A Short Introduction to OpenMP

Mark Bull,

EPCC, University of Edinburgh

Overview

- Shared memory systems
- Basic Concepts in Threaded Programming
- Basics of OpenMP
- Parallel regions
- Parallel loops

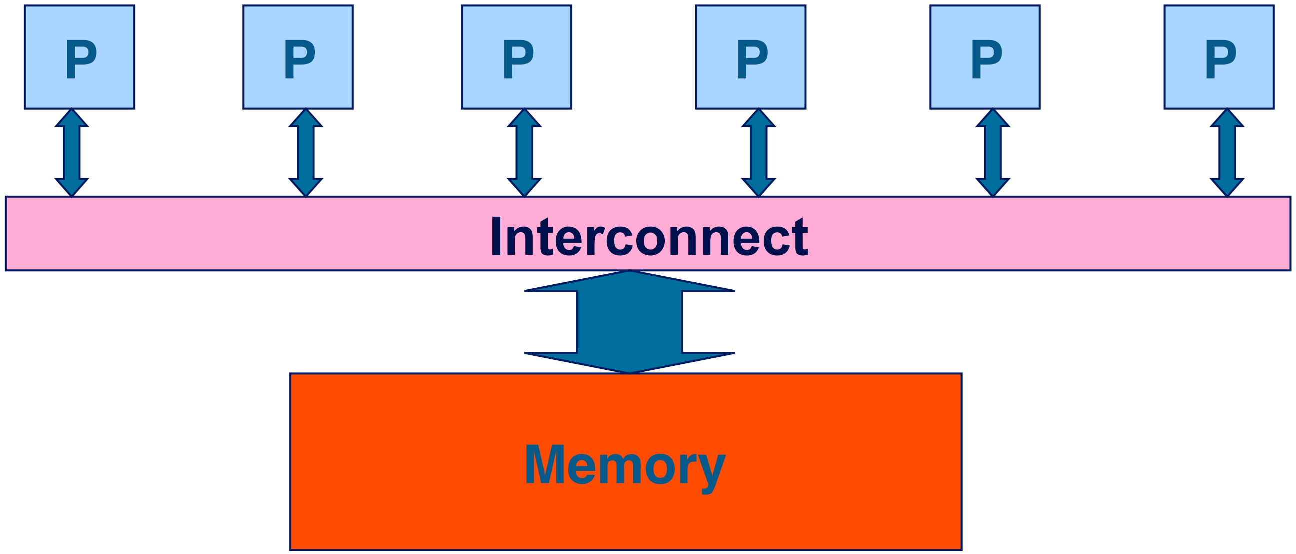
Shared memory systems

- Threaded programming is most often used on shared memory parallel computers.

- A shared memory computer consists of a number of processing units (CPUs) together with some memory

- Key feature of shared memory systems is a *single address space* across the whole memory system.
  - every CPU can read and write all memory locations in the system
  - one logical memory space
  - all CPUs refer to a memory location using the same address

