# CS 426

## Parallel Programming with OpenMP

# Overview

- Introduction to OpenMP
- Parallel regions
- Worksharing directives
- More about parallel loops
- Synchronization
- Additional features
- OpenMP 2.0

# 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 includes all major vendors.
- 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.
  - Major revision (3.0) in May 2008.
  - Minor revision (3.1) in July 2011.

# 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

- Book: “Parallel Programming in OpenMP”, Chandra et. al., Morgan Kaufmann, ISBN 1558606718.

# Resources

- LLNL offers an online OpenMP tutorial
  - https://computing.llnl.gov/tutorials/openMP/
- Intel's OpenMP tutorials
  - *http://software.**intel**.com/file/24569*

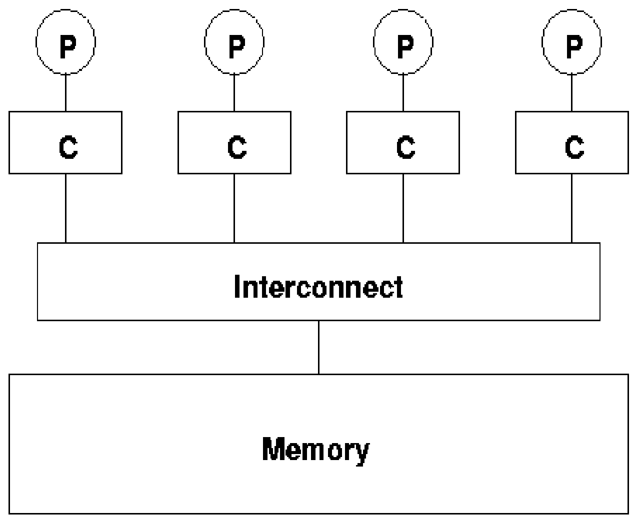
# Shared memory systems

- OpenMP is designed for programming shared memory parallel computers.

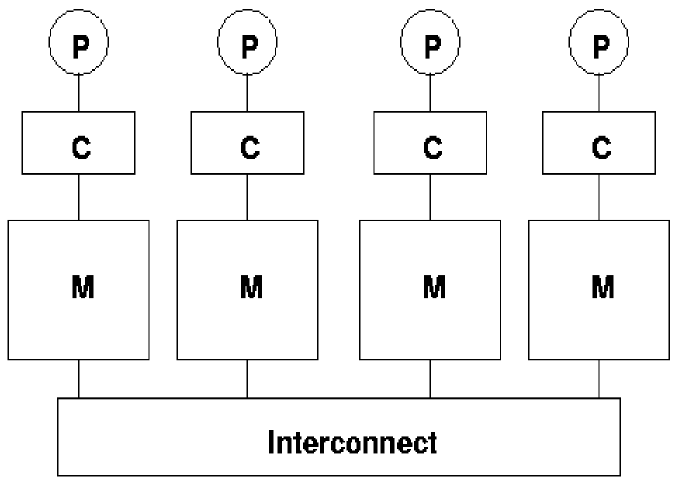
- Key feature is a *single address space* across the whole memory system.
  - every processor can read and write all memory locations
  - one logical memory space

- Two main types of hardware:
  - true shared memory
  - distributed shared memory

# Shared memory systems (cont)




True shared memory



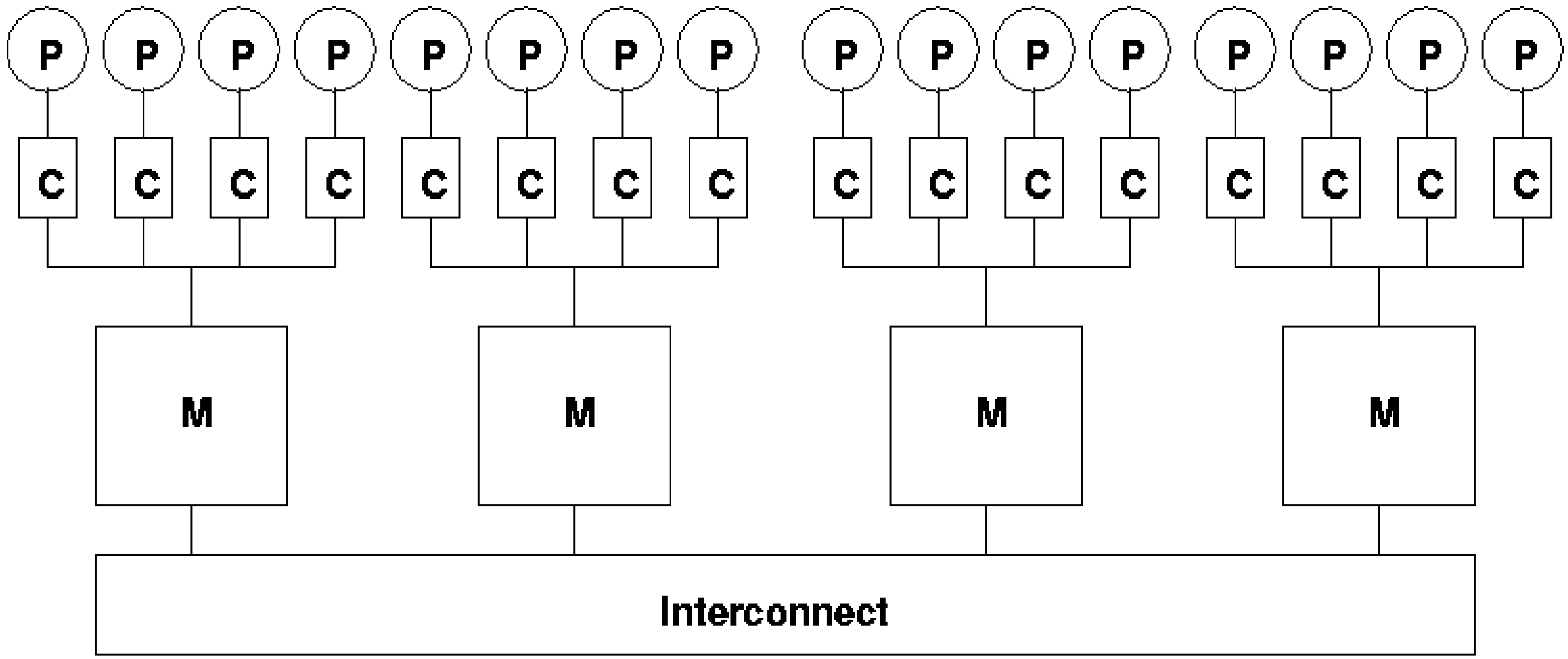

Distributed shared memory

Sun Enterprise/SunFire, Cray SV1, Compaq ES, multiprocessor PCs, nodes of IBM SP, NEC SX5

None!

