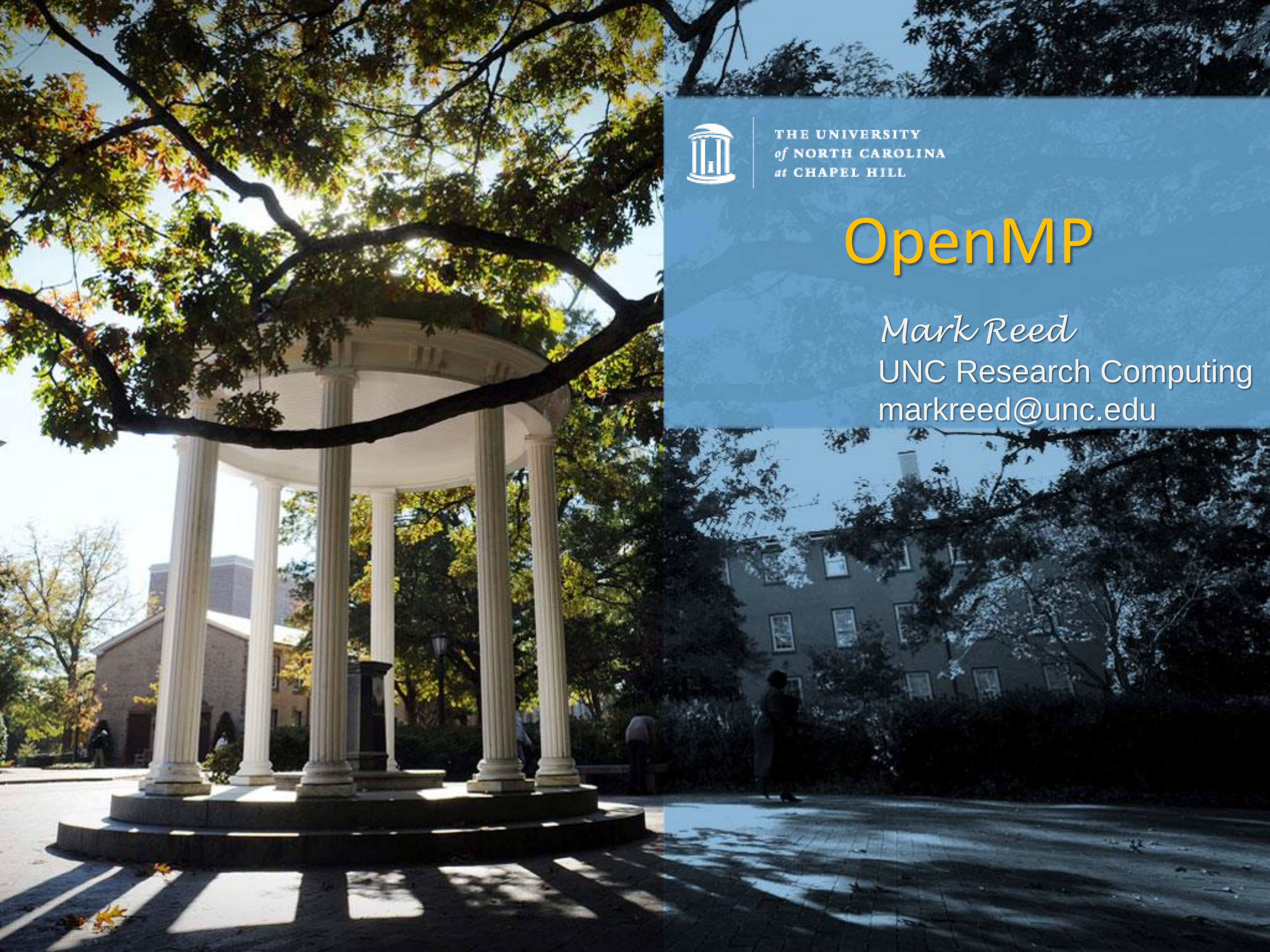


THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

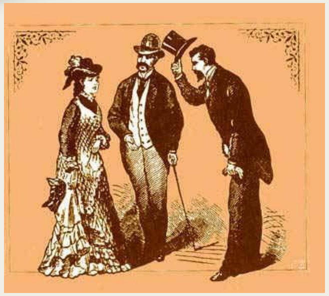
OpenMP

Mark Reed
UNC Research Computing
markreed@unc.edu

Logistics

- Course Format
- Lab Exercises
- Breaks
- Getting started:
  - Kure: http://help.unc.edu/help/getting-started-on-kure/
  - Killdevil: http://help.unc.edu/help/getting-started-on-killdevil/
- UNC Research Computing
  - *http://its.unc.edu/research*

Course Overview



Introduction

- Objectives, History, Overview, Motivation

Getting Our Feet Wet

- Memory Architectures, Models (programming, execution, memory, …), compiling and running

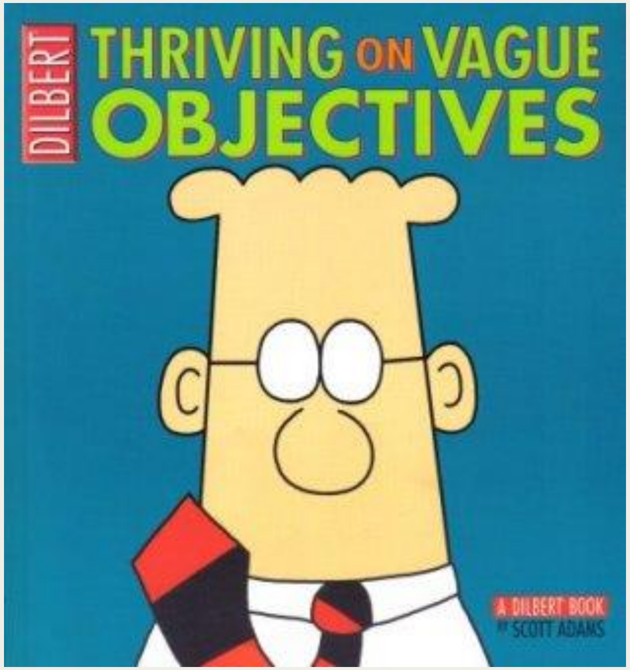
Diving In

- Control constructs, worksharing, data scoping, synchronization, runtime control

OpenMP Introduction

Course Objectives

- Introduction to the OpenMP standard
- Cover all the basic constructs
- After completion of this course users should be ready to begin parallelizing their application using OpenMP



Why choose OpenMP ?

- Portable
  - standardized for shared memory architectures
- Simple and Quick
  - incremental parallelization
  - supports both fine grained and coarse grained parallelism
  - scalable algorithms without message passing
- Compact API
  - simple and limited set of directives

In a Nutshell

- Portable, Shared Memory Multiprocessing API
  - Multi-vendor support
  - Multi-OS support (Unixes, Windows, Mac)
- Standardizes fine grained (loop) parallelism
- Also supports coarse grained algorithms
- The MP in OpenMP is for multi-processing
- *Don't confuse OpenMP with Open MPI! :)*

Version History

- First
  - Fortran 1.0 was released in October 1997
  - C/C++ 1.0 was approved in November 1998
- Recent
  - OpenMP 3.0 API released May 2008
- Current – Still Active
  - OpenMP 4.5 released November 2015
    - Major new release
    - significantly improved support for devices

A First Peek: Simple OpenMP Example

Consider arrays a, b, c and this simple loop:

Fortran

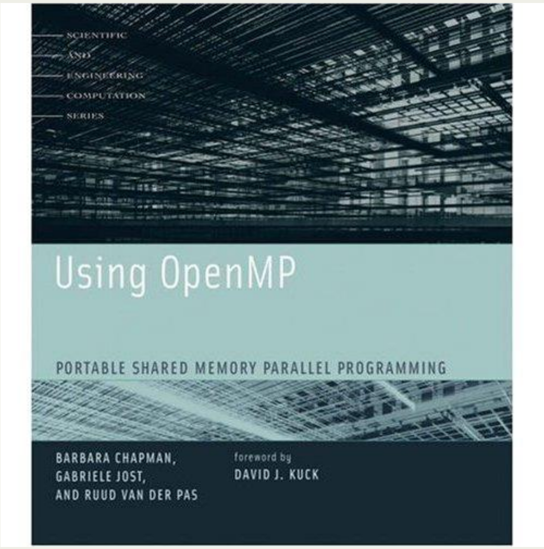
```fortran
!$OMP parallel do
!$OMP& shared (a, b, c)
do i=1,n
   a(i) = b(i) + c(i)
enddo
!$OMP end parallel do
```

C/C++

```c
#pragma omp parallel for\
        shared (a,b,c)
for (i=0; i<n; i++) {
    a(i) = b(i) + c(i)
}
```

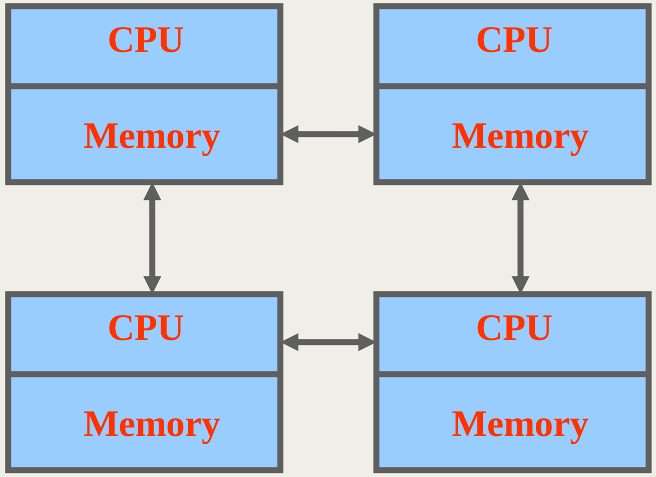
References

- See online tutorial at www.openmp.org
- OpenMP Tutorial from SC98
  - Bob Kuhn, Kuck & Associates, Inc.
  - Tim Mattson, Intel Corp.
  - Ramesh Menon, SGI
- SGI course materials
- *"Using OpenMP" book*
  - *Chapman, Jost, and Van Der Past*
- Blaise Barney LLNL tutorial
  - https://computing.llnl.gov/tutorials/openMP/



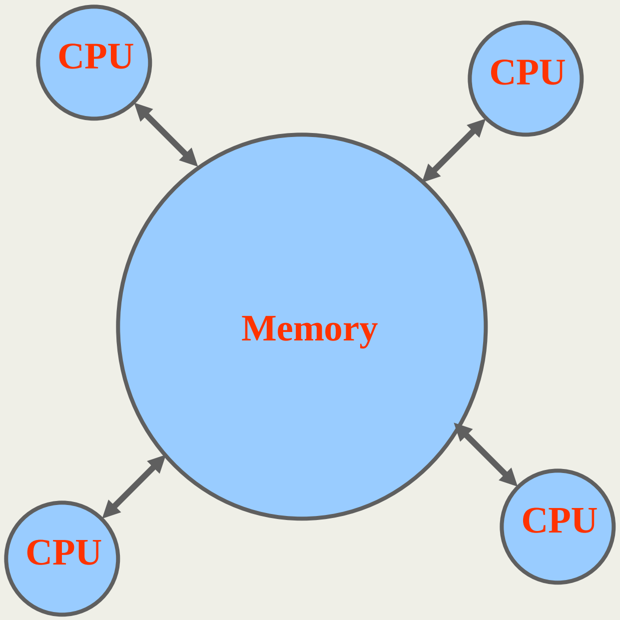
Getting our feet wet

Memory Types

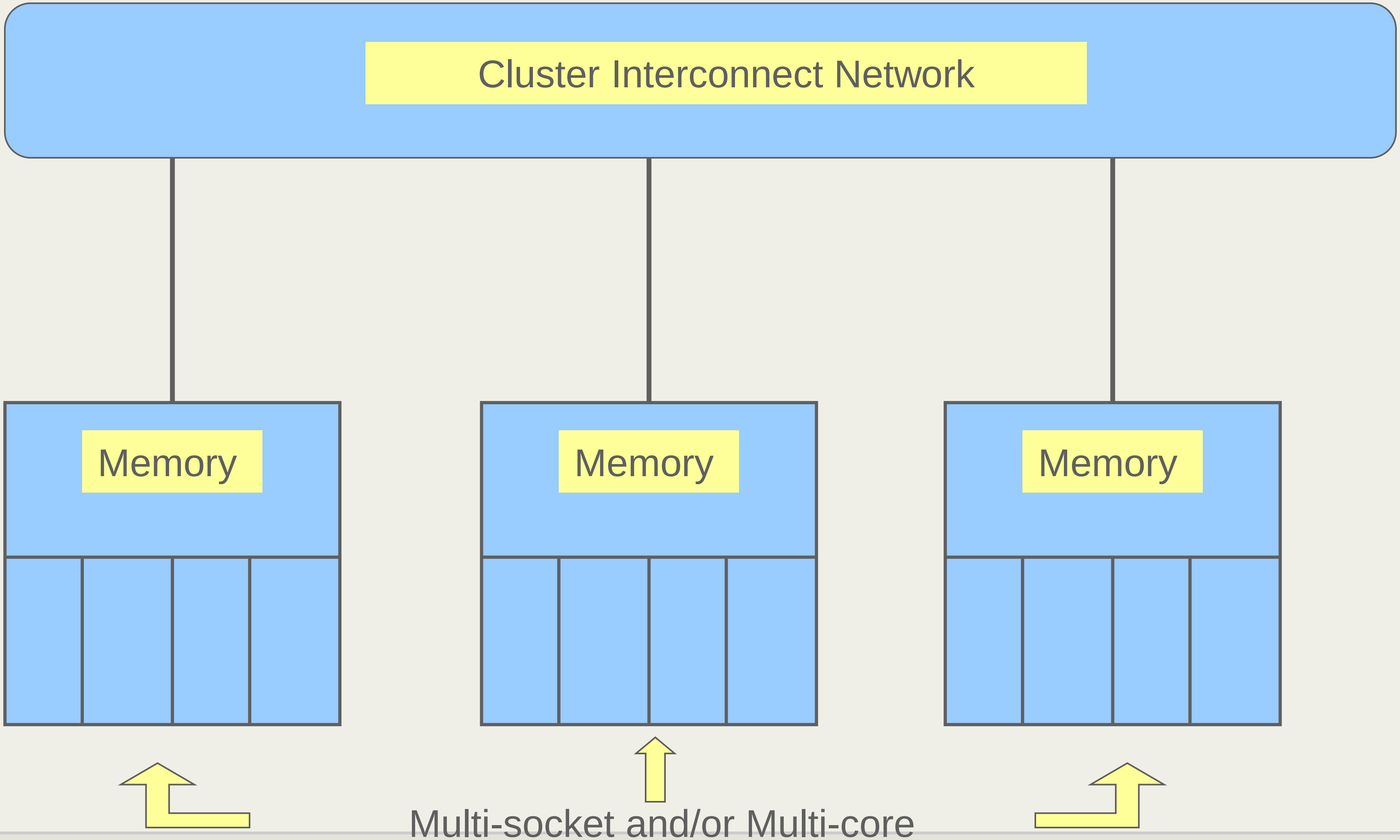


Distributed

Shared



Clustered SMPs



Distributed vs. Shared Memory

- Shared - all processors share a global pool of memory
  - simpler to program
  - bus contention leads to poor scalability
- Distributed - each processor physically has it's own (private) memory associated with it
  - scales well
  - memory management is more difficult

Models, models, models …

No Not These!

