# Parallel Algorithms on a cluster of PCs

Ian Bush
Computational Science & Engineering Department
Daresbury Laboratory
I.J.Bush@dl.ac.uk

(With thanks to Lorna Smith and Mark Bull at EPCC)

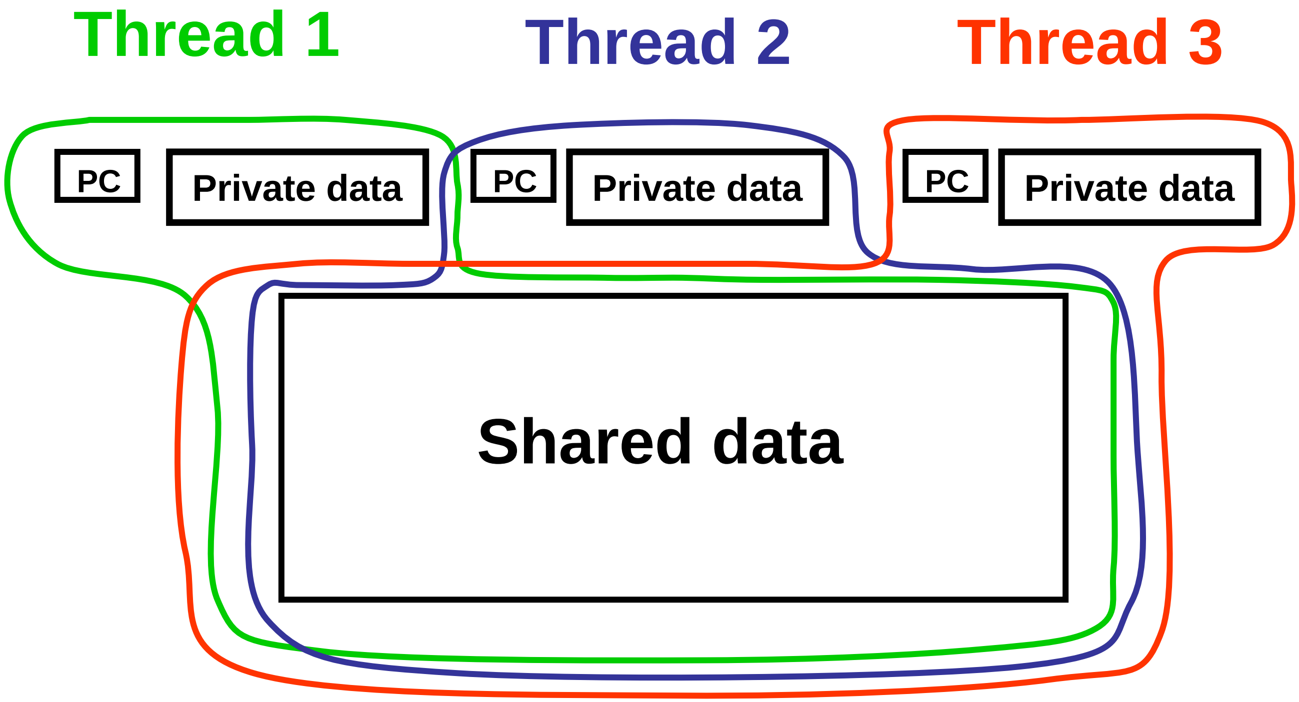
# Overview

- This lecture will cover
  - **shared variables model**
    - *threads*
    - *synchronisation*
    - *shared and private data*
  - **A very brief introduction to OpenMP**

## Shared Variables Model

- Shared variable programming model 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**

# Threads


Thread 1
Thread 2
Thread 3
PC
Private data
PC
Private data
PC
Private data
Shared data