Conceptual model

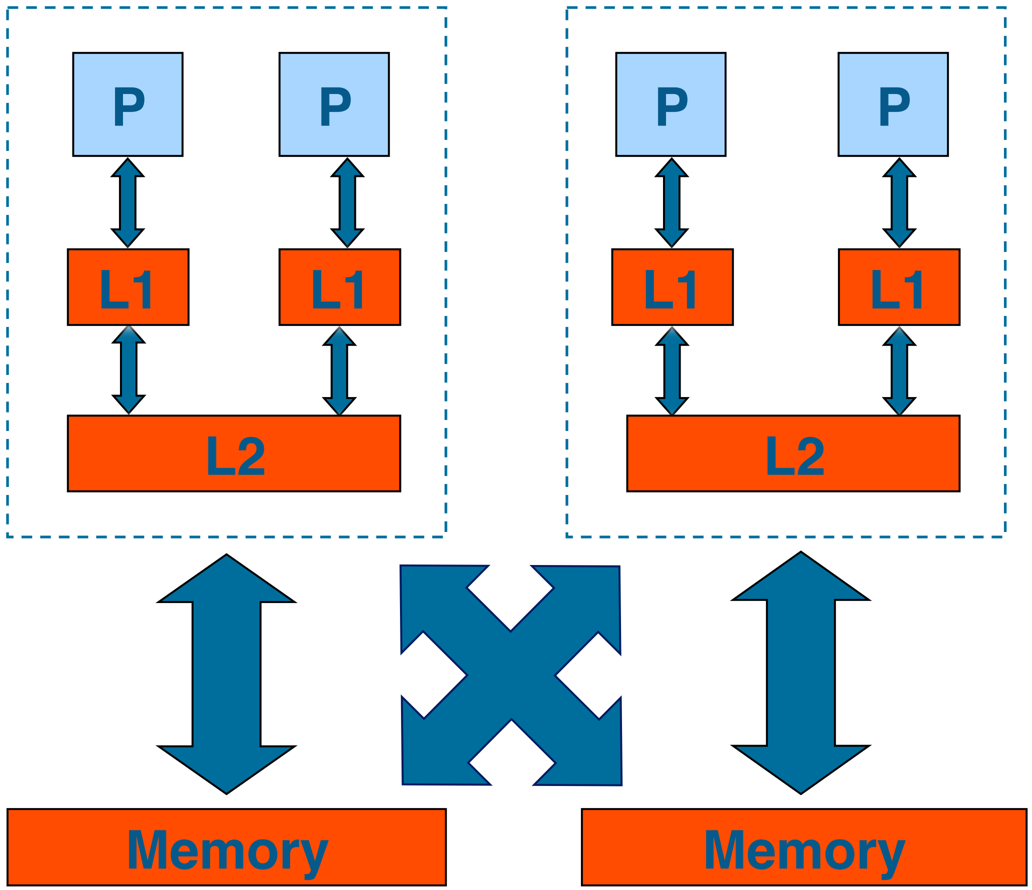


Real hardware

- Real shared memory hardware is more complicated than this…..
  - Memory may be split into multiple smaller units
  - There may be multiple levels of cache memory
    - some of these levels may be shared between subsets of processors
  - The interconnect may have a more complex topology

- ….but a single address space is still supported
  - Hardware complexity can affect performance of programs, but not their correctness