# Shared memory systems (cont)

- Most distributed shared memory systems are clustered:

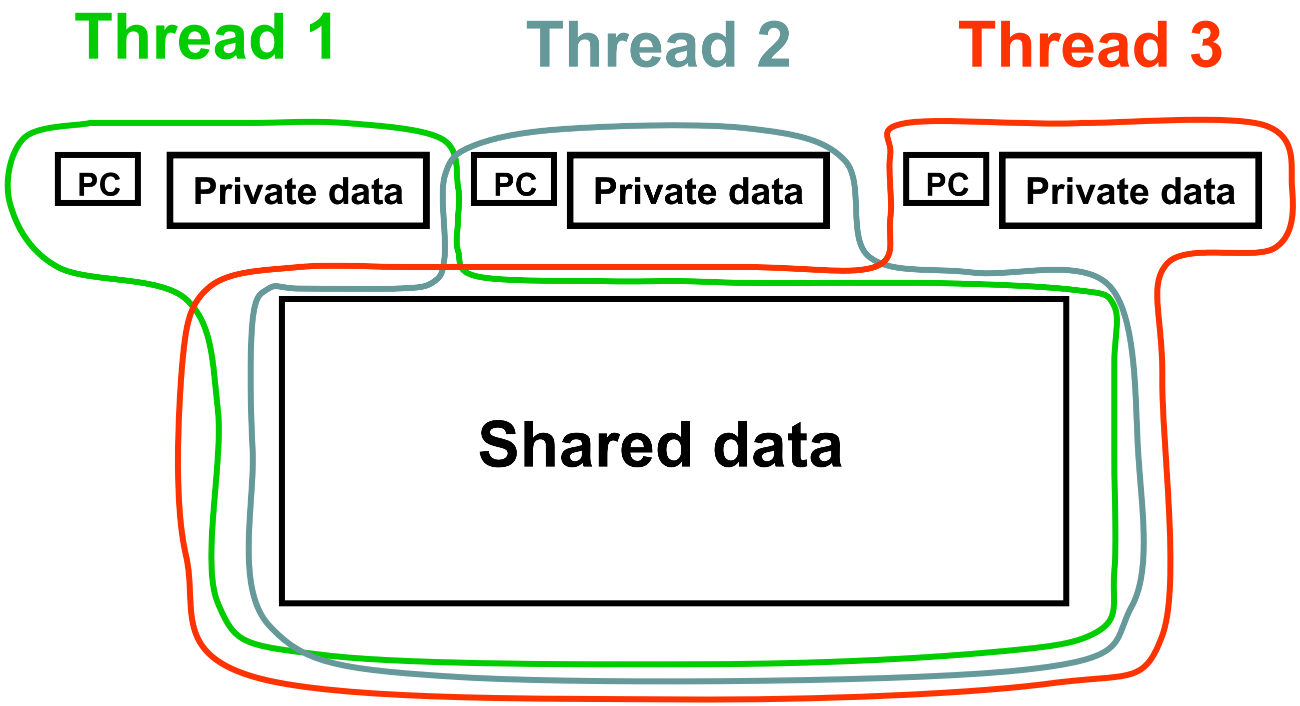



**Clustered distributed shared memory**
SGI Origin, HP Superdome, Compaq GS

# Threads and thread teams

- A thread is a (lightweight) process - an instance of a program + its data.
- Each thread can follow its own flow of control through a program.
- Threads can share data with other threads, but also have private data.
- Threads communicate with each other via the shared data.
- A *thread team* is a set of threads which co-operate on a task.
- The *master thread* is responsible for co-ordinating the team.

# Threads (cont.)



# 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`**)
  - C/C++: **`#pragma omp`**

# 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.

# Synchronization

- 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!

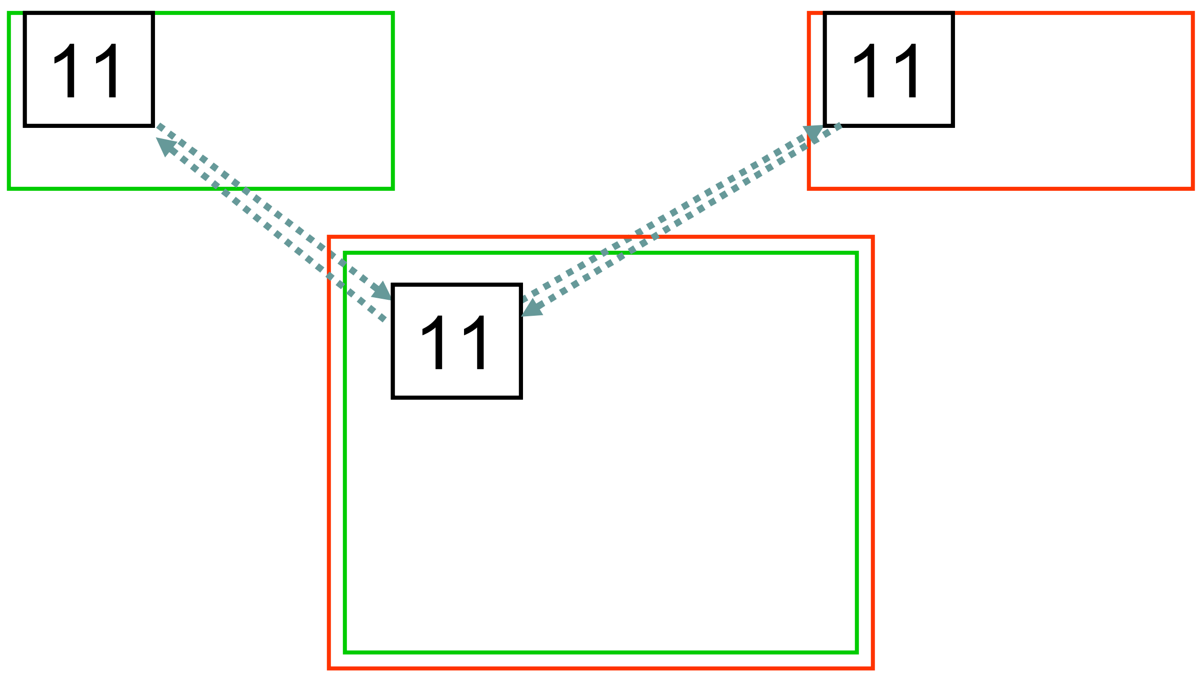
If two threads try to do this at the same time, one of the updates may get overwritten.

# Synchronization example

| | Thread 1 | Thread 2 |
|---|---|---|
| Program | `load a`<br>`add a 1`<br>`store a` | `load a`<br>`add a 1`<br>`store a` |