# More About Threads

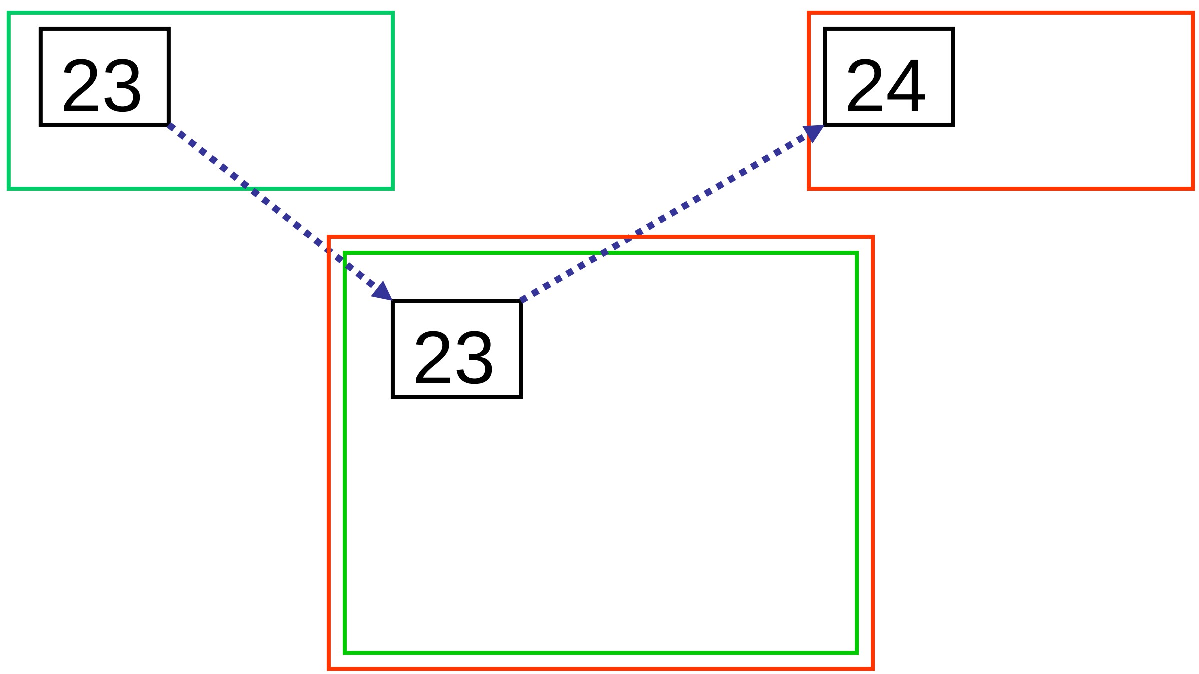
- Often uses SPMD
  - **all threads execute same program**
  - **each thread has its own identifier**
- Usually run one thread per processor
  - **but could be more**
- Threads communicate with each other only via shared data (no messages!)
  - **thread 1 writes a value to a shared variable A**
  - **thread 2 can then read the value from A**

## Thread Communication

| | Thread 1 | Thread 2 |
|---|---|---|
| Program | `mya=23`<br>`a=mya` | `mya=a+1` |

Private data

Shared data



# Synchronisation

- Threads execute their programs asynchronously
- Writes and reads of shared data are always non-blocking
  - **need some mechanisms to ensure that these actions occur in the correct order**
- In previous example
  - **write of *a* must occur before the read**
  - **may also require read before write**

# Synchronisation Concepts

- Most common constructs are:
  - **Master section**
    - a section of code executed by one thread only
    - e.g. initialisation, writing a file
  - **Barrier**
    - all threads must arrive at a barrier before any thread can proceed past it
    - e.g. delimiting phases of computation (e.g. a timestep)
  - **Critical section**
    - only one thread at a time can enter a section of code
    - e.g. modification of shared variables

# Summary of Shared Variables

- Shared Variables
  - code is executed by independent threads
  - each can access the same memory space
  - can have private data as well
  - need synchronization to ensure correctness

# Message Passing compared to Shared Variables

- Maps closely to highly scalable architectures.
- Can be easier to debug
  - **Harder to induce non-deterministic behaviour**
  - **But far from impossible**
- Easier to find causes of poor performance (communication is explicit)
- Can overlap communication and computation
- Naturally minimises synchronisation

## Shared Variables Compared to Message Passing

- Easier to program than message passing
  - **Maybe ….**
- Implementation can be incremental
  - **More easily than message passing**
- No message start-up costs as no messages
  - **But shared memory can mean that loads and stores become very expensive**
  - **False sharing**
  - **Extra synchronization**
- Can cope with irregular / data dependent communication patterns
- Load balancing more straightforward
  - **Finer grained parallelism more straightforward**
- More often that not run serial it really is a serial code

# But we need more

Shared variables allow a very simple communication method, you simply assign as you want. However How to decide which thread does which work? In message passing this is simple – you can only work on your own data. For threads we could

- Let the compiler decide by itself
  - **In practice does not very successful. It is very difficult to work out all data dependencies:**
- Give the compiler hints
  - **E.g. tell it in the above that indx( i ) contains unique values**
  - **This is where OpenMP comes in**
  - **c.f. vectorisation ( if you remember that …. )**

## Brief history of OpenMP

- Historical lack of standardisation in shared memory directives. Each vendor did their own thing.
- Previous attempt (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 also supported by HP, Sun and ASCI programme.
- 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.