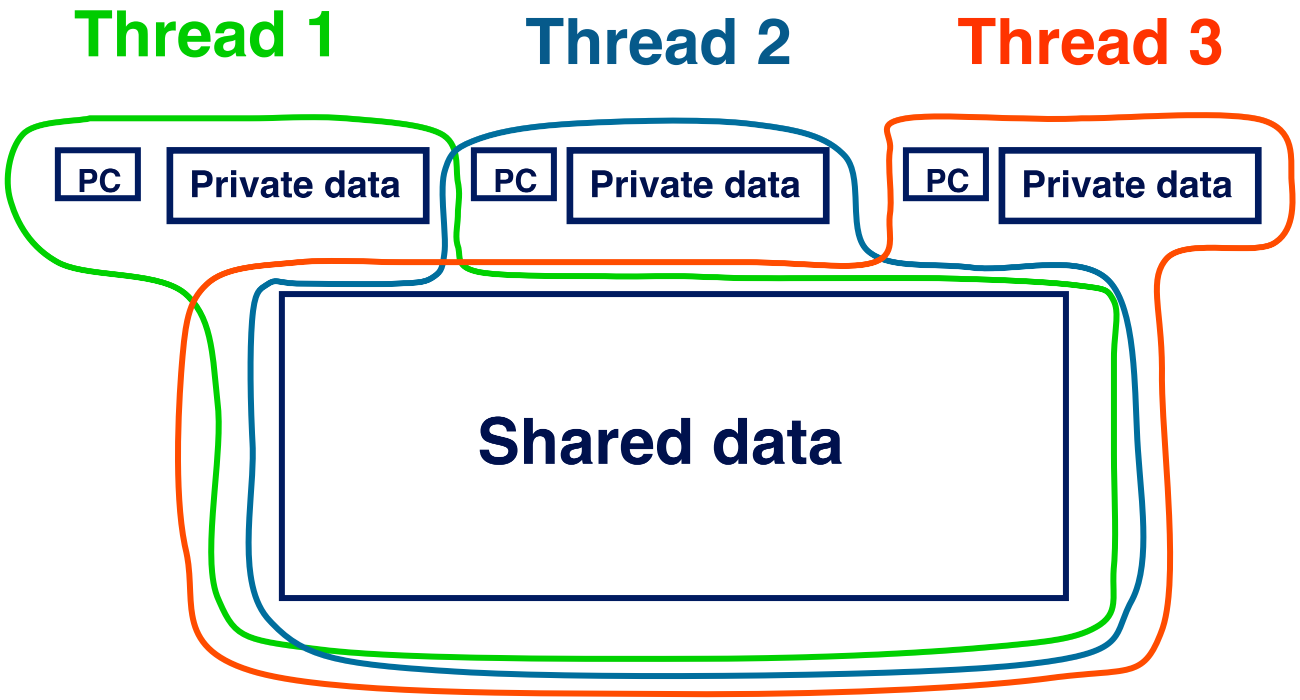
Real hardware example



Threaded Programming Model

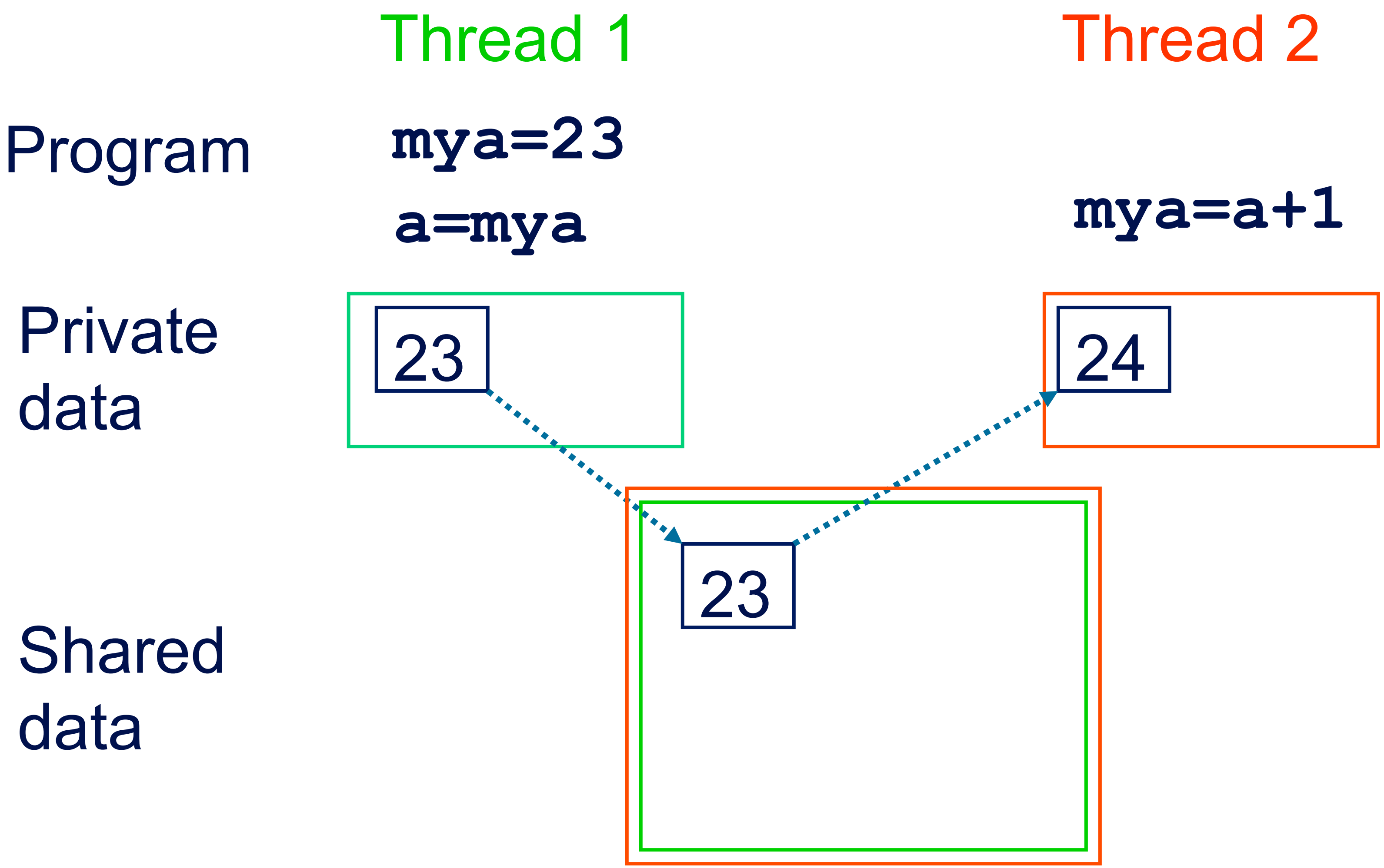
- The programming model for shared memory is based on the notion of threads
  - threads are like processes, except that threads can share memory with each other (as well as having private memory)
- Shared data can be accessed by all threads
- Private data can only be accessed by the owning thread
- Different threads can follow different flows of control through the same program
  - each thread has its own program counter
- Usually run one thread per CPU/core
  - but could be more
  - can have hardware support for multiple threads per core

Threads (cont.)