Private data

Shared data



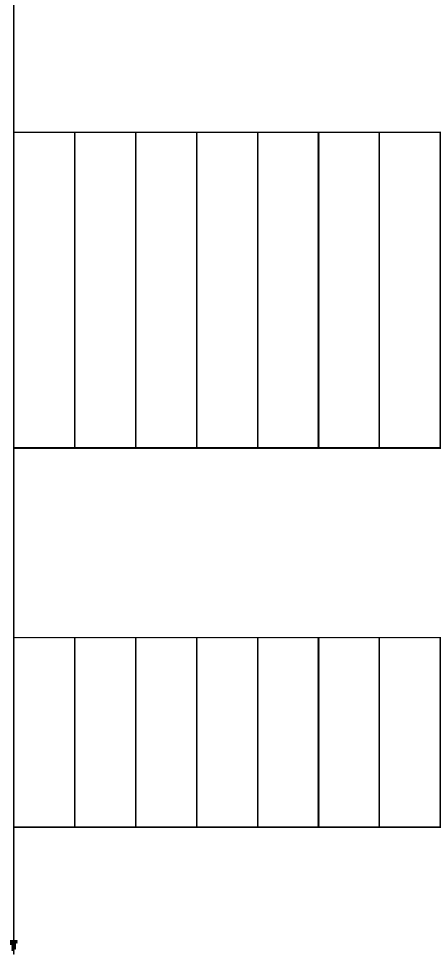
# CS 426

## Parallel Regions

# 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



```
      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 region directive

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

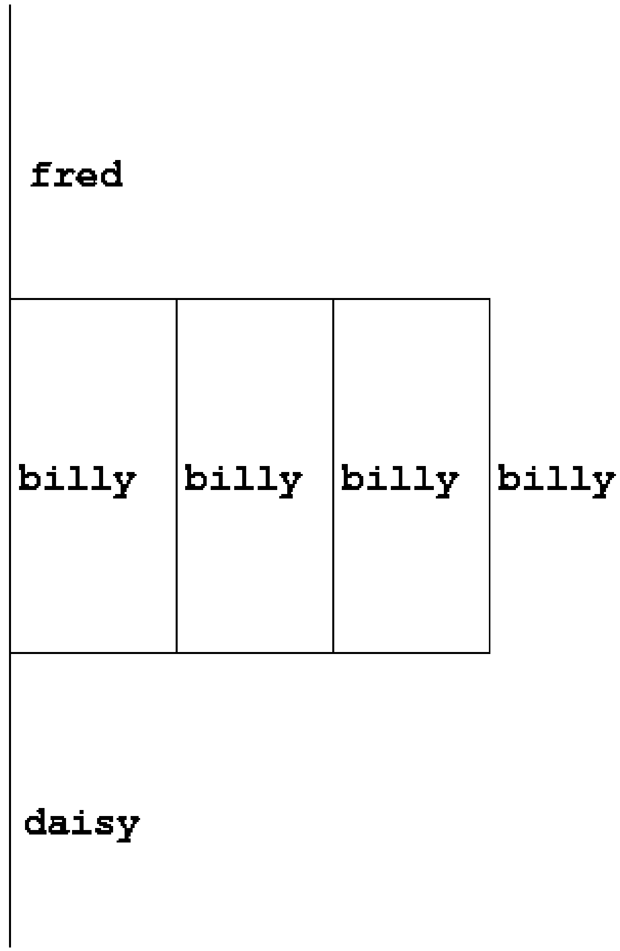
Fortran:
```
!$OMP PARALLEL
        block
!$OMP END PARALLEL
```

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);
```

- Important 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 its own copy).
- Shared, private and default clauses

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

**PRIVATE(***list***)**

**DEFAULT(SHARED|PRIVATE|NONE)**

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

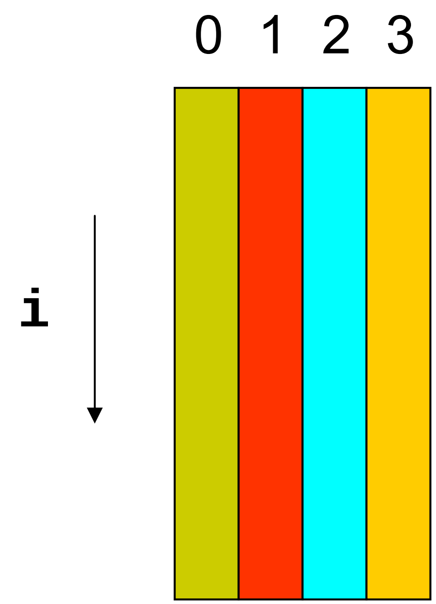
**private(***list***)**

**default(shared|none)**

# Shared and private (cont)

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
```



# Shared and private (cont)

- How do we decide which variables should be shared and which private?
  - Most variables are shared
  - Loop indices are private
  - Loop temporaries are private
  - Read-only variables - shared
  - Main arrays - shared
  - Write-before-read scalars - usually private
  - Sometimes either is semantically OK, but there may be performance implications in making the choice.

# Multi-line directives

- Fortran: fixed source form

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID),
!$OMP& SHARED(A,N)
```

- Fortran: free source form

```
!$OMP PARALLEL DEFAULT(NONE),PRIVATE(I,MYID), &
!$OMP SHARED(A,N)
```

- C/C++:

```
#pragma omp parallel default(none) \
private(i,myid) shared(a,n)
```

# Initializing private variables

- Private variables are uninitialized at the start of the parallel region.

- If we wish to initialize them, we use the FIRSTPRIVATE clause:

Fortran: **`FIRSTPRIVATE(`***list***`)`**

C/C++: **`firstprivate(`***list***`)`**

# Initializing private variables (cont)

Example:

```
      b = 23.0;
       . . . . .
#pragma omp parallel firstprivate(b), private(i,myid)
   {
      myid = omp_get_thread_num();
      for (i=0; i<n; i++){
         b += c[myid][i];
      }
      c[myid][n] = b;
   }
```

# 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.

# Reductions (cont.)

- Use REDUCTION clause:

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

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

- Cannot have reduction arrays, only scalars or array elements! (except in Fortran 2.0)

# Reductions (cont.)

Example:

```
      b = 0
!$OMP PARALLEL REDUCTION(+:b),
!$OMP& PRIVATE(I,MYID)
      myid = omp_get_thread_num() + 1
      do i = 1,n
         b = b + c(i,myid)
      end do
!$OMP END PARALLEL
```

# IF clause

- We can make the parallel region directive itself conditional.
- Can be useful if there is not always enough work to make parallelism worthwhile.

Fortran: **`IF (`***scalar logical expression***`)`**

**`C/C++`**: **`if (`***scalar expression***`)`**

# IF clause (cont.)

Example:

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

# CS 426

## Work sharing directives

# 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.
- We will just introduce the basic form here: more details will follow in the next session.

# 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/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)

- 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.

e.g.

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

# Parallel do/for loops (cont)

2.

```
ix = base
do i=1,n
  a(ix) = a(ix)*b(i)
  ix  = ix + stride
end do
```

3.

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

# 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
```

# Parallel DO/FOR directive

- This construct is so common that there is a shorthand form which combines parallel region and DO/FOR directives:

Fortran:

**`!$OMP PARALLEL DO`** *[clauses]*

*do loop*

*[* **`!$OMP END PARALLEL DO`** *]*

C/C++:

**`#pragma omp parallel for`** *[clauses]*

*for loop*

# Clauses

- DO/FOR directive can take PRIVATE and FIRSTPRIVATE clauses which refer to the scope of the loop.
- Note that the parallel loop index variable is PRIVATE by default (but other loop indices are not).
- PARALLEL DO/FOR directive can take all clauses available for PARALLEL directive.

# Parallel sections

- Allows separate blocks of code to be executed in parallel (e.g. several independent subroutines)
- Not scalable: the source code determines the amount of parallelism available.

# Parallel sections (cont)

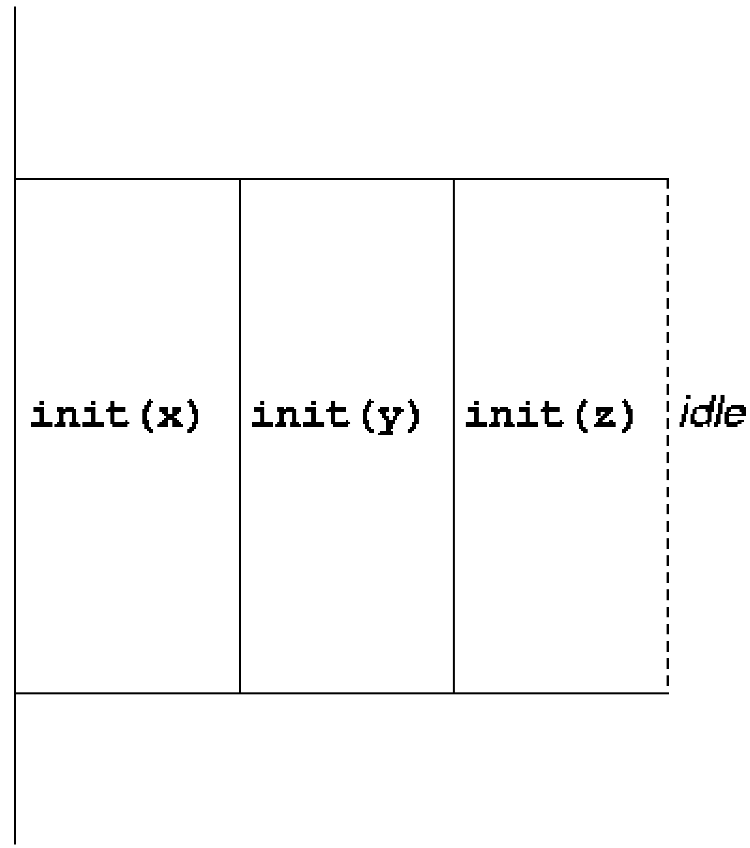
C/C++:

```
#pragma omp sections [clauses]
{
 [ #pragma omp section ]
       structured-block
 [ #pragma omp section
       structured-block
   . . . ]
}
```

# Parallel sections (cont)

Example:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
      call init(x)
!$OMP SECTION
      call init(y)
!$OMP SECTION
      call init(z)
!$OMP END SECTIONS
!$OMP END PARALLEL
```



# Parallel sections (cont)

- SECTIONS directive can take PRIVATE, FIRSTPRIVATE, LASTPRIVATE (see later) and  clauses.