Nor These Either!

We want programming models, execution models, communication models and memory models!

OpenMP - User Interface Model

- Shared Memory with thread based parallelism
- Not a new language
- Compiler directives, library calls and environment variables extend the base language
  - f77, f90, f95, C, C++
- Not automatic parallelization
  - user explicitly specifies parallel execution
  - compiler does not ignore user directives even if wrong

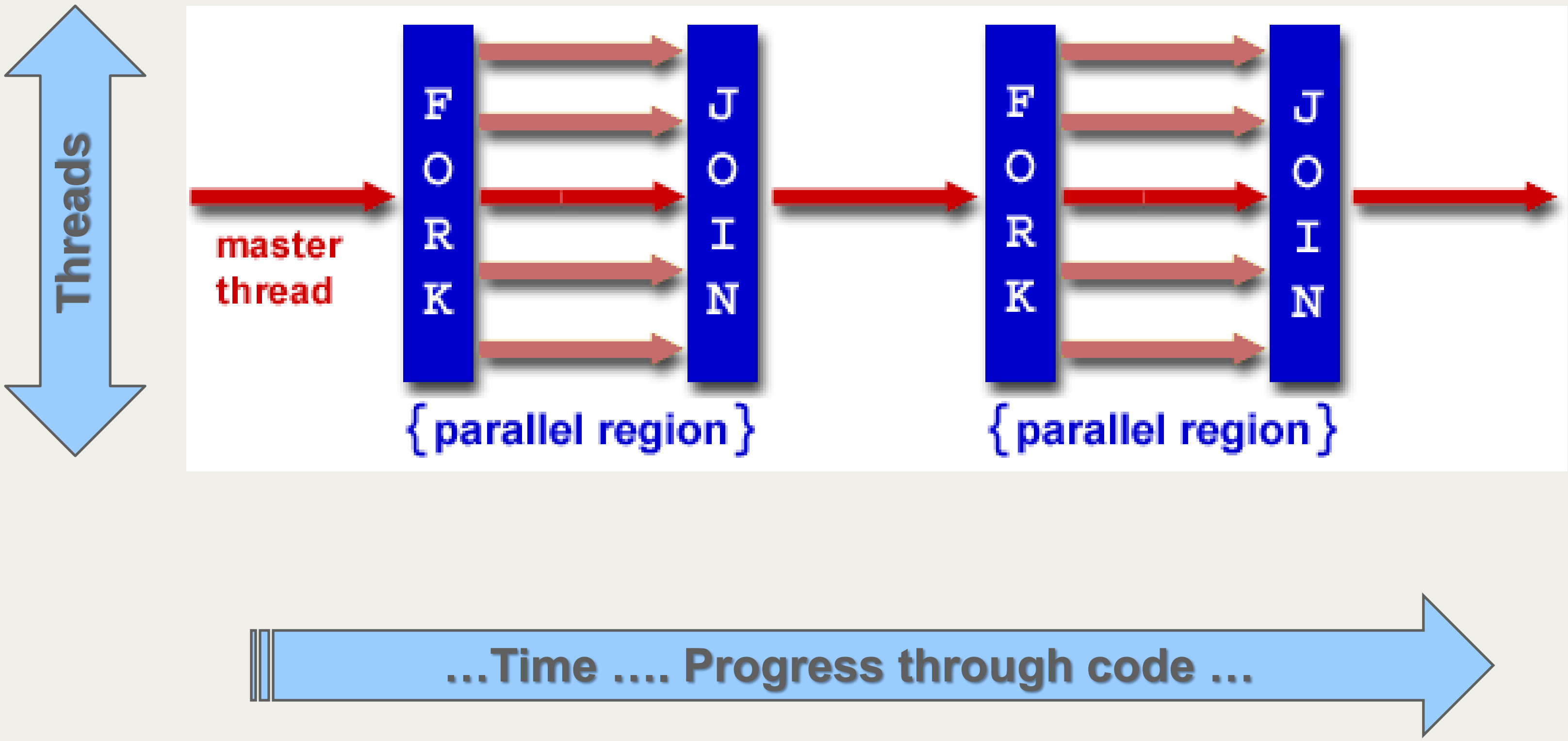
What is a thread?

- A thread is an independent instruction stream, thus allowing concurrent operation
- threads tend to share state and memory information and may have some (usually small) private data
- Similar (but distinct) from processes. Threads are usually lighter weight allowing faster context switching
- in OpenMP one usually wants no more than one thread per core

Execution Model

- OpenMP program starts single threaded
- To create additional threads, user starts a parallel region
  - additional threads are launched to create a team
  - original (master) thread is part of the team
  - threads "go away" at the end of the parallel region: usually sleep or spin
- Repeat parallel regions as necessary
  - **Fork-join model**

Fork - Join Model



Communicating Between Threads

- Shared Memory Model
  - threads read and write shared variables
    - no need for explicit message passing
  - use synchronization to protect against race conditions
  - change storage attributes for minimizing synchronization and improving cache reuse

Storage Model – Data Scoping

- Shared memory programming model: variables are shared by default
- Global variables are **`SHARED`** among threads
  - Fortran: **`COMMON`** blocks, **`SAVE`** variables, **`MODULE`** variables
  - C: file scope variables, **`static`**
- Private Variables:
  - exist only within the new scope, i.e. they are uninitialized and undefined outside the data scope
  - loop index variables
  - Stack variables in sub-programs called from parallel regions

Putting the models together - Summary

Model	Implementation
Programming	Put directives in code
Execution	create parallel regions, Fork-Join
Memory	Data scope is private or shared
Communication	Only shared variables carry information between threads

Creating Parallel Regions

- Only one way to create threads in OpenMP API:
- Fortran:

```
!$OMP parallel
   < code to be executed in parallel >
!$OMP end parallel
```

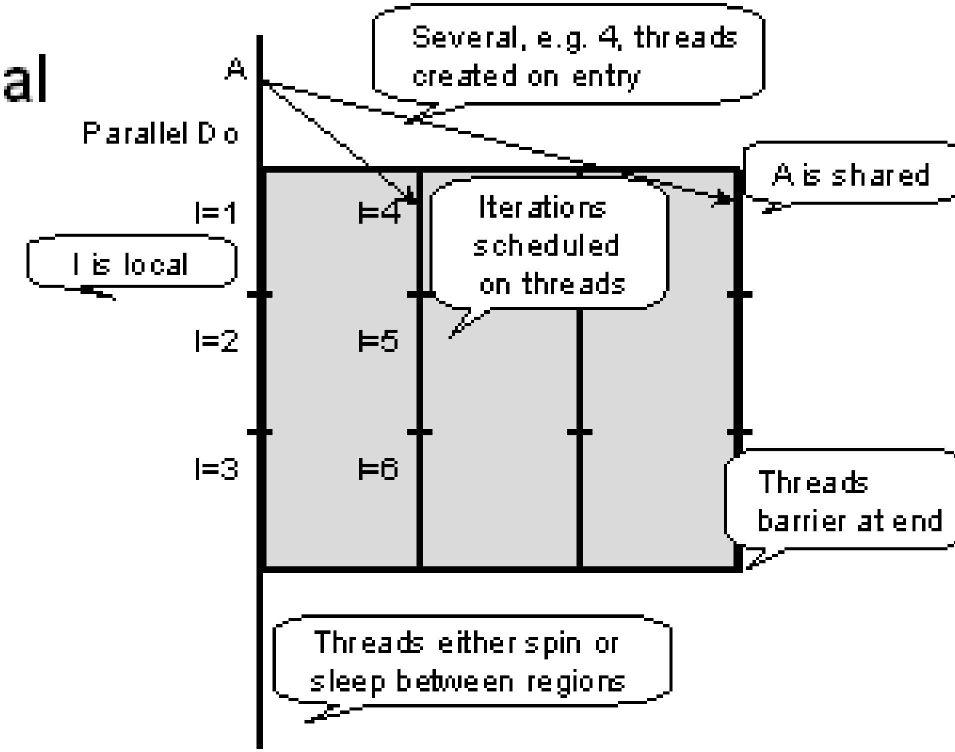
- C

```
#pragma omp parallel
{
   code to be executed by each thread
}
```