Thread communication

- In order to have useful parallel programs, threads must be able to exchange data with each other
- Threads communicate with each via reading and writing shared data
  - thread 1 writes a value to a shared variable A
  - thread 2 can then read the value from A
- Note: there is no notion of messages in this model


Thread 1
Thread 2
Program
mya=23
a=mya
mya=a+1
Private data
23
24
Shared data
23

Synchronisation

- By default, threads execute asynchronously
- Each thread proceeds through program instructions independently of other threads
- This means we need to ensure that actions on shared variables occur in the correct order: e.g.

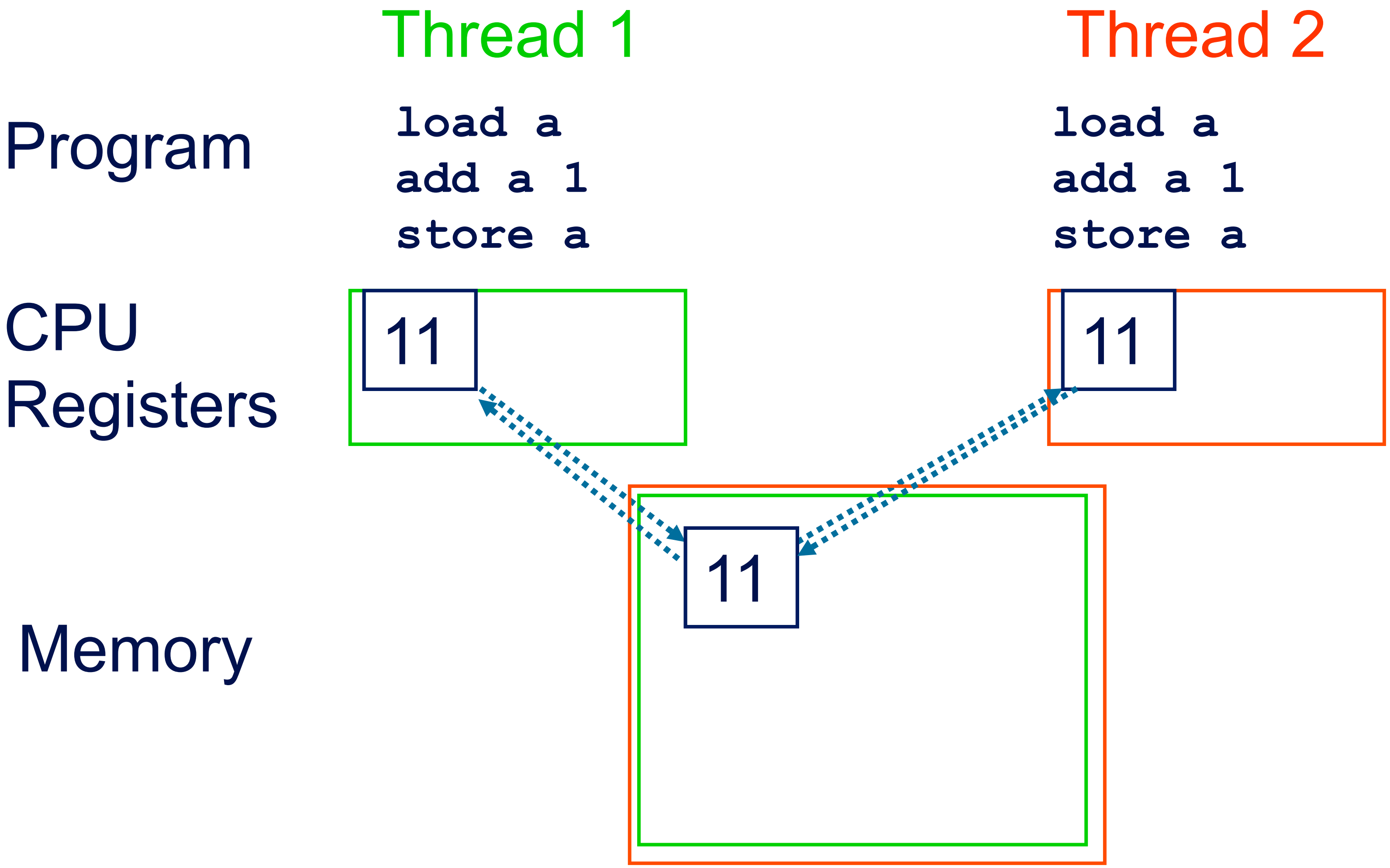
  thread 1 must write variable A before thread 2 reads it,

