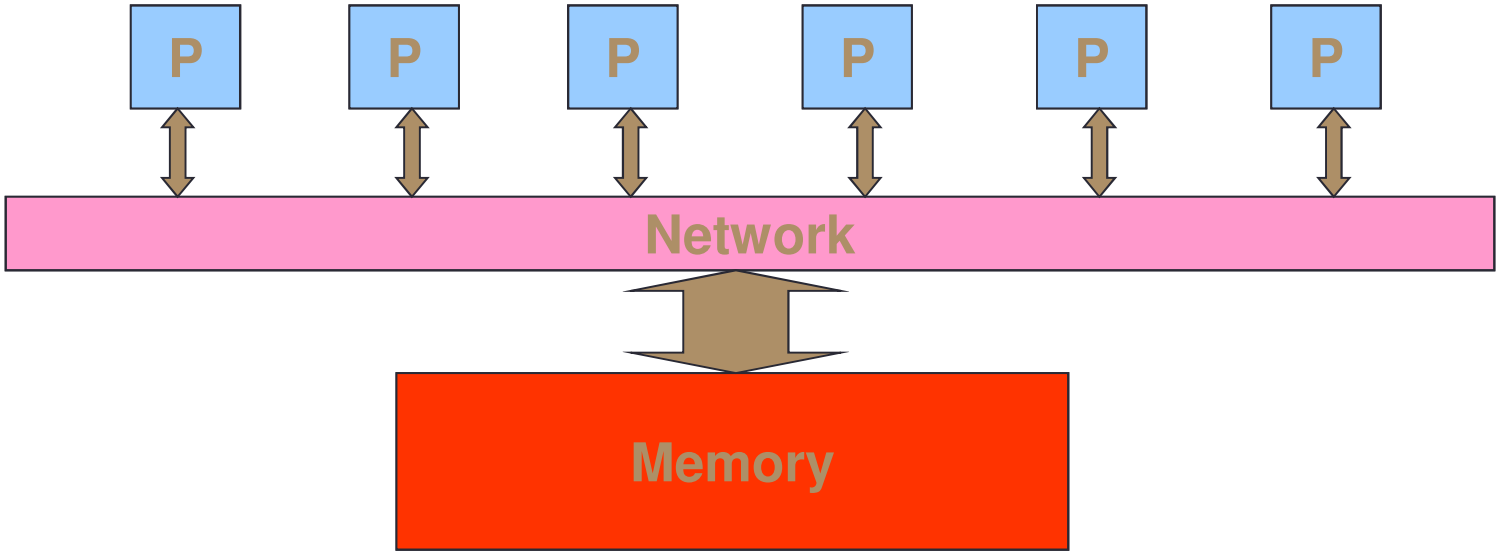
Shared memory systems

- OpenMP is designed for programming shared memory parallel computers.
- A shared memory computer consists of a number of processors together with some memory
- Key feature of shared memory systems is a *single address space* across the whole memory system.
  - every processor can read and write all memory locations in the system
  - one logical memory space
  - all processors refer to a memory location using the same address