1.A.2 Parallel Loop Model

- Note threads, shared, local

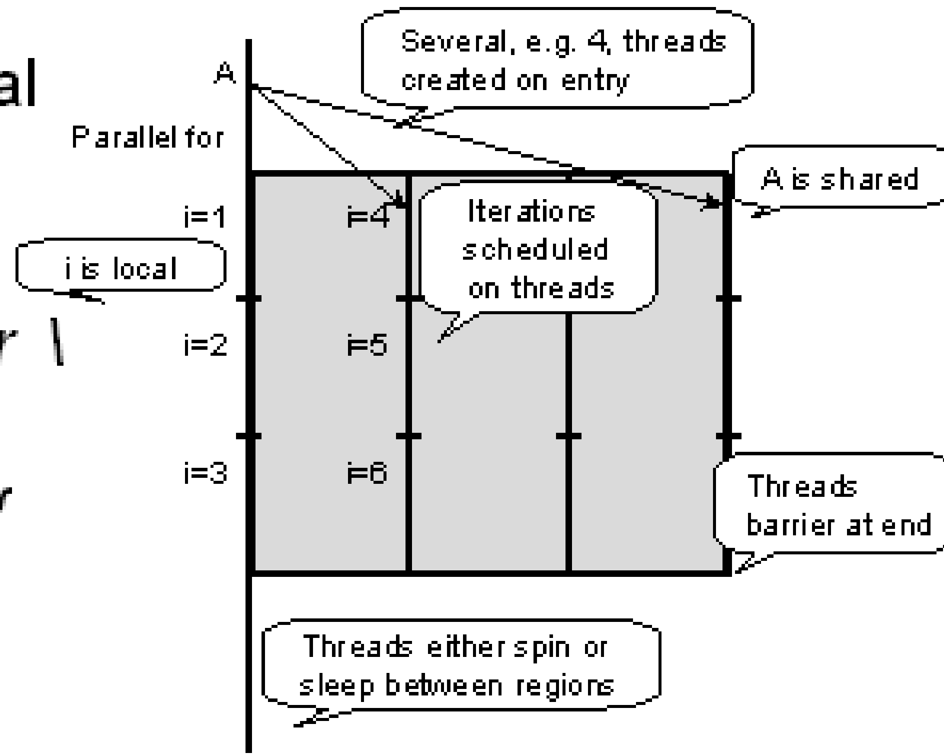
```
    program
c$omp   parallel do
c$omp& shared(A) private(I)
    do I=1,100
       . . .
    enddo
c$omp end parallel
    end
```



Parallel Loop Model in C

- Note threads, shared, local

```
main() {
#pragma omp parallel for \
    shared(A)private(i)
    for( i=1; i<=100; i++ ) {
        . . .
    }
}
```



Comparison of Programming Models

Feature	Open MP	MPI
Portable	yes	yes
Scalable	less so	yes
Incremental Parallelization	yes	no
Fortran/C/C++ Bindings	yes	yes
High Level	yes	mid level

Compiling

Intel (icc, ifort, icpc)

- -qopenmp *(-openmp is now deprecated)*

PGI (pgcc, pgf90, pgCC, …)

- -mp

GNU (gcc, gfortran, g++)

- -fopenmp
- need version 4.2 or later
- g95 was based on GCC but branched off
  - I don't think it has Open MP support

Compilers

- No specific Fortran 90 or C++ features are required by the OpenMP specification
- Most compilers support OpenMP, see compiler documentation for the appropriate compiler flag to set for other compilers, e.g. IBM, Cray, …

Compiler Directives

C Pragmas

- C pragmas are case sensitive
- Use curly braces, {}, to enclose parallel regions
- Long directive lines can be "continued" by escaping the newline character with a backslash ("\") at the end of a directive line.

Fortran

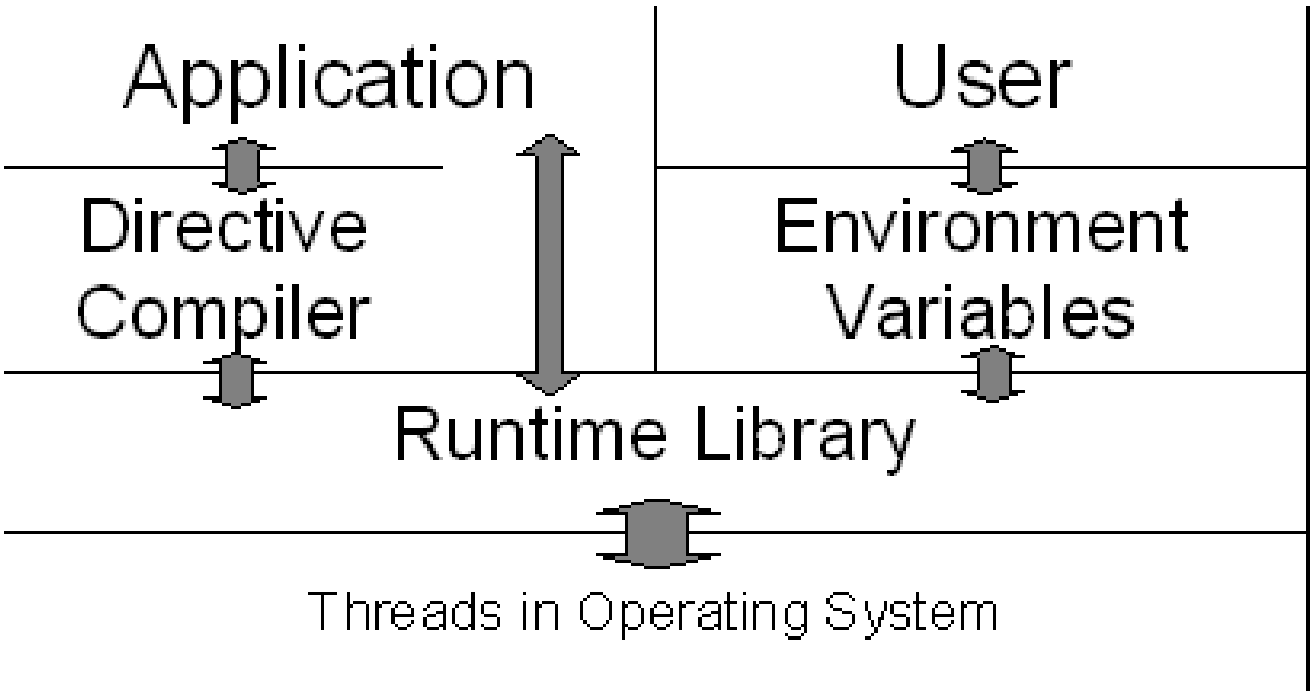
- !OMP, cOMP, *$OMP – fixed format
- !$OMP – free format
- Comments may *not* appear on the same line
- continue w/ &, e.g. !$OMP&

Specifying threads

- The simplest way to specify the number of threads used on a parallel region is to set the environment variable (in the shell where the program is executing)
  - OMP_NUM_THREADS
- For example, in csh/tcsh
  - setenv OMP_NUM_THREADS 4
- in bash
  - export OMP_NUM_THREADS=4
- Later we will cover other ways to specify this