  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



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

```
for (i=0; i<100; i++){
    a[i] += b[i];
}
```

we could do iteration 0-49 on one thread and iterations 50-99 on the other.

- Can think of an iteration, or a set of iterations, as a task.

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.
- If the number of operations is much larger than the number of threads, most of the operations can proceed in parallel

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

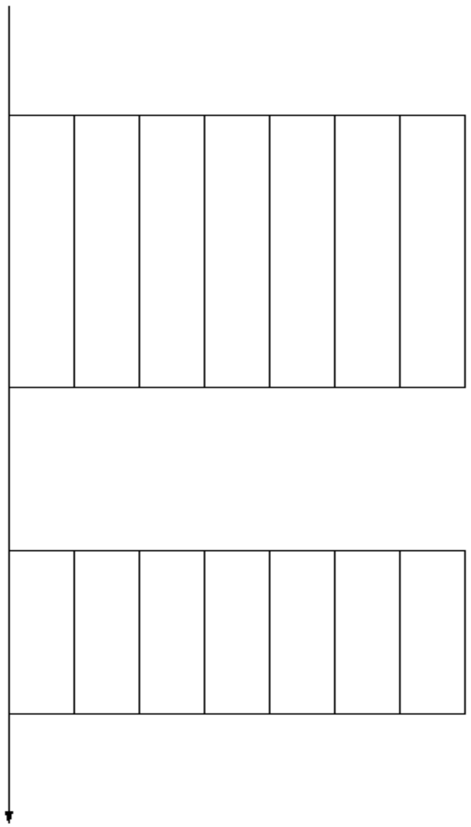
Sequential part

Parallel region

Sequential part

Parallel region

Sequential part



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

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

- 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. delimiting phases of computation
- Critical region
  - a section of code which only one thread at a time can enter
  - e.g.
- Atomic update
  - an update to a variable which can be performed only by one thread at a time
  - e.g. modification of shared variables
- Master region
  - a section of code executed by one thread only
  - e.g. initialisation, writing a file

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.
  - **`-mp`** for PGI pgcc/pgf90
  - **`-fopenmp`** for gcc/gfortran
  - **`-openmp`** for Intel 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

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`

- Can run just as you would a sequential program.

Parallel region directive

- Code within a parallel region is executed by all threads.
- Syntax:

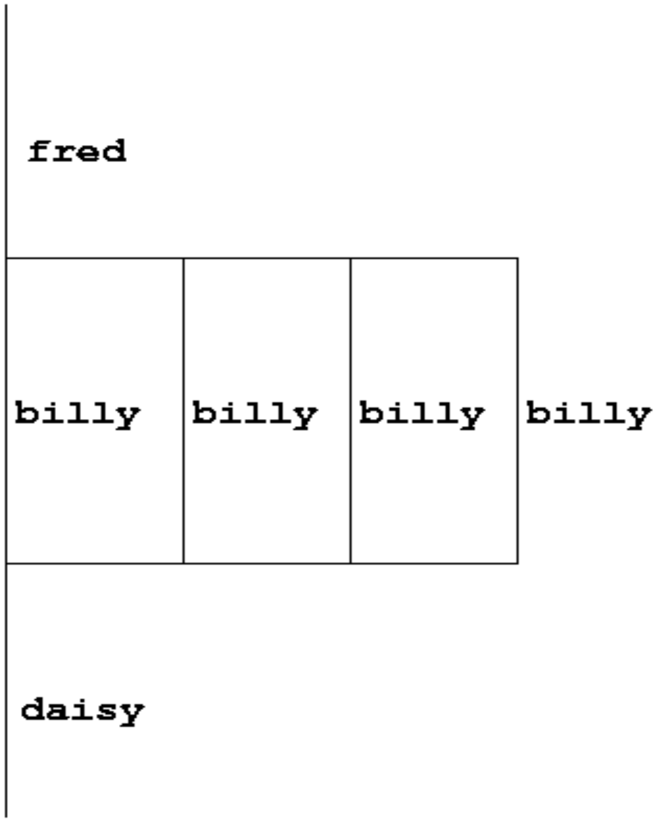
Fortran:
```
!$OMP PARALLEL
        block
!$OMP END PARALLEL
```