Shared memory hardware

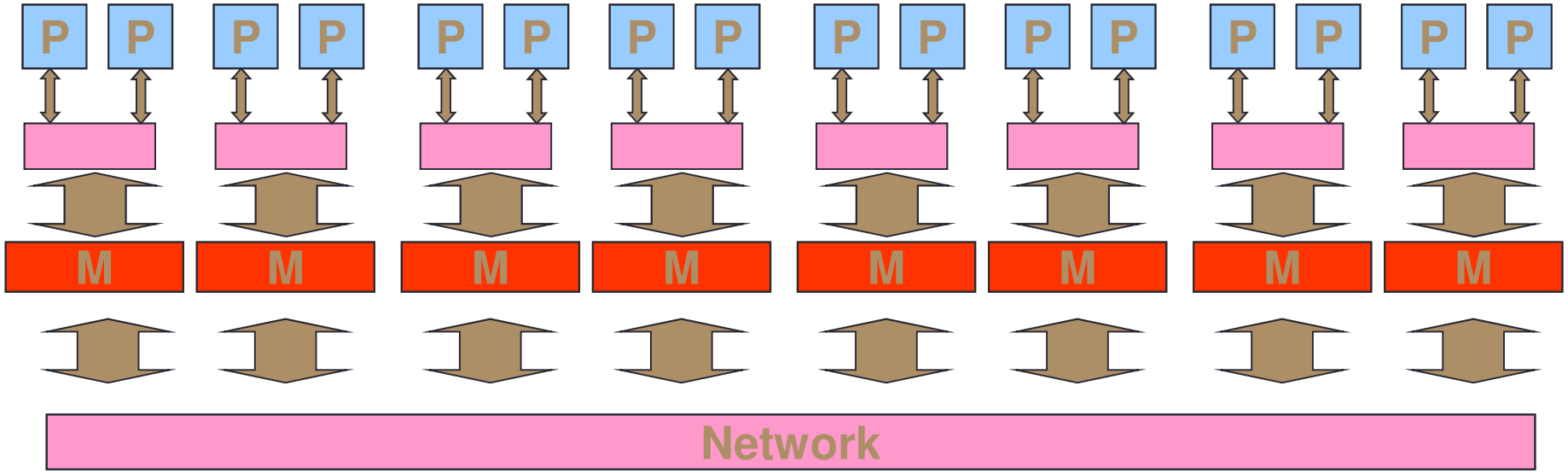
- Two main types of hardware:
  - true shared memory
  - distributed shared memory
- Difference is in how memory is physically organised
  - one large memory vs. multiple smaller memory units
- Difference is (almost) invisible to the programmer
  - some subtle performance implications