- Each section must contain a structured block: cannot branch into or out of a section.

# Parallel section (cont)

Shorthand form:

Fortran:

```
!$OMP PARALLEL SECTIONS [clauses]
. . .
!$OMP END PARALLEL SECTIONS
```

C/C++:

```
#pragma omp parallel sections [clauses]
{
 . . .
 }
```

# SINGLE directive

- Indicates that a  block of code is to be executed by a single thread only.

- The first thread to reach the SINGLE directive will execute the block

- Other threads wait until block has been executed.

# SINGLE directive (cont)

Syntax:

Fortran:

```
!$OMP SINGLE [clauses]
        block
!$OMP END SINGLE
```

C/C++:

```
#pragma omp single [clauses]
        structured block
```

# SINGLE directive (cont)

Example:

```
#pragma omp parallel
{
   setup(x);
#pragma omp single
  {
     input(y);
  }
   work(x,y);
}
```

# SINGLE directive (cont)

- SINGLE directive can take PRIVATE and FIRSTPRIVATE clauses.
- Directive must contain a structured block: cannot branch into or out of it.

# MASTER directive

- Indicates that a block of code should be executed by the master thread (thread 0) only.
- Other threads skip the block and continue executing: different from SINGLE in this respect.
- Most often used for I/O.

# MASTER directive (cont)

Syntax:

Fortran:

```
!$OMP MASTER
      block
!$OMP END MASTER
```

C/C++:

```
#pragma omp master
      structured block
```

# CS 426

## More about parallel do/for loops

# LASTPRIVATE clause

- Sometimes need the value a private variable would have had on exit from loop (normally undefined).

Syntax:

Fortran: **`LASTPRIVATE(list)`**

C/C++: **`lastprivate(list)`**

- Also applies to *sections* directive (variable has value assigned to it in the last section.)

# LASTPRIVATE clause (cont)

Example:

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(i)
      do i=1,func(l,m,n)
         d(i)=d(i)+e*f(i)
      end do
      ix = i-1
      . . .
!$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.

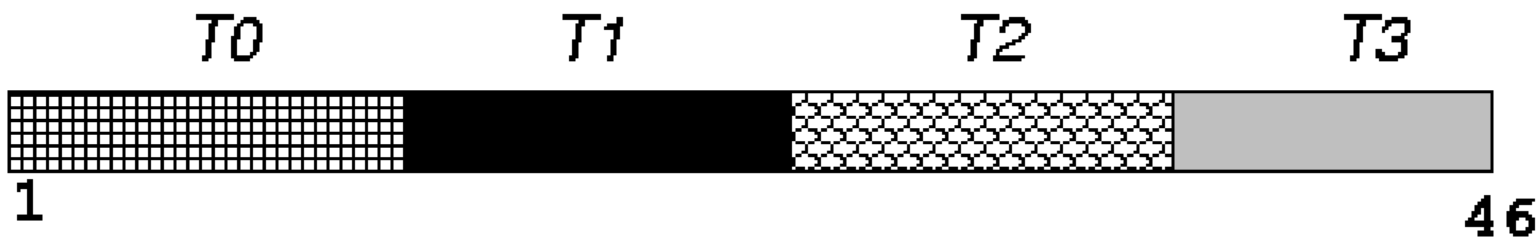
- E.g. **!$OMP DO SCHEDULE(DYNAMIC,4)**

# STATIC schedule

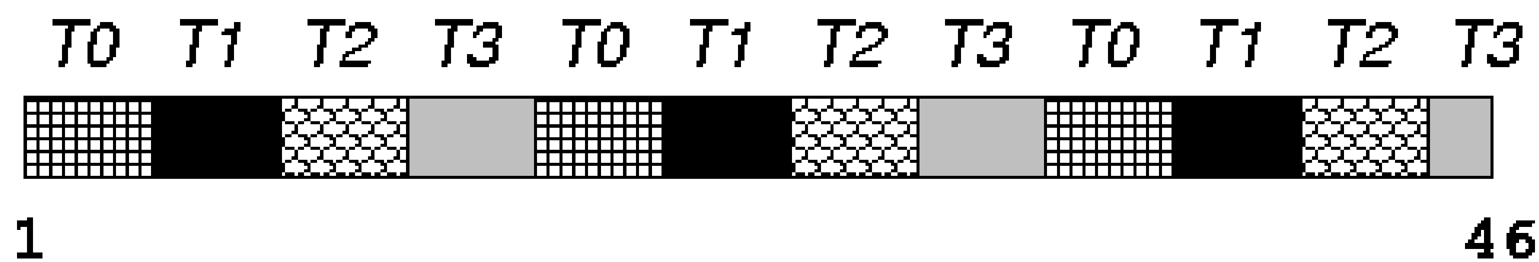
- With no *chunksize* specified, the iteration space is divided into (approximately) equal chunks, and one chunk is assigned to each thread (**block** schedule).

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

# STATIC schedule




SCHEDULE(STATIC)




SCHEDULE(STATIC,4)

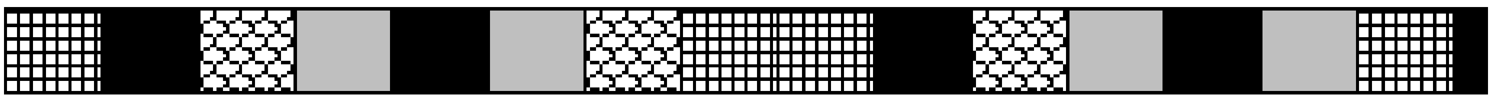
# DYNAMIC schedule

- 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

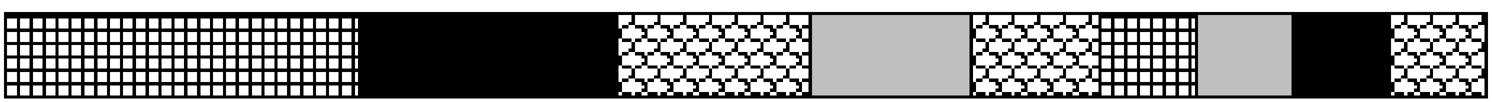
- GUIDED schedule is similar to DYNAMIC, but the chunks start off large and get smaller exponentially.
- The size of the next chunk is (roughly) the number of remaining iterations divided by the number of threads.
- The *chunksize* specifies the minimum size of the chunks.
- When no *chunksize* is specified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

# DYNAMIC and GUIDED schedules



1 46

SCHEDULE(DYNAMIC,3)



1 46

SCHEDULE(GUIDED,3)

# Choosing a schedule

When to use which schedule?