# Overview of OpenMP

- OpenMP is a set of extensions to Fortran and C/C++ which implements the shared variables model.
- Based on compiler directives, together with library routines and environment variables.
- Available on most single address space machines.
- Industry standard supported by most major vendors.

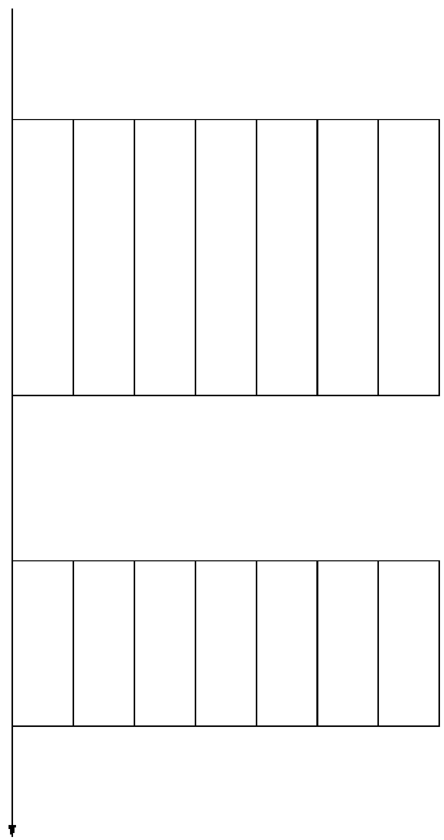
## Directives and sentinels

- A directive is a special line of source code with meaning only to a compiler that understands it.
- A directive is distinguished by a sentinel at the start of the line.
- OpenMP sentinels are:
  - **Fortran:** `!$OMP` **(or** `C$OMP` **or** `*$OMP`**)**
  - **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)
  - **Typically how many set by the OMP_NUMTHREADS environment variable**
- 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
  - **Note implied synchronization**

# Parallel region



```
      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.

- N.B. It is up to YOU to ensure the iterations are independent, NOT the compiler

# 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.

- And it is up to YOU to ensure this

## Synchronisation example

Thread 1 | Thread 2

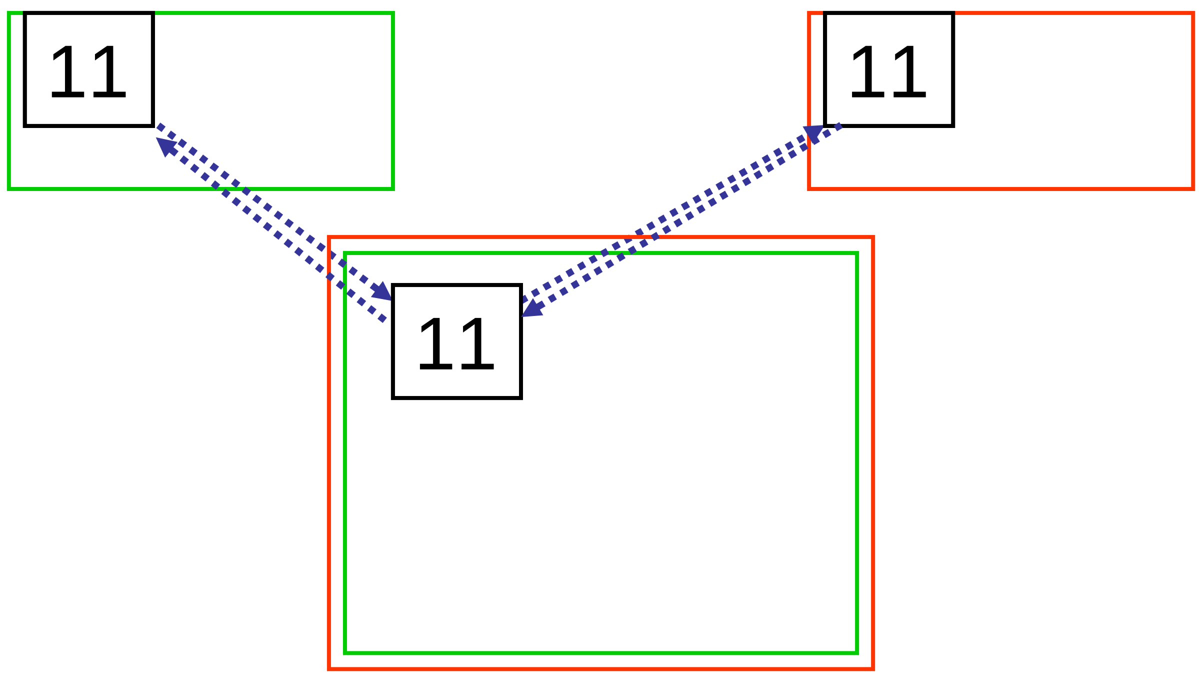
Program