True shared memory



Examples: Sun X4600, all multicore PCs, IBM p575, NEC SX8, Fujitsu PRIMEQUEST

Distributed shared memory



Examples: SGI Altix, HP Superdome

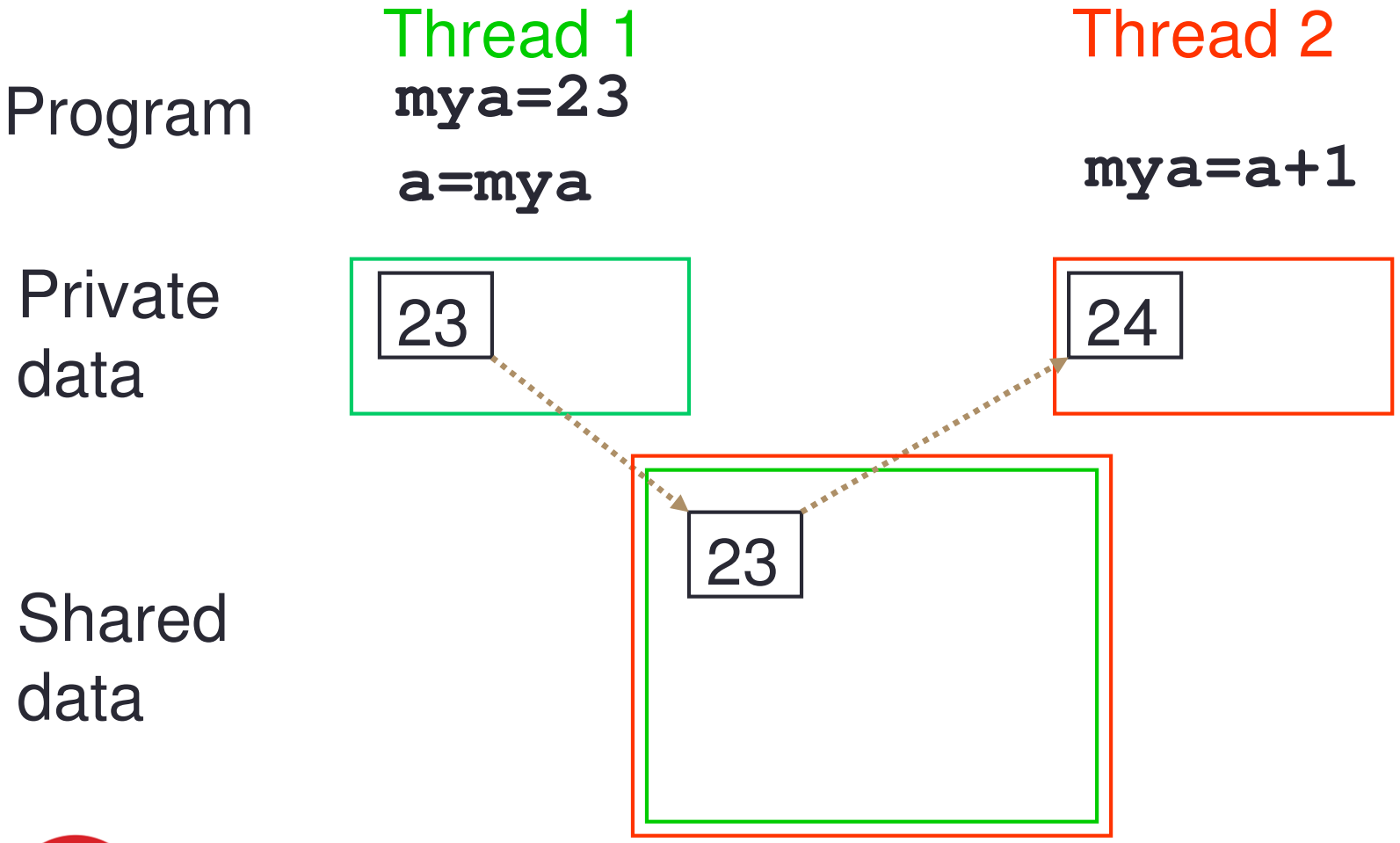
Programming Model

- The programming model for shared memory is based on the notion of threads
  - threads are like processes, except that threads can share memory with each other (as well as having private memory)
- Shared data can be accessed by all threads
- Private data can only be accessed by the owning thread
- Different threads can follow different flows of control through the same program
  - details of thread/process relationship is very OS dependent

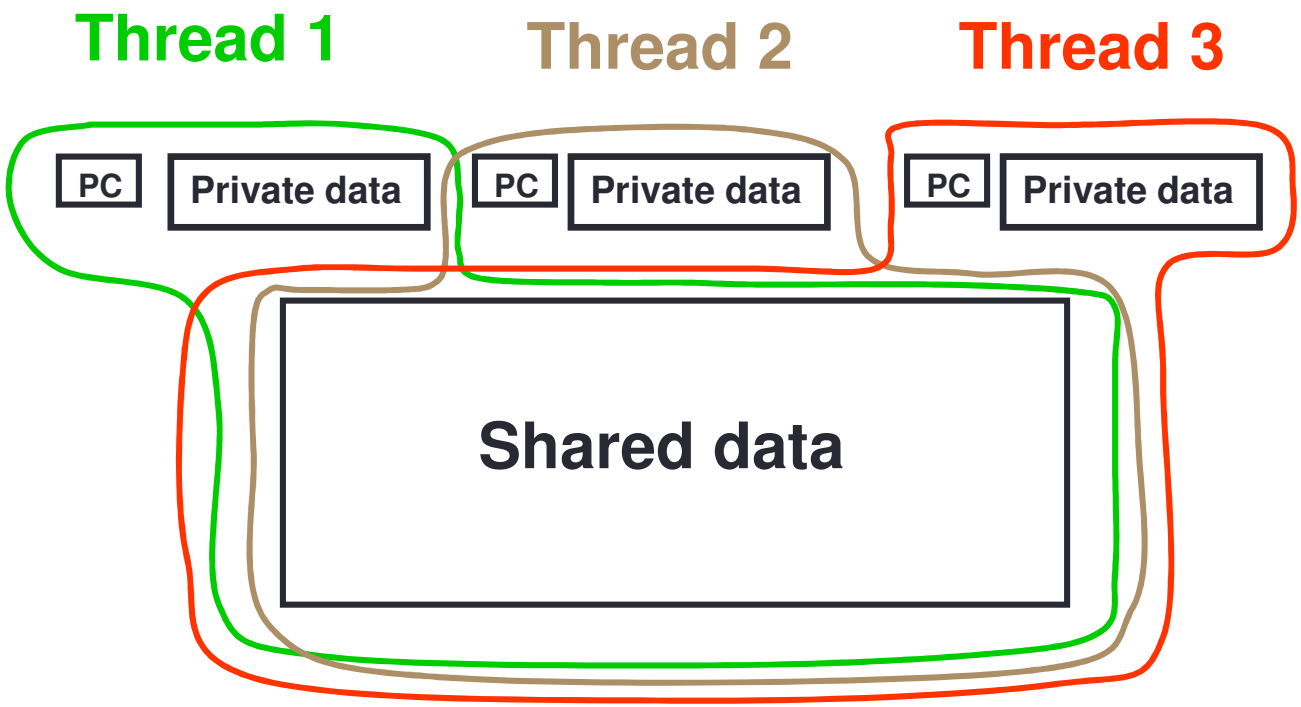
More About Threads

- Usually run one thread per processor
  - but could be more
- Threads communicate with each other only via shared data (no messages!)
  - thread 1 writes a value to a shared variable A
  - thread 2 can then read the value from A
- A *thread team* is a set of threads which co-operate on a task.
- The *master thread* is responsible for co-ordinating the team.