- STATIC best for load balanced loops - least overhead.
- 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!
- Use RUNTIME for convenient experimentation.

# ORDERED directive

- Can specify code within a loop which must be done in the order it would be done if executed sequentially.
- Syntax:

Fortran: **`!$OMP ORDERED`**

*block*

**`!$OMP END ORDERED`**

C/C++: **`#pragma omp ordered`**

*structured block*

- Can only appear inside a DO/FOR directive which has the ORDERED clause specified.

# ORDERED directive (cont)

Example:

```
#pragma omp for ordered [clauses...]
   (loop region)

#pragma omp ordered

   structured_block

   (endo of loop region)
```

# CS 426

## Synchronization

# Why is it required?

Recall:

- Need to synchronise actions on shared variables.
- Need to ensure correct ordering of reads and writes.
- 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 PRIVATE(I,MYID,NEIGHB)
   myid = omp_get_thread_num()
   neighb = myid - 1
   if (myid.eq.0) neighb = omp_get_num_threads()-1
   ...
   a(myid) = a(myid)*3.5

   b(myid) = a(neighb) + c
   ...
!$OMP END PARALLEL
```

# NOWAIT clause

- The NOWAIT clause can be used to suppress the implicit barriers at the end of DO/FOR, SECTIONS and SINGLE directives. (Barriers are expensive!)
- Syntax:

Fortran: **!$OMP DO**

*do loop*

**!$OMP END DO NOWAIT**

C/C++: **#pragma omp for nowait**

*for loop*

- Similarly for SECTIONS and SINGLE .

# NOWAIT clause (cont)

Example: Two loops with no dependencies

```
!$OMP PARALLEL
!$OMP DO
      do j=1,n
         a(j) = c * b(j)
      end do
!$OMP END DO NOWAIT
!$OMP DO
      do i=1,m
         x(i) = sqrt(y(i)) * 2.0
      end do
!$OMP END PARALLEL
```

# NOWAIT clause

- Use with EXTREME CAUTION!
- All too easy to remove a barrier which is necessary.
- This results in the worst sort of bug: non-deterministic behaviour (sometimes get right result, sometimes wrong, behaviour changes under debugger, etc.).
- May be good coding style to use NOWAIT everywhere and make all barriers explicit.

# NOWAIT clause (cont)

Example:

```
!$OMP DO
      do j=1,n
         a(j) = b(j) + c(j)
      end do
!$OMP DO
      do j=1,n
           d(j) = e(j) * f
      end do
!$OMP DO
      do j=1,n
          z(j) = (a(j)+a(j+1)) * 0.5
      end do
```

# 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: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
      ...
!$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
```

# ATOMIC directive

- Used to protect a single update to a shared variable.
- Applies only to a single statement.
- Syntax:

Fortran: **`!$OMP ATOMIC`**

*statement*

where *statement* must have one of these forms:

*x* **=** *x op expr*, *x* **=** *expr op x*, *x* **=** *intr* **(***x*, *expr***)** or

*x* = *intr***(***expr*, *x***)**

*op* is one of **`+`**, **`*`**, **`-`**, **`/`**, **`.and.`**, **`.or.`**, **`.eqv.`**, or **`.neqv.`**

*intr* is one of **`MAX`**, **`MIN`**, **`IAND`**, **`IOR`** or **`IEOR`**

# ATOMIC directive (cont)

C/C++: **`#pragma omp atomic`**

*statement*

where *statement* must have one of the forms:

*x binop* **=** *expr*, *x*++, ++*x*, *x*--, or --*x*

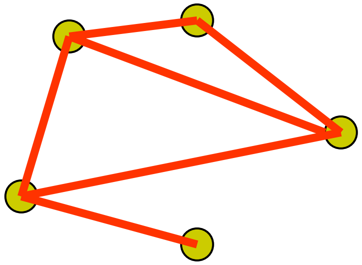
and *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **<<**, or **>>**

- Note that the evaluation of *expr* is not atomic.
- May be more efficient than using CRITICAL directives, e.g. if different array elements can be protected separately.

# ATOMIC directive (cont)

Example (compute degree of each vertex in a graph):

```
#pragma omp parallel for
      for (j=0; j<nedges; j++){
#pragma omp atomic
         degree[edge[j].vertex1]++;
#pragma omp atomic
         degree[edge[j].vertex2]++;
      }
```



# Choosing synchronisation

- As a rough guide, use ATOMIC directives if possible, as these allow most optimisation.
- If this is not possible, use CRITICAL directives. Make sure you use different *names* wherever possible.
- As a last resort you may need to use the lock routines, but this should be quite a rare occurrence.

# FLUSH directive

- The FLUSH directive ensures that a variable is written to/read from main memory.

- The variable will be *flushed* out of the register file (and out of cache on a system without sequentially consistent caches). Also sometimes called a *memory fence.*

# FLUSH directive (cont)

- Syntax:

Fortran: **!$OMP FLUSH** *[***(***list***)***]*

C/C++: **#pragma omp flush** *[***(***list***)***]*

- *list* specifies a list of variables to be flushed. If no list is specified, all shared variables are flushed.
- A FLUSH directive is implied by a BARRIER, at entry and exit to CRITICAL and ORDERED sections, and at the end of PARALLEL, DO/FOR, SECTIONS and SINGLE directives (except when a NOWAIT clause is present).

# FLUSH directive (cont)

Example (point-to-point synchronisation):

```
!$OMP PARALLEL PRIVATE(MYID,I, NEIGHB)
      . . .
      do j = 1, niters
        do i = lb(myid), ub(myid)
          a(i) = (a(i-1) + a(i))*0.5
        end do
        ndone (myid) = ndone (myid) + 1
!$OMP FLUSH (NDONE)
        do while (ndone(neighb).lt. ndone(myid))
!$OMP FLUSH (NDONE)
        end do
      end do
```

Must wait for previous iteration to finish on neighbour

Make sure write is to main memory

Make sure read is from main memory

# CS 426

## Additional Features

# Additional features

- Nested parallelism
- Orphaned directives and binding rules
- Dynamic parallelism
- Thread private global variables
- Conditional compilation
- I/O