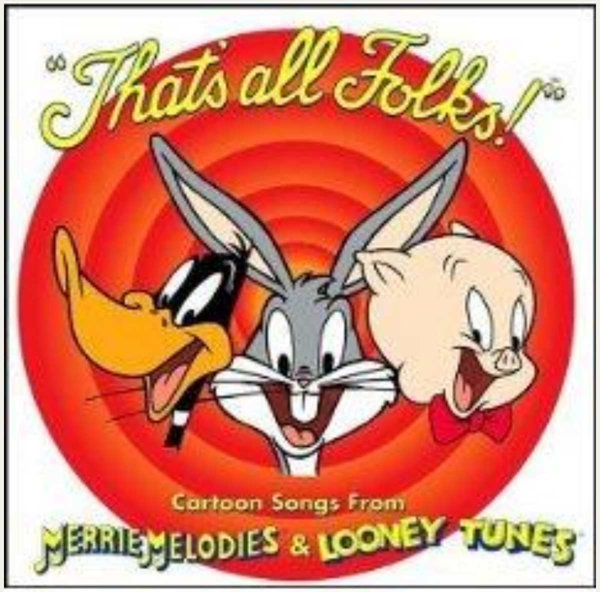
OpenMP - Diving In

OpenMP Architecture



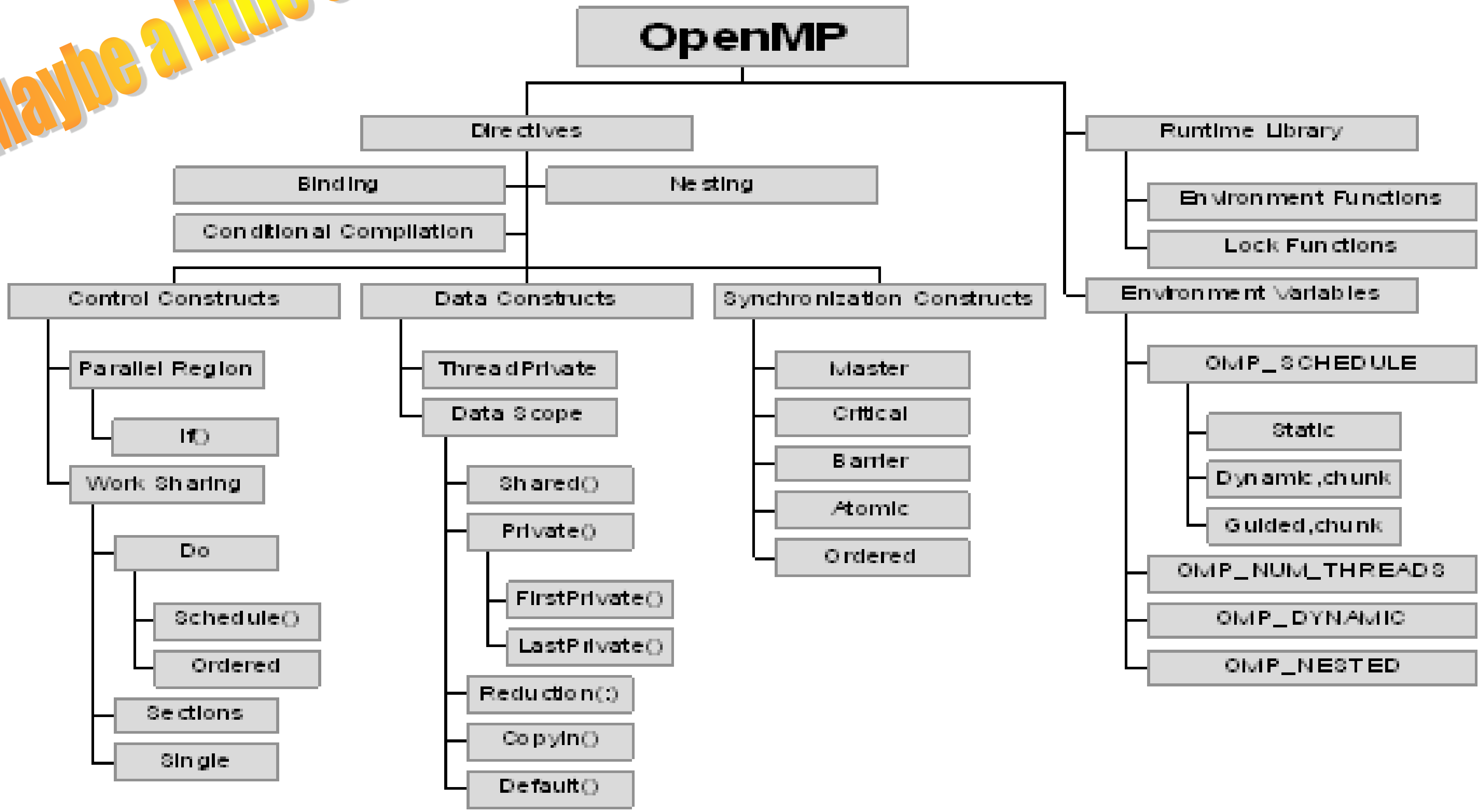
OpenMP Language Features

- Compiler Directives – 3 categories
  - Control Constructs
    - parallel regions, distribute work
  - Data Scoping for Control Constructs
    - control shared and private attributes
  - Synchronization
    - barriers, atomic, …
- Runtime Control
  - Environment Variables
    - OMP_NUM_THREADS
  - Library Calls
    - OMP_SET_NUM_THREADS(…)



1.A.3 OpenMP Constructs

Maybe a little outdated but ...



Parallel Construct

- Fortran
  - `!$OMP parallel [clause[[,] clause]… ]`
  - `!$OMP end parallel`
- C/C++
  - `#pragma omp parallel [clause[[,] clause]… ]`
  - `{structured block}`

Supported Clauses for the Parallel Construct

Valid Clauses:

- if (expression)
- num_threads (integer expression)
- private (list)
- firstprivate (list)
- shared (list)
- default (none|shared|private **fortran only**)
- copyin (list)
- reduction (operator: list)

Data Scoping Basics

- Shared
  - this is the default
  - variable exists just once in memory, all threads access it
- Private
  - each thread has a private copy of the variable
    - even the original is replaced by a private copy
  - copies are independent of one another, no information is shared
  - variable exists only within the scope it is defined

Worksharing Directives

- Loop (do/for)
- Sections
- Single
- Workshare (Fortran only)
- Task

UNC INFORMATION TECHNOLOGY SERVICES

Loop Worksharing directive

- Loop directives:
  - !$OMP DO [clause[[,] clause]…] — Fortran do
  - [!$OMP END DO [NOWAIT]] — optional end
  - #pragma omp for [clause[[,] clause]…] — C/C++ for
- Clauses:
  - PRIVATE(list)
  - FIRSTPRIVATE(list)
  - LASTPRIVATE(list)
  - REDUCTION({op|intrinsic}:list})
  - ORDERED
  - SCHEDULE(TYPE[, chunk_size])
  - NOWAIT

All Worksharing Directives

- Divide work in enclosed region among threads
- Rules:
  - must be enclosed in a parallel region
  - does not launch new threads
  - **no** implied barrier on entry
  - implied barrier upon exit
  - must be encountered by all threads on team or none

Loop Constructs

- Note that many of the clauses are the same as the clauses for the parallel region. Others are not, e.g. shared must clearly be specified before a parallel region.
- Because the use of parallel followed by a loop construct is so common, this shorthand notation is often used (note: directive should be followed immediately by the loop)
  - !$OMP parallel do …
  - !$OMP end parallel do
  - #pragma parallel for …

Shared Clause

- Declares variables in the list to be shared among all threads in the team.
- Variable exists in only 1 memory location and all threads can read or write to that address.
- It is the user's responsibility to ensure that this is accessed correctly, e.g. avoid race conditions
- Most variables are shared by default (a notable exception, loop indices)

Private Clause

- Private, uninitialized copy is created for each thread
- Private copy is not storage associated with the original

```
   program wrong
   I = 10
!$OMP parallel private(I)
   I = I + 1
!$OMP end parallel
   print *, I
```

Uninitialized!