C/C++:
```
#pragma omp parallel
{
    block
}
```

Parallel region directive (cont)

Example:

```
fred();
#pragma omp parallel
{
   billy();
}
daisy();
```



Useful functions

- Often useful to find out number of threads being used.

Fortran:

```
USE OMP_LIB
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

C/C++:

```
#include <omp.h>
    int omp_get_num_threads(void);
```

- Important note: returns 1 if called outside parallel region!

Useful functions (cont)

- Also useful to find out number of the executing thread.

Fortran:

```
USE OMP_LIB
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

C/C++:

```
#include <omp.h>
    int omp_get_thread_num(void)
```

- Takes values between 0 and **`OMP_GET_NUM_THREADS()`** - 1

Clauses

- Specify additional information in the parallel region directive through *clauses*:

Fortran : **!$OMP PARALLEL** *[clauses]*

C/C++: **#pragma omp parallel** *[clauses]*

- Clauses are comma or space separated in Fortran, space separated in C/C++.

Shared and private variables

- Inside a parallel region, variables can be either shared (all threads see same copy) or private (each thread has its own copy), or reduction (see later)
- Shared, private and default clauses

Fortran: **SHARED(***list***)**

PRIVATE(*list***)**

DEFAULT(SHARED|PRIVATE|NONE)

C/C++: **shared(***list***)**

private(*list***)**

default(shared|none)

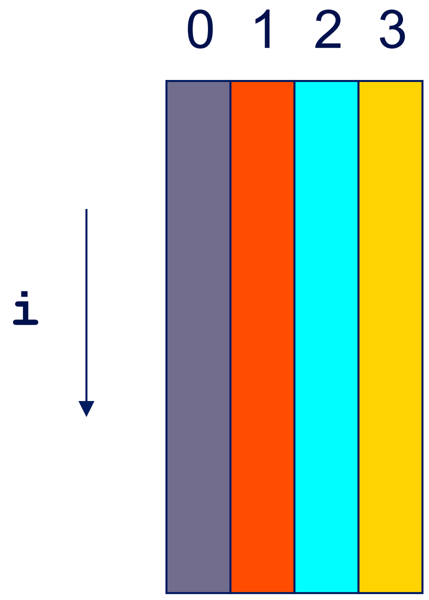
Shared and private (cont.)

- On entry to a parallel region, private variables are uninitialised.
- Variables declared inside the scope of the parallel region are automatically private.
- After the parallel region ends the original variable is unaffected by any changes to private copies.
- Not specifying a DEFAULT clause is the same as specifying DEFAULT(SHARED)
  - Danger!
  - Always use DEFAULT(NONE)

Shared and private (cont)

Example: each thread initialises its own column of a shared array:

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
      myid = omp_get_thread_num() + 1
      do i = 1,n
         a(i,myid) = 1.0
      end do
!$OMP END PARALLEL
```



Reductions

- A *reduction* produces a single value from associative operations such as addition, multiplication,max, min, and, or.
- Would like each thread to reduce into a private copy, then reduce all these to give final result.
- Use REDUCTION clause:

Fortran: **`REDUCTION(`***op***`:`***list***`)`**

C/C++: **`reduction(`***op***`:`***list***`)`**

- Can have reduction arrays in Fortran, but not in C/C++

Reductions (cont.)

Example:

```
b = 10;
#pragma omp parallel reduction(+:b) private(myid)
{
  myid = omp_get_thread_num();
  for (int i=0;i<n;i++) {
      b+=c[myid][i];
  }
}
a=b;
```

Value in original variable is saved

Each thread gets a private copy of **b**, initialised to 0

All accesses to **b** inside the parallel region are to the private copies

At the end of the parallel region, all the private copies are added into the original variable

Parallel loops

- Loops are the most common source of parallelism in most codes. Parallel loop directives are therefore very important!
- A parallel do/for loop divides up the iterations of the loop between threads.
- The loop directive appears inside a parallel region and indicates that the work should be shared out between threads, instead of replicated
- There is a synchronisation point at the end of the loop: all threads must finish their iterations before any thread can proceed

Parallel do/for loops (cont)

Syntax:

Fortran:

```
!$OMP DO [clauses]
      do loop
[ !$OMP END DO ]
```

C/C++:

```
#pragma omp for [clauses]
      for loop
```

Restrictions in C/C++

- Because the for loop in C is a general while loop, there are restrictions on the form it can take.

- It has to have determinable trip count - it must be of the form:

for (var = a; var *logical-op* **b;** *incr-exp***)**

where *logical-op* is one of **<, <=, >, >=**

and *incr-exp* is **var = var +/- incr** or semantic equivalents such as **var++.**

Also cannot modify **var** within the loop body.

Parallel do loops (example)

Example:

```
!$OMP PARALLEL
!$OMP DO
   do i=1,n
      b(i) = (a(i)-a(i-1))*0.5
   end do
!$OMP END DO
!$OMP END PARALLEL
```

```
#pragma omp parallel
{
#pragma omp for
  for (int i=0;i<n;i++){
    b[i] = (a[i]*a[i-1])*0.5
  }
}
```

Parallel do/for loops (cont)

- With no additional clauses, the DO/FOR directive will partition the iterations as equally as possible between the threads.

- However, this is implementation dependent, and there is still some ambiguity:

e.g. 7 iterations, 3 threads. Could partition as 3+3+1 or 3+2+2

SCHEDULE clause

- The SCHEDULE clause gives a variety of options for specifying which loops iterations are executed by which thread.
- Syntax:

Fortran: **SCHEDULE (***kind[, chunksize]***)**

C/C++: **schedule (***kind[, chunksize]***)**

where *kind* is one of

STATIC, **DYNAMIC**, **GUIDED**, **AUTO** or **RUNTIME**

and *chunksize* is an integer expression with positive value.

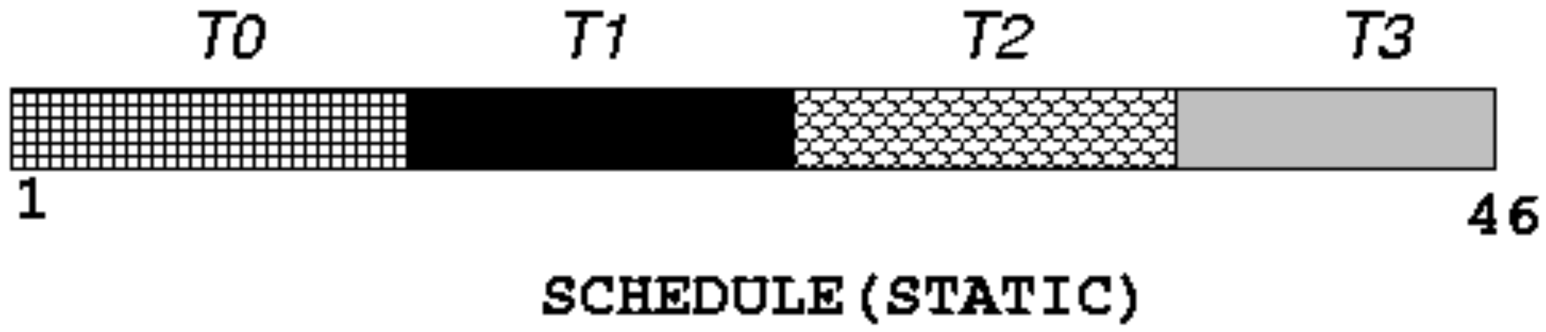
- E.g. **!$OMP DO SCHEDULE(DYNAMIC,4)**

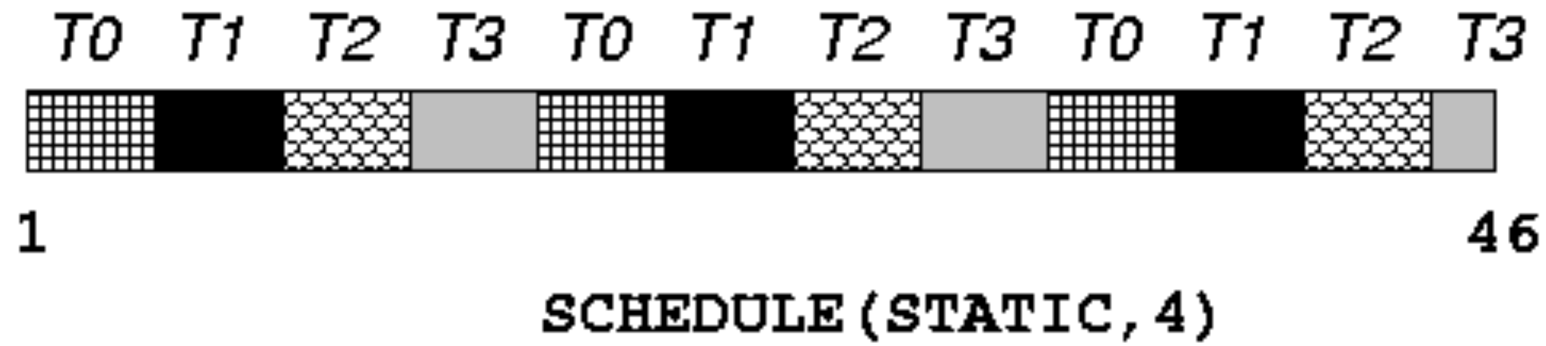
STATIC schedule

- With no *chunksize* specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread in order (**block** schedule).

- If *chunksize* is specified, the iteration space is divided into chunks, each of *chunksize* iterations, and the chunks are assigned cyclically to each thread in order (**block cyclic** schedule)