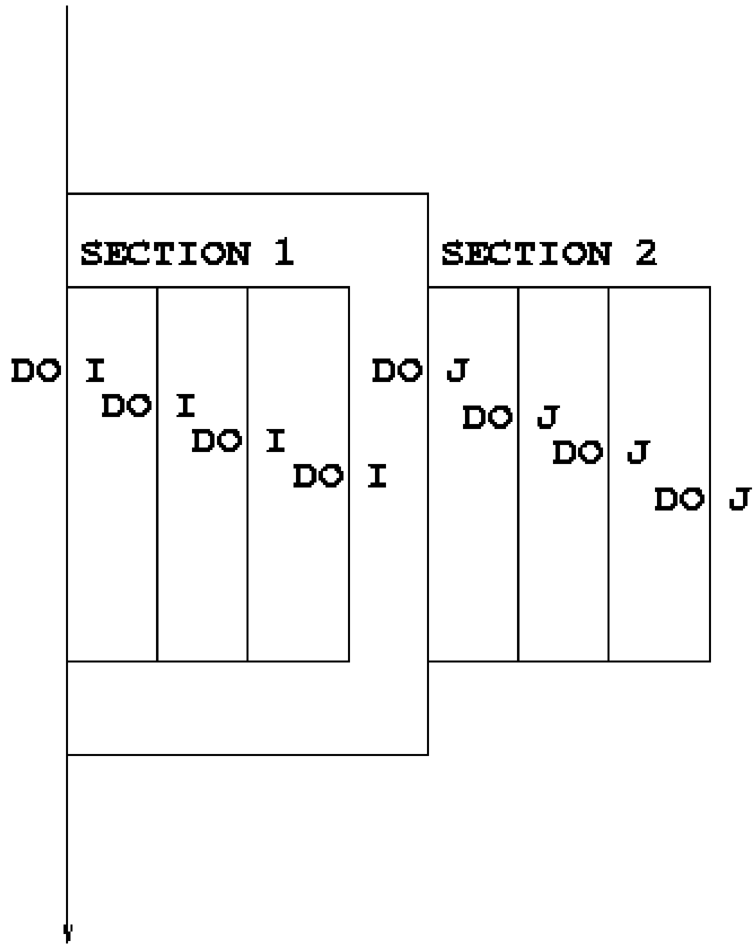
# Nested parallelism

- Unlike most previous directive systems, nested parallelism is permitted in OpenMP.
- This is enabled with the **`OMP_NESTED`** environment variable or the **`OMP_SET_NESTED`** routine.
- If a PARALLEL directive is encountered within another PARALLEL directive, a new team of threads will be created.
- The new team will contain only one thread unless nested parallelism is enabled.

# Nested parallelism (cont)

Example:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
!$OMP PARALLEL DO
      do i = 1,n
         x(i) = 1.0
      end do
!$OMP SECTION
!$OMP PARALLEL DO
      do j = 1,n
         y(j) = 2.0
      end do
!$OMP END SECTIONS
!$OMP END PARALLEL
```



# Nested parallelism (cont)

- Not often needed, but can be useful to exploit non-scalable parallelism (SECTIONS).

- Note: nested parallelism isn’t supported in many current implementations (the code will execute, but as if OMP_NESTED was not set).

- This was an unsatisfactory area in the original standard.
  - there was no way to control how many threads are used at each level of nesting
  - this was fixed in 2.0, but still not many implementations

# CS 426

## OpenMP – More Details

# Lock routines

- Occasionally we may require more flexibility than is provided by CRITICAL and ATOMIC directions.
- A lock is a special variable that may be *set* by a thread. No other thread may *set* the lock until the thread which set the lock has *unset* it.
- Setting a lock can either be blocking or non-blocking.
- A lock must be initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.

# Lock routines - syntax

Fortran:

```
SUBROUTINE OMP_INIT_LOCK(var)
SUBROUTINE OMP_SET_LOCK(var)
LOGICAL FUNCTION OMP_TEST_LOCK(var)
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_DESTROY_LOCK(var)
```

*var* should be an INTEGER of the same size as addresses (e.g. INTEGER*8 on a 64-bit machine)

# Lock routines - syntax

C/C++:

```
#include <omp.h>
  void omp_init_lock(omp_lock_t *lock);
  void omp_set_lock(omp_lock_t *lock);
  int omp_test_lock(omp_lock_t *lock);
  void omp_unset_lock(omp_lock_t *lock);
  void omp_destroy_lock(omp_lock_t *lock);
```

There are also nestable lock routines which allow the same thread to set a lock multiple times before unsetting it the same number of times.

# Lock routines (cont)

Example:

```
// omp_test_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>
omp_lock_t simple_lock;
int main() {
    omp_init_lock(&simple_lock);
    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();

        while (!omp_test_lock(&simple_lock))
            printf_s("Thread %d - failed to acquire simple_lock\n",tid);

        printf_s("Thread %d - acquired simple_lock\n", tid);

        printf_s("Thread %d - released simple_lock\n", tid);
        omp_unset_lock(&simple_lock);
    }
    omp_destroy_lock(&simple_lock);
}
```

# Orphaned directives

- Directives are active in the *dynamic* scope of a parallel region, not just its *lexical* scope.
- Example:

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

      subroutine fred()
!$OMP DO
      do i = 1,n
         a(i) = a(i) + 23.5
      end do
      return
      end
```

# Orphaned directives (cont)

- This is very useful, as it allows a modular programming style….

- But it can also be rather confusing if the call tree is complicated (what happens if **`fred`** is also called from outside a parallel region?)

- There are some extra rules about data scope attributes….

# Data scoping rules

When we call a subroutine from inside a parallel region:

- Variables in the argument list inherit their data scope attribute from the calling subroutine.
- Global variables and COMMON blocks are shared, unless declared THREADPRIVATE (see later).
- **`static`** local variables in C/C++ and **SAVE** variables in Fortran are shared.
- All other local variables are private.

# Orphaned directives (cont)

- We can find out if we are in a parallel region or not with the OMP_IN_PARALLEL function:

Fortran: `LOGICAL FUNCTION OMP_IN_PARALLEL()`

C/C++:
```
#include <omp.h>
     int omp_in_parallel(void);
```

# Binding rules

- There could be ambiguity about which parallel region directives refer to, so we need some rules….
- DO/FOR, SECTIONS, SINGLE, MASTER and BARRIER directives always bind to the nearest enclosing PARALLEL directive.
- ORDERED directive binds to nearest enclosing DO directive.

# Dynamic parallelism

- It is possible to let the system choose how many threads execute each parallel region, to let it optimise resource allocation.
- The number of threads will be equal to or less than that set by the user, and remains fixed for the duration of each parallel region.
- Can be set by OMP_SET_DYNAMIC routine or by the OMP_DYNAMIC environment variable.
- Its default value is implementation dependent: if your code relies on using a certain number of threads (not recommended) you should disable dynamic parallelism.

# Thread private global variables

- It can be convenient for each thread to have its own copy of variables with global scope (COMMON blocks in Fortran, or file-scope and namespace-scope variables in C/C++).

- Outside parallel regions and in MASTER directives, accesses to these variables refer to the master thread's copy.

# Thread private globals (cont)

Syntax:

Fortran: **`!$OMP THREADPRIVATE (`***`/cb/[,/cb/]`***`)`**

where *cb* is a named common block.

This directive must come after all the declarations for the common blocks.

C/C++: **`#pragma omp threadprivate (`***`list`***`)`**

This directive must be a file or namespace scope, after all declarations of variables in *list* and before any references to variables in *list*. See standard document for other restrictions.

# COPYIN clause

- Allows the values of the master thread's THREADPRIVATE data to be copied to all other threads at the start of a parallel region.

Syntax:

Fortran: **`COPYIN(`***list***`)`**

C/C++: **`copyin(`***list***`)`**

In Fortran the list can contain both COMMON blocks and variables in COMMON blocks.

# COPYIN clause

Example:

```
      common /junk/ nx
      common /stuff/ a,b,c
!$OMP THREADPRIVATE (/JUNK/,/STUFF/)
      nx = 32
      c = 17.9
      . . .
!$OMP PARALLEL PRIVATE(NX2,CSQ) COPYIN(/JUNK/,C)
      nx2 = nx * 2
      csq = c*c
      . . .
```

# Conditional compilation

- Allows source lines to be recognised by an OpenMP compiler and ignored (treated as comments) by other compilers.
- In C/C++ this is done in the traditional way with the preprocessor macro **`_OPENMP`**
- In Fortran, in addition to this macro, any line beginning with the sentinels !$, C$ or *$ (latter two only in fixed source form), is conditionally compiled.
- The sentinel is replaced with two spaces.

# Conditional compilation (cont)

Example (read value of OMP_NUM_THREADS):

```
     nthreads = 1
!$OMP PARALLEL
!$OMP MASTER
!$    nthreads = omp_get_num_threads()
!$OMP END MASTER
!$OMP END PARALLEL
      print *, “No. of threads = “, nthreads
```

# I/O

- Should assume that I/O is not thread-safe.
- Need to synchronise multiple threads writing to *or reading from* the same file.
  - Note that there is no way for multiple threads to have private file positions.
- OK to have multiple threads reading/writing to different files.