Undefined!

Firstprivate Clause

- Firstprivate, initializes each private copy with the original

Initialized!

```
  program correct
  I = 10
!$OMP parallel firstprivate(I)
  I = I + 1
!$OMP end parallel
```

LASTPRIVATE clause

- Useful when loop index is live out
  - Recall that if you use PRIVATE the loop index becomes undefined

```
do i=1,N-1
  a(i)= b(i+1)
enddo
a(i) = b(0)
```

In Sequential case
i=N

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(i)
      do i=1,N-1
        a(i)= b(i+1)
      enddo
      a(i) = b(0)
!$OMP END PARALLEL
```

Changing the default

- List the variables in one of the following clauses
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE, LASTPRIVATE
  - DEFAULT
  - THREADPRIVATE, COPYIN

Default Clause

- Note that the default storage attribute is DEFAULT (SHARED)
- To change default: DEFAULT(PRIVATE)
  - each variable in static extent of the parallel region is made private as if specified by a private clause
  - mostly saves typing

USE THIS!

- DEFAULT(none): no default for variables in static extent. Must list storage attribute for each variable in static extent

NOWAIT clause

- **NOWAIT clause**

By default loop index is PRIVATE

```
!$OMP PARALLEL
!$OMP DO
      do i=1,n
        a(i)= cos(a(i))
      enddo
!$OMP END DO
!$OMP DO
      do i=1,n
        b(i)=a(i)+b(i)
      enddo
!$OMP END DO
!$OMP END PARALLEL
```

Implied BARRIER

```
!$OMP PARALLEL
!$OMP DO
      do i=1,n
        a(i)= cos(a(i))
      enddo
!$OMP END DO NOWAIT
!$OMP DO
      do i=1,n
        b(i)=a(i)+b(i)
      enddo
!$OMP END DO
```

No BARRIER

Reductions

- Assume no reduction clause

```
do i=1,N
  X = X + a(i)
enddo
```

Sum Reduction

Wrong!

```
!$OMP PARALLEL DO SHARED(X)
      do i=1,N
        X = X + a(i)
      enddo
!$OMP END PARALLEL DO
```

```
!$OMP PARALLEL DO SHARED(X)
      do i=1,N
!$OMP CRITICAL
        X = X + a(i)
!$OMP END CRITICAL
      enddo
!$OMP END PARALLEL DO
```

What's wrong?

REDUCTION clause

- Parallel reduction operators
  - Most operators and intrinsics are supported
  - Fortran: +, *, -, .AND. , .OR., MAX, MIN, …
  - C/C++ : +,*,-, &, |, ^, &&, ||
- Only scalar variables allowed

```
do i=1,N
  X = X + a(i)
enddo
```

```
!$OMP PARALLEL DO REDUCTION(+:X)
      do i=1,N
        X = X + a(i)
      enddo
!$OMP END PARALLEL DO
```

Ordered clause

- Executes in the same order as sequential code
- Parallelizes cases where ordering needed

```
do i=1,N
  call find(i,norm)
  print*, i,norm
enddo


1 0.45
2 0.86
3 0.65
```

```
!$OMP PARALLEL DO ORDERED PRIVATE(norm)
      do i=1,N
        call find(i,norm)
!$OMP ORDERED
        print*, i,norm
!$OMP END ORDERED
      enddo
!$OMP END PARALLEL DO
```

Schedule clause

- the Schedule clause controls how the iterations of the loop are assigned to threads
- There is always a trade off between load balance and overhead
- Always start with static and go to more complex schemes as load balance requires

The 4 choices for schedule clauses

- static: Each thread is given a “chunk” of iterations in a round robin order
  - Least overhead - determined statically
- dynamic: Each thread is given “chunk” iterations at a time; more chunks distributed as threads finish
  - Good for load balancing
- guided: Similar to dynamic, but chunk size is reduced exponentially
- runtime: User chooses at runtime using environment variable (note no space before chunk value)
  - setenv OMP_SCHEDULE “dynamic,4”

Performance Impact of Schedule

- Static vs. Dynamic across multiple do loops
  - In static, iterations of the do loop executed by the same thread in both loops
  - If data is small enough, may be still in cache, good performance
- Effect of chunk size
  - Chunk size of 1 may result in multiple threads writing to the same cache line
  - Cache thrashing, bad performance

```
!$OMP DO SCHEDULE(STATIC)
      do i=1,4

!$OMP DO SCHEDULE(STATIC)
      do i=1,4
```

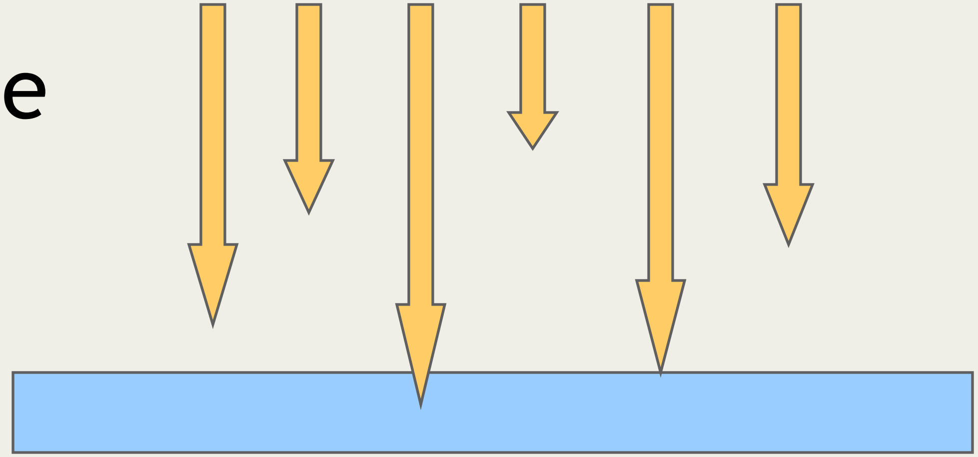
```
a(1,1) a(1,2)
a(2,1) a(2,2)
a(3,1) a(3,2)
a(4,1) a(4,2)
```

Synchronization

- Barrier Synchronization
- Atomic Update
- Critical Section
- Master Section
- Ordered Region
- Flush

Barrier Synchronization

- Syntax:
  - !$OMP barrier
  - #pragma omp barrier
- Threads wait until all threads reach this point
- implicit barrier at the end of each parallel region



Atomic Update

- Specifies a specific memory location can only be updated atomically, i.e. 1 thread at a time
- Optimization of mutual exclusion for certain cases (i.e. a single statement CRITICAL section)
  - applies only to the statement immediately following the directive
  - enables fast implementation on some HW
- Directive:
  - !$OMP atomic
  - #pragma atomic

Mutual Exclusion - Critical Sections

- Critical Section
  - only 1 thread executes at a time, others block
  - can be named (names are global entities and must not conflict with subroutine or common block names)
  - It is good practice to name them
  - all unnamed sections are treated as the same region
- Directives:
  - !$OMP CRITICAL [name]
  - !$OMP END CRITICAL [name]
  - #pragma omp critical [name]

Parallel Synchronization Model

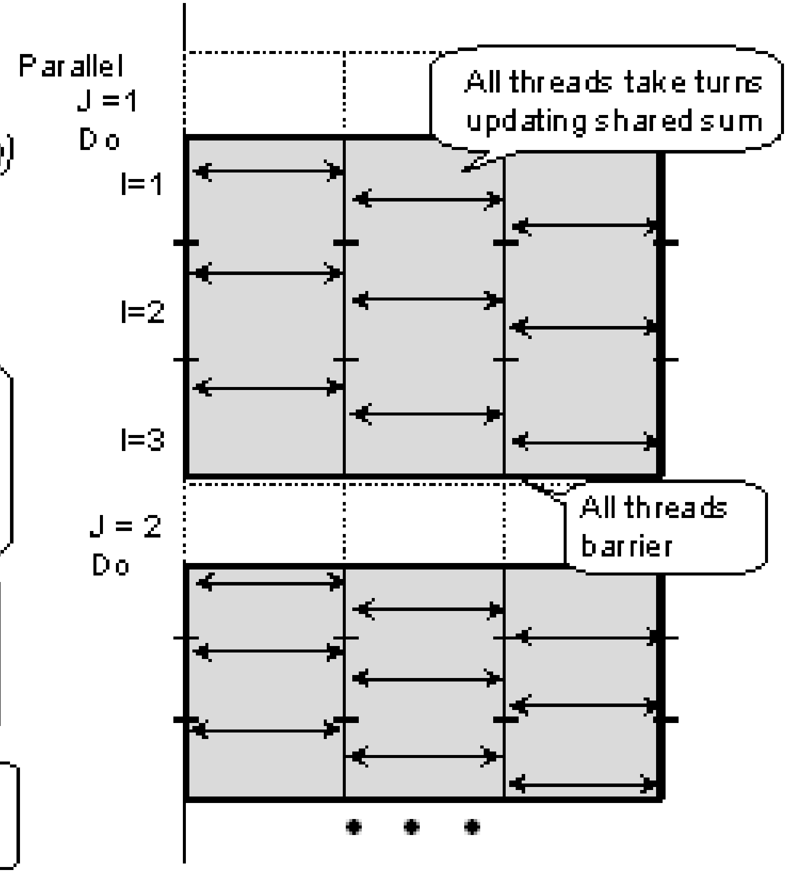
■ Critical section and Barrier