```
load a
add a 1
store a
```

```
load a
add a 1
store a
```

Private data

Shared data



# 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 = 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.

# Parallel region directive

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

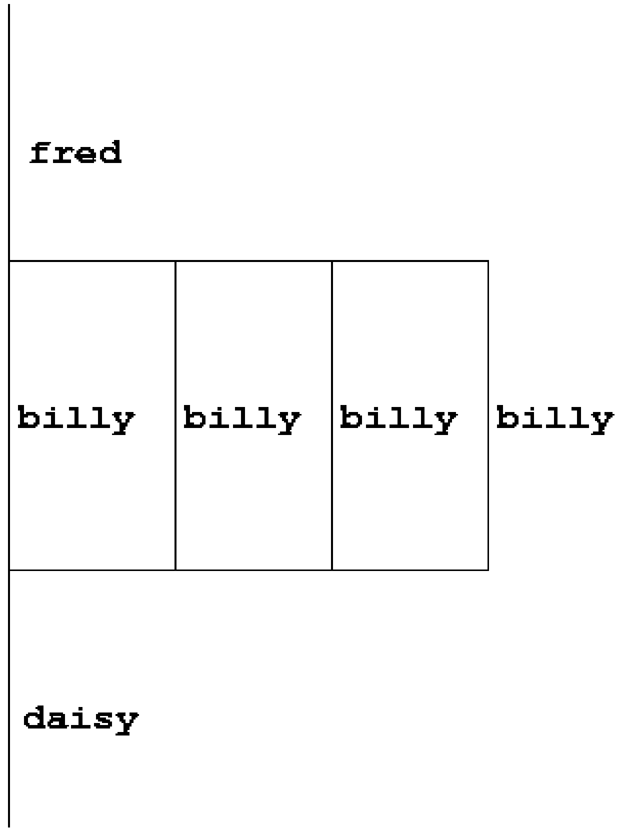
Fortran:
```
!$OMP PARALLEL
      block
!$OMP END PARALLEL
```

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

# Parallel region directive (cont)

Example:

```
      call fred()
!$OMP PARALLEL
      call billy()
!$OMP END PARALLEL
      call daisy()
```



# Useful functions

- Often useful to find out number of threads being used.
  - Fortran:
    - `INTEGER FUNCTION OMP_GET_NUM_THREADS()`
  - C/C++:
    - `#include <omp.h>`

    `int omp_get_num_threads(void);`

- Note: returns 1 if called outside parallel region!

# Useful functions (cont)

- Also useful to find out number of the executing thread.
  - Fortran:
    - `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++.

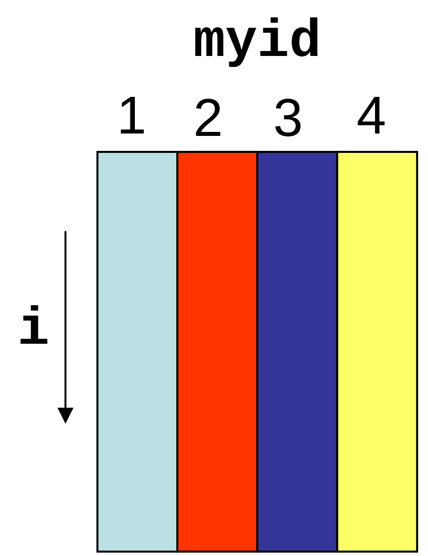
# Shared and private variables

- Inside a parallel region, variables can be either shared (all threads see same copy) or private (each thread has private copy).
- Shared, private and default clauses
  - **Fortran:**
    - ***SHARED(*** list ***)***
    - ***PRIVATE(*** list ***)***
    - ***DEFAULT(SHARED|PRIVATE|NONE)***
  - **C/C++:**
    - ***shared(*** list ***)***
    - ***private(*** list ***)***
    - ***default(shared|none)***
  - **Strongly recommend 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***`)`
- N.B. Cannot have reduction arrays, only scalars or array elements!

# Reductions (cont.)

Example:

```
!$OMP PARALLEL DEFAULT(NONE), REDUCTION(+:B),
!$OMP& PRIVATE(I,MYID), SHARED(C,N)
      myid = omp_get_thread_num() + 1
      do i = 1,n
         b = b + c(i,myid)
      end do
!$OMP END PARALLEL
```

# Work sharing directives

- Directives which appear inside a parallel region and indicate how work should be shared out between threads
  - **Parallel do/for loops**
  - **Parallel sections**
  - **‘One thread only’ directives**

# Parallel do 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.

# Parallel do/for loops (cont)

Syntax:

Fortran:

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

C/C++:

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

## Parallel do/for loops (cont)

- With no additional clauses, the DO/FOR directive will usually 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

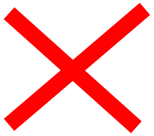
## Parallel do/for loops (cont)

- If you tell the compiler that the loop should be parallelised it will parallelise it !
  - **It is up to you to be sure**
  - **You may have more information than the compiler can see, e.g. an indexing array does not have repeated values**
- How can you tell if a loop is parallel or not?
  - **Useful test: if the loop gives the same answers if it is run in reverse order, then it is almost certainly parallel**
- Jumps out of the loop are not permitted.

# Parallel do/for loops (cont)

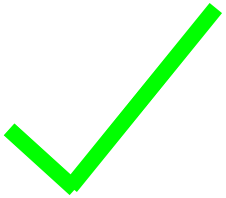
1.

```
do i=2,n
      a(i)=2*a(i-1)
end do
```



2.

```
do i=1,n
    b(i)= (a(i)-a(i-1))*0.5
end do
```



# Parallel do loops (example)

Example:

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I),
!$OMP& SHARED(A,B,N)
!$OMP DO
      do i=1,n
         b(i) = (a(i)-a(i-1))*0.5
      end do
!$OMP END DO
!$OMP END PARALLEL
```

# 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` **or** `RUNTIME` **and *chunksize* is an integer expression with positive value.**

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

# Synchronization

Recall:

- Need to synchronise actions on shared variables.
- Need to respect dependencies.
- Need to protect updates to shared variables (not atomic by default)

# BARRIER directive

- No thread can proceed past a barrier until all the other threads have arrived.
-
- Note that there is an implicit barrier at the end of DO/FOR, SECTIONS and SINGLE directives.

- Syntax:
  - **Fortran:**
    - `!$OMP BARRIER`
  - **C/C++:**
    - `#pragma omp barrier`

- Either all threads or none must encounter the barrier: otherwise DEADLOCK!!

## BARRIER directive (cont)

Example:

```
!$OMP PARALLEL DEFAULT(NONE), PRIVATE(I,MYID),
!$OMP& SHARED(A,B,C,NEIGHB)
      myid = omp_get_thread_num()
      a(myid) = a(myid)*3.5
!$OMP BARRIER
      b(myid) = a(neighb(myid)) + c
!$OMP END PARALLEL
```

- Barrier required to force synchronisation on ***a***

## Critical sections

- A critical section is a block of code which can be executed by only one thread at a time.
- Can be used to protect updates to shared variables.
- The CRITICAL directive allows critical sections to be named.
- If one thread is in a critical section with a given name, no other thread may be in a critical section with the same name, though they can be in critical sections with other names.

# CRITICAL directive

- Syntax:
  - **Fortran:**
    - **`!$OMP CRITICAL`** [**(** name **)**]

      *block*

      **`!$OMP END CRITICAL`** *[**(** name **)**]*
  - **C/C++:**
    - **`#pragma omp critical`** [**(** name **)**]

      *structured block*

- In Fortran, the names on the directive pair must match.

- If the name is omitted, a null name is assumed (all unnamed critical sections effectively have the same null name).

## CRITICAL directive (cont)

Example:

```
!$OMP PARALLEL DEFAULT(NONE),
!$OMP& SHARED(STACK),PRIVATE(INEXT,INEW)
!$OMP CRITICAL (STACKPROT)
      inext = getnext(stack)
!$OMP END CRITICAL (STACKPROT)
      call work(inext,inew)
!$OMP CRITICAL (STACKPROT)
      if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL (STACKPROT)
!$OMP END PARALLEL
```

## Other features

- Loads of other clauses on the directives so far considered
- Atomic directive: Ensure only one thread updates a global variable
- THREADPRIVATE directive: private copies of global variables.
- NOWAIT clause to suppress barriers
- Lock routines.
- Ordered sections in parallel loops.
- Directives can be *orphaned* - they can appear in subroutines called from inside a parallel region.
- Environment variables for setting number of threads, etc.
- Nested parallelism.
- Conditional compilation.

…….

## OpenMP resources

- Official web site: **`www.openmp.org`**
  - **Language specifications, links to compilers and tools, mailing list.**
- Kuck and Associates: **`www.kai.com`**
  - **Compiler and tool vendors**
- Microbenchmarks: **`www.epcc.ed.ac.uk/research/openmpbench`**
- Book: “Parallel Programming in OpenMP”, Dagum et. al., Academic Press, ISBN 1558606718.