STATIC schedule





DYNAMIC schedule

- DYNAMIC schedule divides the iteration space up into chunks of size *chunksize*, and assigns them to threads on a first-come-first-served basis.

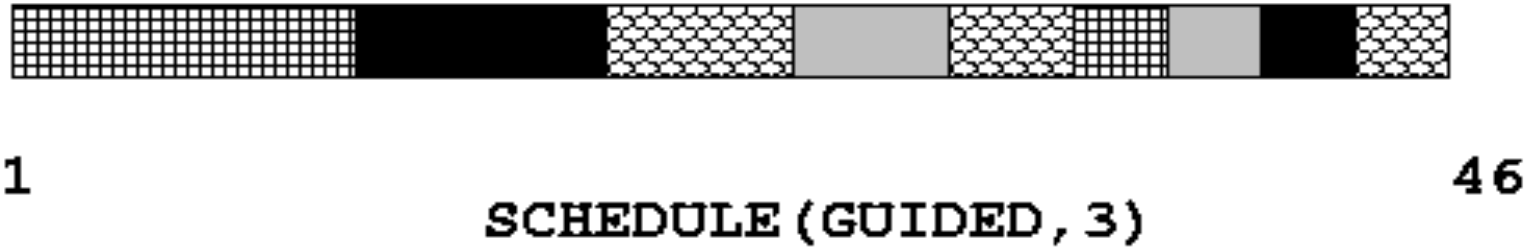
- i.e. as a thread finish a chunk, it is assigned the next chunk in the list.

- When no *chunksize* is specified, it defaults to 1.

GUIDED schedule

- GUIDED schedule is similar to DYNAMIC, but the chunks start off large and get smaller exponentially.
- The size of the next chunk is proportional to the number of remaining iterations divided by the number of threads.
- The *chunksize* specifies the minimum size of the chunks.
- When no *chunksize* is specified it defaults to 1.

DYNAMIC and GUIDED schedules



AUTO schedule

- Lets the runtime have full freedom to choose its own assignment of iterations to threads
- If the parallel loop is executed many times, the runtime can evolve a good schedule which has good load balance and low overheads.

Choosing a schedule

When to use which schedule?

- STATIC best for load balanced loops - least overhead.
- STATIC,*n* good for loops with mild or smooth load imbalance, but can induce overheads.
- DYNAMIC useful if iterations have widely varying loads, but ruins data locality.
- GUIDED often less expensive than DYNAMIC, but beware of loops where the first iterations are the most expensive!
- AUTO may be useful if the loop is executed many times over

Exercise

Area of the Mandelbrot set

- Aim: introduction to using parallel regions and loops.
- Estimate the area of the Mandelbrot set by Monte Carlo sampling.
  - Generate a grid of complex numbers in a box surrounding the set
  - Test each number to see if it is in the set or not.
  - Ratio of points inside to total number of points gives an estimate of the area.
  - Testing of points is independent