Thread Communication



Threads (cont.)



Overview of OpenMP

- Directives and sentinels
- Parallel regions
- Shared and private data
- Parallel loops
- Synchronisation
- Reductions

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: `!$OMP` (or `C$OMP` or `*$OMP` for F77)
  - C/C++: `#pragma omp`

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

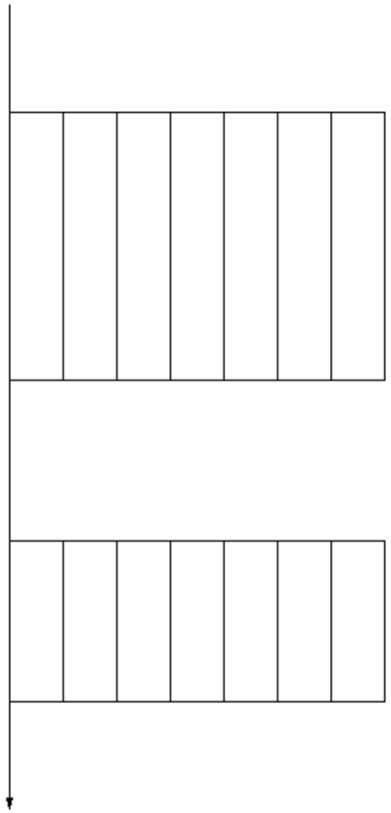
Sequential part

Parallel region

Sequential part

Parallel region

Sequential part



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

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.

Parallel loops

- Loops are the main source of parallelism in many applications.
- If the iterations of a loop are *independent* (can be done in any order) then we can share out the iterations between different threads.
- e.g. if we have two threads and the loop

```
do i = 1, 100
   a(i) = a(i) + b(i)
end do
```

we could do iteration 1-50 on one thread and iterations 51-100 on the other.

Synchronisation

- Need to ensure that actions on shared variables occur in the correct order: e.g.

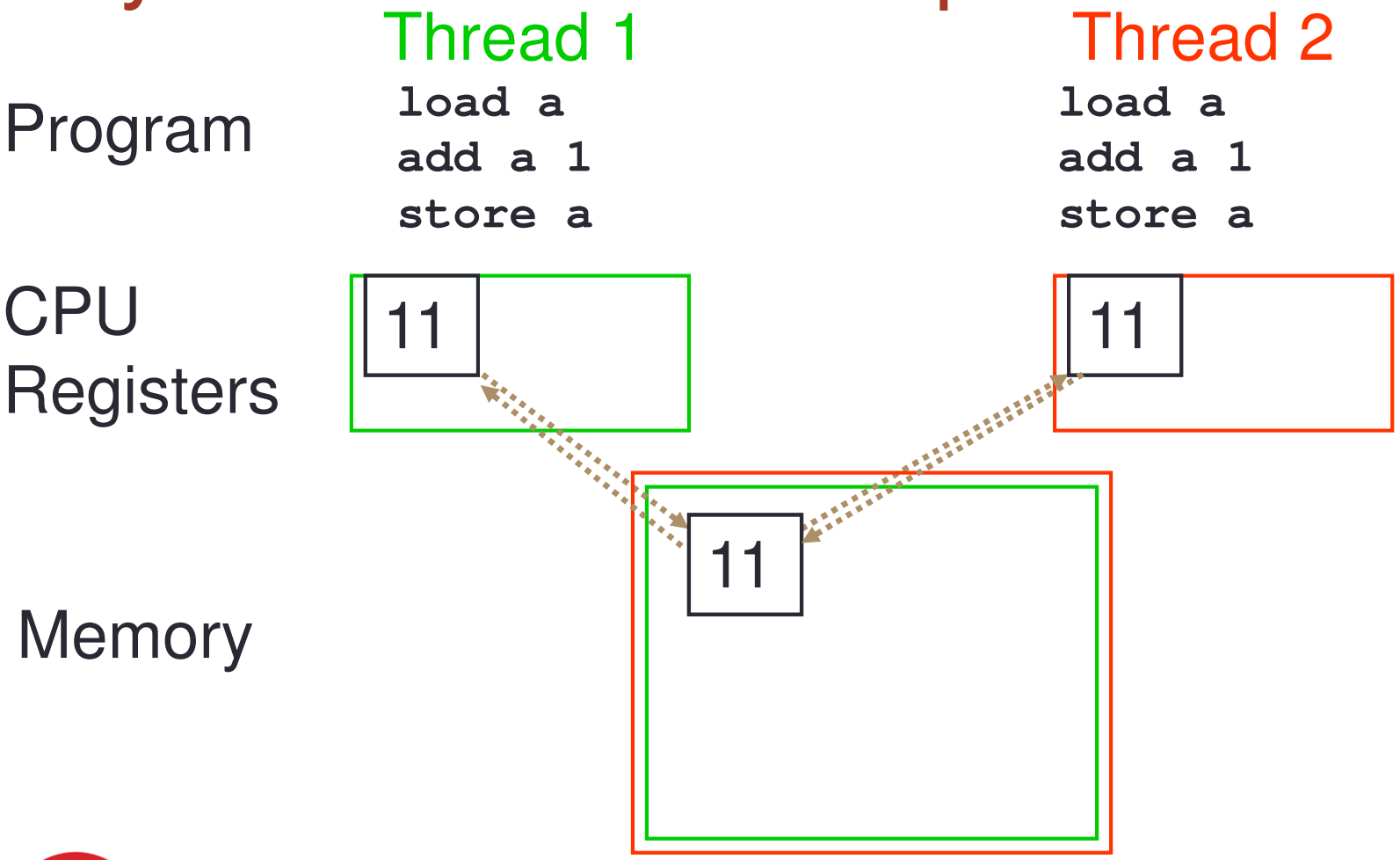
  thread 1 must write variable A before thread 2 reads it,