```
c$omp parallel private(i,j)shared(a,b,m,n,sum)
      do 20 j=1,n
c$omp do
         do 10 i=1,m
c$omp critical
            sum=sum+a(i,j)
c$omp end critical
10       continue
c$omp barrier
c$omp single
         b(j) = sum
c$omp end single
20     continue
c$omp end parallel
```

Mutual exclusion as threads update sum (Reduction and Atomic also available)

Wait until all threads have finished column sum. (Barrier actually built into end do.)

Single thread executes the update.



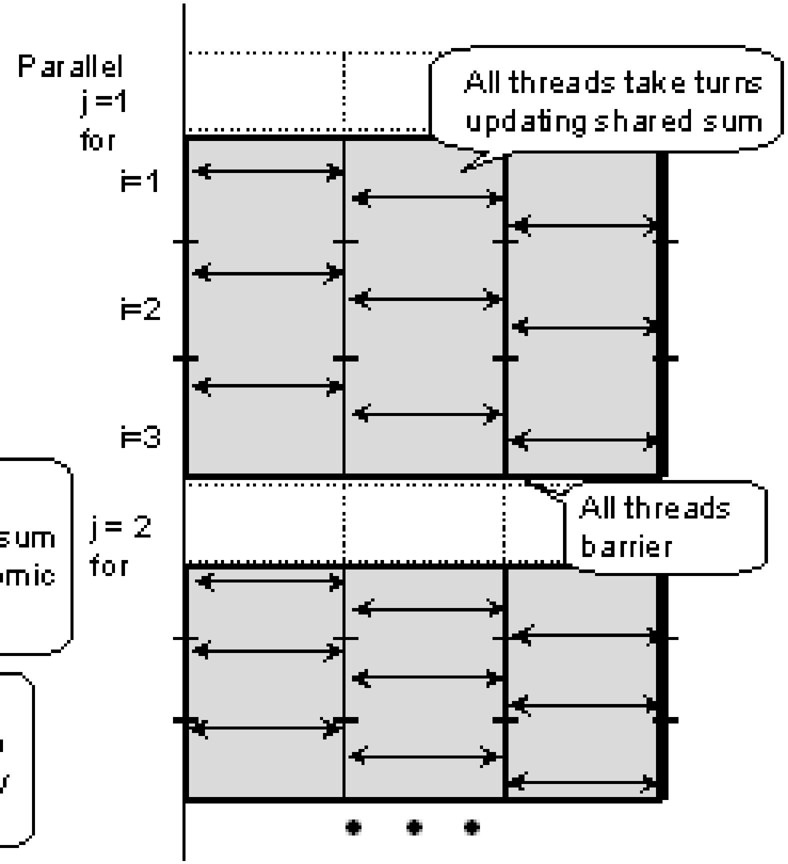
Parallel Synchronization Model in C

- Critical section and Barrier

```
#pragma omp parallel
    private(i,j)shared(a,b,m,n,sum)
     for( j=1; j<=n; j++ ) {
#pragma omp for
        for( i=1; i<=m; i++ ) {
#pragma omp critical
            sum=sum+a(i,j);
        }
#pragma omp barrier
        b(j) = sum;
     }
```

Mutual exclusion as threads update sum (Reduction and Atomic also available)

Wait until all threads have finished column sum. (Barrier actually built into end do.)



Clauses by Directives Table

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●				●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Sub-programs in parallel regions

- Sub-programs can be called from parallel regions
- static extent is code contained lexically
- dynamic extent includes static extent + the statements in the call tree
- the called sub-program can contain OpenMP directives to control the parallel region
  - directives in dynamic extent but not in static extent are called ***Orphan*** directives

Scope Definitions

```
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
      call whoami
```

***Static* extent of parallel region**

***Dynamic* extent includes *static* extent**

One compilation unit

```
      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

***Orphan* directive**

Different compilation unit

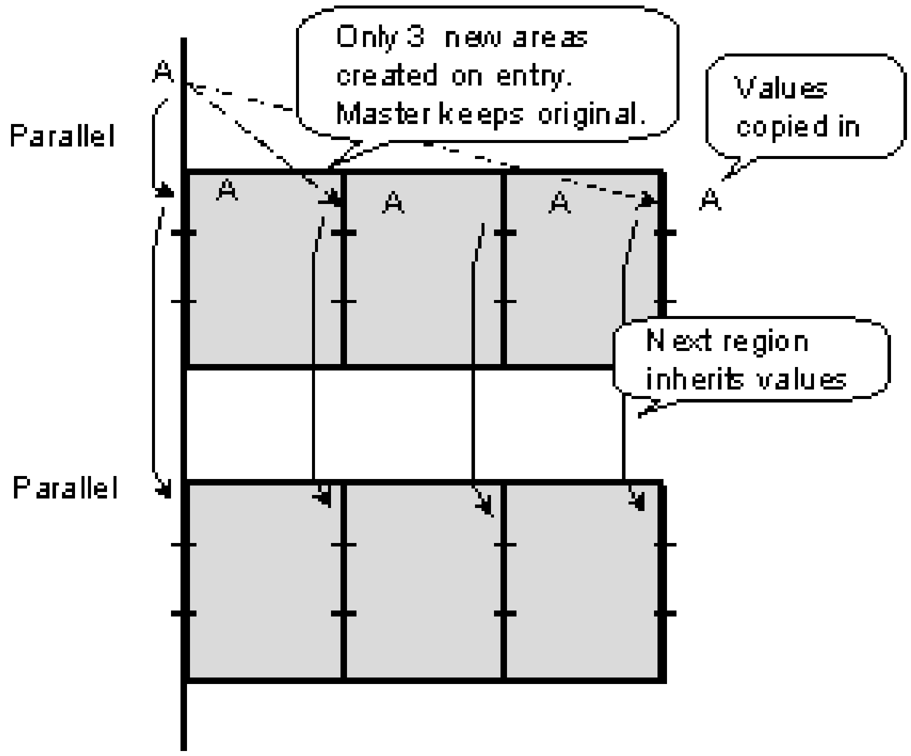
Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: file scope and static variables
- Different from making them PRIVATE
  - with PRIVATE global scope is lost
  - THREADPRIVATE preserves global scope for each thread
- Threadprivate variables can be initialized using COPYIN