# CS 426

## OpenMP functions listed

# Directives

- General format:
  **`#pragma omp`** *directive-name* [*clause, …*] *newline*
- **`#pragma omp`**
  - Required for all OpenMP C/C++ directives
- *directive-name*
  - A valid OpenMP directive. Must appear after pragma and before clauses
- [*clause, …*]
  - Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- *newline*
  - Required. Followed by structured block

# Directives

- General Rules:
  - Case sensitive
  - Follow conventions of C/C++ standards for compile direvtives
  - Only one directive name may be specified per directive
  - Each directive applies to at most one succeeding statement, which must be a structured block
  - Long directive lines can be continued by succeeding lines by excaping the newline character with a backslash ('\') at the end of a directive line

# Syntax – `atomic`

- `#pragma omp atomic 'statement'`
- `'statement'` can be:
  - `x bin_op= expr`
    - **`bin_op: {+ * - / & ^ | << >>}`**
    - ***`expr`*****`:`** an expression of scalar type that does not reference **`x`**
  - `x++`
  - `++x`
  - `x--`
  - `--x`
- Indicates that the specified memory location must be updated atomically and not be exposed to multiple, simultaneous writing threads.

# Syntax – `parallel`

- **`#pragma omp parallel 'clause'`**
- **`'clause'`** can be:
  - **`if(`*`exp`*`)`**
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`num_threads(`*`int_exp`*`)`**
  - **`shared(`*`list`*`)`**
  - **`default(shared|none)`**
  - **`copyin(`*`list`*`)`**
  - **`reduction(`*`operator: list`*`)`**
- Indicates that the code section is to be parallelized

# Syntax – `for`

- **`#pragma omp for 'clause'`**
- **`'clause'`** can be:
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`lastprivate(`*`list`*`)`**
  - **`reduction(`*`operator: list`*`)`**
  - **`ordered`**
  - **`schedule(`*`type`*`)`**
  - **`Nowait`**
- Compiler distributes loop iterations within team of threads

# Syntax – `ordered`

- **`#pragma omp ordered`**
  - Indicates that the code section must be executed in sequential order

# **Syntax** – `parallel for`

- **`#pragma omp parallel for ‘clause’`**
- **`‘clause’`** can be:
  - **`if(`*`exp`*`)`**
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`lastprivate(`*`list`*`)`**
  - **`num_threads(`*`int_exp`*`)`**
  - **`shared(`*`list`*`)`**
  - **`default(shared|none)`**
  - **`copyin(`*`list`*`)`**
  - **`reduction(`*`operator: list`*`)`**
  - **`ordered`**
  - **`schedule(`*`type`*`)`**
- Combines the **`omp parallel`** and **`omp for`** directives

# **Syntax** – `sections`

- **`#pragma omp sections 'clause'`**
- **`'clause'`** can be:
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`lastprivate(`*`list`*`)`**
  - **`reduction(`*`operator: list`*`)`**
  - **`nowait`**
- In structured block following the directive, an **`opm section`** directive will indicate that the following sub-block can be distributed for parallel execution.

# **Syntax** – `parallel sections`

- **`#pragma omp parallel sections 'clause'`**
- **`'clause'`** can be:
  - **`if(`*`exp`*`)`**
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`lastprivate(`*`list`*`)`**
  - **`shared(`*`list`*`)`**
  - **`default(shared|none)`**
  - **`copyin(`*`list`*`)`**
  - **`reduction(`*`operator: list`*`)`**
  - **`Nowait`**
- Combines the **`omp parallel`** and **`omp sections`** directives

# Syntax – `single`

- **`#pragma omp single 'clause'`**
- **`'clause'`** can be:
  - **`private(`*`list`*`)`**
  - **`copyprivate(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`Nowait`**
- Indicates that the code section must only be run by a single available thread.

# Syntax – `master`

- **`#pragma omp master`**
  - Indicates that the code section must only be run by master thread

# Syntax – `critical`

- **`#pragma omp critical`**
  - Indicates that the code section can only be executed by a single thread at any given time

# Syntax – `barrier`

- **`#pragma omp barrier`**
  - Identifies a synchronization point at which threads in a parallel region will not continue until all other threads in that section reach the same spot
  - Explicit for a few directives
    - **`omp parallel`**
    - **`omp for`**

# Syntax – `flush`

- **`#pragma omp flush (`*`list`*`)`**
  - Identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory. If no list is given, then all shared objects are synchronized.
  - **`flush`** is implicit for the following directives:
    - **`omp barrier`**
    - Entrance and exit of **`omp critical`**
    - Exit of **`omp parallel`**
    - Exit of **`omp for`**
    - Exit of **`omp sections`**
    - Exit of **`omp single`**

# Syntax – `threadprivate`

- **`#pragma omp threadprivate (`*`var`*`)`**
  - **`omp threadprivate`** makes the variable private to a thread

# OpenMP Functions

- `void omp_set_num_threads (int)`
  - Called inside serial section. Can exceed available processors
- `int omp_get_num_threads (void)`
  - Returns number of active threads
- `int omp_get_max_threads (void)`
  - Returns max system allowed threads
- `int omp_get_thread_num (void)`
  - Returns thread’s ID number (ranges from `0` to `t-1`)
- `int omp_get_num_procs(void)`
  - Returns number of processors available to the program

# OpenMP Functions

- `int omp_in_parallel (void)`
  - Returns **1** if called inside a parallel block
- `void omp_set_dynamic (int)`
  - Enable (**1**) or disable (**0**) dynamic threads
- `int omp_get_dynamic (void)`
  - Returns **1** if dynamic threads enabled
- `void omp_set_nested (int)`
  - Enable (**1**) or disable (**0**) nested parallelism
- `int omp_get_nested (void)`
  - Returns **1** if nested parallelism enabled (default **0**)

# OpenMP Functions

- `void omp_init_lock(omp_lock_t*)`
  - Initializes a lock associated with the lock variable
- `void omp_destroy_lock(omp_lock_t*)`
  - Disassociates the given lock variable from any locks
- `void omp_set_lock(omp_lock_t*)`
  - Wait until specified lock is available
- `void omp_unset_lock(omp_lock_t*)`
  - Releases the lock from executing routine
- `int omp_test_lock(omp_lock_t*)`
  - Attempts to set a lock, but does not wait if the lock is unavailable
  - Returns non-zero value on success

# OpenMP Functions

- `double omp_get_wtime(void)`
  - Returns the number of elapsed seconds since some point in the past
- `double omp_get_wtick(void)`
  - Returns the number of elapsed seconds between successive clock ticks

# Environment Variables

- OMP_SCHEDULE
  - Applies only to parallel for directives with their schedule clause set to **`RUNTIME`**
  - Determines how iterations of the loop are scheduled
- OMP_NUM_THREADS
  - Maximum number of threads to use for execution
- OMP_DYNAMIC
  - Enable (**`1`**) or disable (**`0`**) dynamic adjustment of threads available for execution
- OMP_NESTED
  - Enable (**`1`**) or disable (**`0`**) nested parallelism

# Clause - `list`

- **`list`**
  - **`private(`*`list`*`)`**
  - **`firstprivate(`*`list`*`)`**
  - **`lastprivate(`*`list`*`)`**
  - **`shared(`*`list`*`)`**
  - **`copyin(`*`list`*`)`**
- List of variables