  or

  thread 1 must read variable A before thread 2 writes it.

- Note that updates to shared variables (e.g. `a = a + 1`) are *not* atomic!
- If two threads try to do this at the same time, one of the updates may get overwritten.

Synchronisation example



Reductions

- A *reduction* produces a single value from associative operations such as addition, multiplication, max, min, and, or.
- For example:

```
b = 0;
for (i=0; i<n; i++)
   b += a[i];
```

- Allowing only one thread at a time to update `b` would remove all parallelism.
- Instead, each thread can accumulate its own private copy, then these copies are reduced to give final result.

Brief history of OpenMP

- Historical lack of standardisation in shared memory directives. Each vendor did their own thing.
  - mainly directive based, almost all for Fortran
  - previous attempt at standardisation (ANSI X3H5, based on work of Parallel Computing forum) failed due to political reasons and lack of vendor interest.
- 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.

History (cont.)

- OpenMP C/C++ standard released October 1998. Major revision (2.0) in March 2002.
- 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
  - only beta compilers available just now

OpenMP resources

- Web sites:

  `www.openmp.org`

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

  `www.compunity.org`

  - OpenMP community site: more links, events, resources

- Books:
  - “Using OpenMP: Portable Shared Memory Parallel Programming” Chapman, Jost and Van der Pas, MIT Press, ISBN: 0262533022
  - “Parallel Programming in OpenMP”, Chandra et. al., Morgan Kaufmann, ISBN 1558606718.

Compiling

- OpenMP is built in to most FORTRAN, C and C++ compilers
- To compile an OpenMP program type with the GNU compilers:

Fortran: `gfortran -fopenmp -o prog prog.f`
C: `gcc -fopenmp -o prog prog.c`

`PGI compilers: -mp`
`Intel compilers: -openmp`

Running

To run an OpenMP program:

- Set the number of threads using the environment variable `OMP_NUM_THREADS`

e.g. `export OMP_NUM_THREADS=8` (bash/ksh)

or `setenv OMP_NUM_THREADS 8` (csh/tcsh)

- Can run just as you would a sequential program.

Running on ARCHER

```
#!/bin/bash --login
#PBS -N testprog
#PBS -l select=2
#PBS -l walltime=0:10:00
#PBS -A y07

cd /work/y07/y07/guest01
cp /work/y07/y07/guest01/testprog .

export OMP_NUM_THREADS=12
aprun -n 1 -N 1 -d $OMP_NUM_THREADS ./testprog
```

Practical session

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?