# **Clause** – `operator: list`

- ***`operator: list`***
  - **`reduction(`*`operator: list`*`)`**
- **Operators includes:**
  - **`+`**
  - **`*`**
  - **`&`**
  - **`|`**
  - **`^`**
  - **`&&`**
  - **`||`**

# **Clause –** `schedule(type, size)`

- `schedule(type, size)`
  - `schedule(static)`
    - Allocates `n / t` contiguous iterations to each thread
  - `schedule(static, C)`
    - Allocates *C* contiguous iterations to each thread
  - `schedule(dynamic)`
    - Allocates `1` iteration at a time, dynamically
  - `schedule(dynamic, C)`
    - Allocates *C* iterations at a time, dynamically
  - `schedule(guided, C)`
    - Allocates decreasingly large iterations to each thread until size reaches *C*
  - `schedule(guided)`
    - Same as `(guided, C)`, with `C = 1`
  - `schedule(runtime)`
    - Based on environment variable `OMP_SCHEDULE`

# Examples – Reduction

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
  int   i, n;
  float a[100], b[100], sum;

  n = 100; /* Some initializations */
  for (i=0; i < n; i++)
    a[i] = b[i] = i * 1.0;
  sum = 0.0;
  #pragma omp parallel for reduction(+:sum)
    for (i=0; i < n; i++)
      sum = sum + (a[i] * b[i]);
  printf("   Sum = %f\n",sum);
}
```

# Examples – OpenMP Functions

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]){
int nthreads, tid, procs, maxt, inpar,
    dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid) {
  /* Obtain thread number */
  tid = omp_get_thread_num();

  /* Only master thread does this */
  if (tid == 0) {
    printf("Thread %d getting info...\n", tid);

    /* Get environment information */
    procs = omp_get_num_procs();
    nthreads = omp_get_num_threads();
    maxt = omp_get_max_threads();
    inpar = omp_in_parallel();
    dynamic = omp_get_dynamic();
    nested = omp_get_nested();

    /* Print environment information */
    printf("Number of processors = %d\n", procs);
    printf("Number of threads = %d\n", nthreads);
    printf("Max threads = %d\n", maxt);
    printf("In parallel? = %d\n", inpar);
    printf("Dynamic threads? = %d\n", dynamic);
    printf("Nested parallelism? = %d\n", nested);
    }
  }  /* Done */
}
```

# CS 426

## OpenMP 2.0

# New features in Fortran 2.0

- Fuller support for Fortran 90/95:
  - WORKSHARE directive for array syntax.
  - THREADPRIVATE/COPYIN on variables (e.g. for module data).
  - In-line comment in directives.
- Reductions on arrays.
- COPYPRIVATE on END SINGLE (propagates value to all threads).
- NUM_THREADS clause on parallel regions.
- Timing routines.
- …. plus some clarifications (e.g. reprivatisation of variables *is* allowed.)

# New features in C/C++ 2.0

- COPYPRIVATE on END SINGLE (propagates value to all threads).
- NUM_THREADS clause on parallel regions.
- Timing routines.
- ...plus a lot of correction/clarifications.

# Workshare directive

- A worksharing directive (!) which allows parallelisation of Fortran 90 array operations, WHERE and FORALL constructs.

- Syntax:

```
!$OMP WORKSHARE
        block
!$OMP END WORKSHARE [NOWAIT]
```

# Workshare directive (cont.)

- Simple example

```
REAL A(100,200), B(100,200), C(100,200)
...
!$OMP PARALLEL
!$OMP WORKSHARE
      A=B+C
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

- N.B. No schedule clause: distribution of work units to threads is entirely up to the compiler!

# Workshare directive (cont.)

- Can also contain array intrinsic functions, WHERE and FORALL constructs, scalar assignment to shared variables, ATOMIC and CRITICAL directives.
- No branches in or out of block.
- No function calls except array intrinsics and those declared ELEMENTAL.
- Combined directive:

```
!$OMP PARALLEL WORKSHARE
          block
!$OMP END PARALLEL WORKSHARE
```

# Workshare directive (cont.)

- Example:

```
!$OMP PARALLEL WORKSHARE
      A = B + C
      WHERE (D .ne. 0) E = 1/D
!$OMP ATOMIC
      t = t + SUM(F)
      FORALL (i=1:n, X(i)=0) X(i)= 1
!$OMP END PARALLEL WORKSHARE
```

# THREADPRIVATE variables

- THREADPRIVATE directive (and COPYIN) clause can be applied to variables not in COMMON.
- Useful for module data and SAVEd variables.

Example:

```
MODULE FRED
REAL XX(100)
!$OMP THREADPRIVATE (XX)
END MODULE FRED
```

```
SUBROUTINE DAISY
USE FRED
!$OMP PARALLEL
....
XX = YY
...
!$OMP END PARALLEL
```

# Array reductions

- Arrays may be used as reduction variables (previously only scalars and array elements).

Example:

```
!$OMP PARALLEL DO PRIVATE(I) REDUCTION(+:B)
      DO J = 1,N
         DO I = 1,M
            B(I) = B(I) + A(I,J)
         END DO
      END DO
```

# COPYPRIVATE clause

- Broadcasts the value of a private variable to all threads at the end of a SINGLE directive.
- Perhaps most useful for reading in the value of private variables.
- Syntax:

Fortran:

```
!$OMP END SINGLE COPYPRIVATE(list)
```

C/C++:

```
#pragma omp single copyprivate(list)
```

# COPYPRIVATE clause

Example:

```
!$OMP PARALLEL PRIVATE(A,B)
   ...
!$OMP SINGLE
      READ(24) A
!$OMP END SINGLE COPYPRIVATE(A)
      B = A*A
...
!$OMP END PARALLEL
```

# Nested parallelism again

- OpenMP 1.0/1.1 specification of nested parallelism has a serious omission: there is no way to specify how many threads should execute each level.

e.g. 2-d decomposition of 2-d loop nest:

```
!$OMP PARALLEL DO
      DO I = 1,4
!$OMP PARALLEL DO
         DO J = 1,N
            A(I,J) = B(I,J)
         END DO
      END DO
```

# NUMTHREADS clause

- This is addressed in OpenMP 2.0 (Fortran and C/C++) with the NUM_THREADS clause.

e.g.:

```
!$OMP PARALLEL DO NUM_THREADS(4)
      DO I = 1,4
!$OMP PARALLEL DO NUM_THREADS(TOTALTHREADS/4)
         DO J = 1,N
            A(I,J) = B(I,J)
         END DO
      END DO
```

Note: The value set in the clause supersedes the value in the environment variable OMP_NUM_THREADS (or that set by **`omp_set_num_threads()`** )

# Nested parallelism

- However, even 2.0 compliant compilers still may not implement nested parallelism......
- Turns out to be very hard to do correctly without impacting performance significantly.

# Other things

- Inline comments in directives

```
!$OMP PARALLEL DO      !Directive added by JMB 1/8/01
```

- Timing routines:
  - return current wall clock time (relative to arbitrary origin) with:

```
 DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
double omp_get_wtime(void);
```

  - return clock precision with

```
 DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
double omp_get_wtick(void);
```

# Using timers

```
DOUBLE PRECISION STARTTIME, TIME

STARTTIME = OMP_GET_WTIME()
......(work to be timed)
TIME = OMP_GET_WTIME()- STARTTIME
```

Note: timers are local to a thread: must make both calls on the same thread.

Also note: no guarantees about resolution!

# Clarifications

- Both Fortran and  C/C++ 2.0 standards contain quite a number of corrections and clarifications.
- If something is not clear in the 1.0/1.1 standard, it is worth reading the relevant section of 2.0, even if you are not using a 2.0 compliant compiler....