Parallel Storage Model

- Note private common

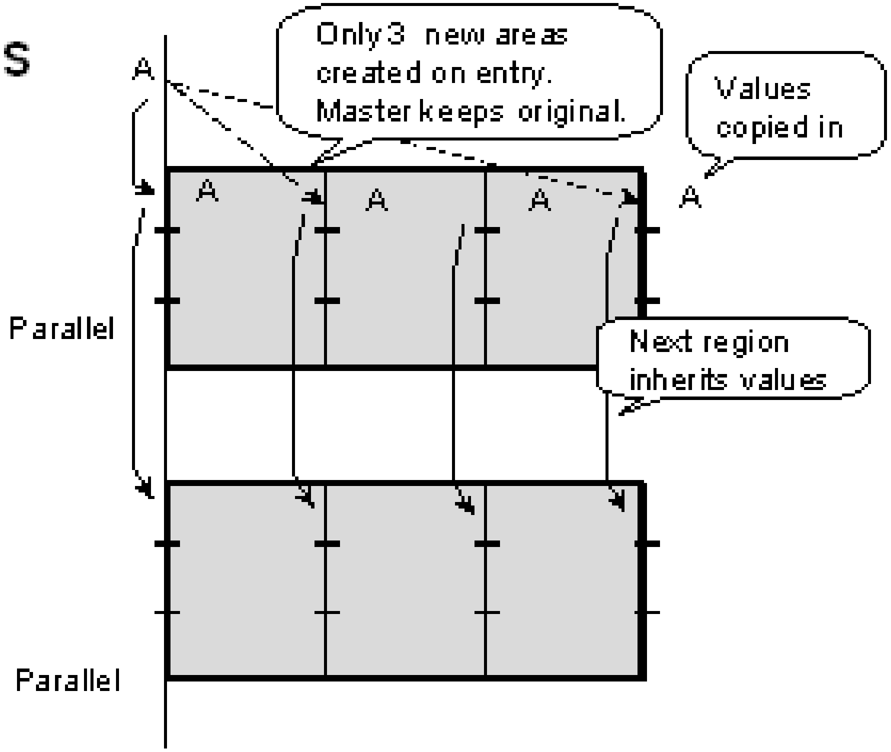
```
c$omp   threadprivate(/A/)
   common /A/
c$omp   parallel
c$omp & copyin (/A/)
c$omp   end parallel
c  Next parallel region
c$omp   parallel
c$omp   end parallel
```



Parallel Storage Model in C

- Private file-scope variables

```
#pragma omp                \
    threadprivate(A)
    struct Astruct A;
#pragma omp parallel \
   copyin (A)
{ . . . }
/* Next parallel region */
#pragma omp parallel
{ . . . }
```



Environment Variables

- These are set outside the program and control execution of the parallel code
- Prior to OpenMP 3.0 there were only 4
  - all are uppercase
  - values are case insensitive
- OpenMP 3.0 adds four new ones
- Specific compilers may have extensions that add other values
  - e.g. KMP* for Intel and GOMP* for GNU

Environment Variables

- OMP_NUM_THREADS – set maximum number of threads
  - integer value
- OMP_SCHEDULE – determines how iterations are scheduled when a schedule clause is set to “runtime”
  - “type[, chunk]”
- OMP_DYNAMIC – dynamic adjustment of threads for parallel regions
  - true or false
- OMP_NESTED – nested parallelism
  - true or false

Run-time Library Routines

- There are 17 different library routines, we will cover some of them now
- omp_get_thread_num()
  - Returns the thread number (w/i the team) of the calling thread. Numbering starts w/ 0.
- ***integer function omp_get_thread_num()***
- ***#include <omp.h>***

int omp_get_thread_num()

Run-time Library: Timing

- There are 2 portable timing routines
- ***omp_get_wtime***
  - portable wall clock timer returns a double precision value that is number of elapsed seconds from some point in the past
  - gives time per thread - possibly not globally consistent
  - difference 2 times to get elapsed time in code
- ***omp_get_wtick***
  - time between ticks in seconds

Run-time Library: Timing

- ***double precision function omp_get_wtime()***
- ***include <omp.h>***

  double omp_get_wtime()

- ***double precision function omp_get_wtick()***
- ***include <omp.h>***

  double omp_get_wtick()

Run-time Library Routines

- omp_set_num_threads(integer)
  - Set the number of threads to use in next parallel region.
  - can only be called from serial portion of code
  - if dynamic threads are enabled, this is the maximum number allowed, if they are disabled then this is the exact number used
- omp_get_num_threads
  - returns number of threads currently in the team
  - returns 1 for serial (or serialized nested) portion of code

Run-time Library Routines Cont.

- omp_get_max_threads
  - returns maximum value that can be returned by a call to omp_get_num_threads
  - generally reflects the value as set by OMP_NUM_THREADS env var or the omp_set_num_threads library routine
  - can be called from serial or parallel region
- omp_get_thread_num
  - returns thread number. Master is 0. Thread numbers are contiguous and unique.

Run-time Library Routines Cont.

- omp_get_num_procs
  - returns number of processors available
- omp_in_parallel
  - returns a logical (fortran) or int (C/C++) value indicating if executing in a parallel region

Run-time Library Routines Cont.

- omp_set_dynamic (logical(fortran) or int(C))
  - set dynamic adjustment of threads by the run time system
  - must be called from serial region
  - takes precedence over the environment variable
  - default setting is implementation dependent
- omp_get_dynamic
  - used to determine of dynamic thread adjustment is enabled
  - returns logical (fortran) or int (C/C++)

Run-time Library Routines Cont.

- omp_set_nested (logical(fortran) or int(C))
  - enable nested parallelism
  - default is disabled
  - overrides environment variable OMP_NESTED
- omp_get_nested
  - determine if nested parallelism is enabled
- There are also 5 lock functions which will not be covered here.

How many threads?

- Order of precedence:
  - if clause
  - num_threads clause
  - omp_set_num_threads function call
  - OMP_NUM_THREADS environment variable
  - implementation default (usually the number of cores on a node)

Weather Forecasting Example 1

```
!$OMP PARALLEL DO
!$OMP& default(shared)
!$OMP& private (i,k,l)
do 50 k=1,nztop
do 40 i=1,nx
cWRM remove dependency
cWRM l = l+1
   l=(k-1)*nx+i
   dcdx(l)=(ux(l)+um(k))
                *dcdx(l)+q(l)
40 continue
50 continue
!$OMP end parallel do
```

- Many parallel loops simply use parallel do
- autoparallelize when possible (usually doesn't work)
- ***simplify code by removing unneeded dependencies***
- Default (shared) simplifies shared list but Default (none) is recommended.

Weather - Example 2a

```
cmass = 0.0
!$OMP parallel default
   (shared)
!$OMP&
   private(i,j,k,vd,help,..
   )
!$OMP& reduction(+:cmass)
   do 40 j=1,ny
!$OMP do
   do 50 i=1,nx
       vd = vdep(i,j)
do 10 k=1,nz
       help(k) = c(i,j,k)
10 continue
```

- Parallel region makes nested do more efficient
  - avoid entering and exiting parallel mode
- Reduction clause generates parallel summing

Weather - Example 2a Continued

```
…

do 30 k=1,nz
   c(i,j,k)=help(k)
   cmass=cmass+help(k)
30 continue
50 continue
!$OMP end do
40 continue
!$omp end parallel
```

- Reduction means
  - each thread gets private cmass
  - private cmass added at end of parallel region
  - serial code unchanged

Weather Example - 3

```
!$OMP parallel
do 40 j=1,ny
!$OMP do schedule(dynamic)
do30 i=1,nx
   if(ish.eq.1)then
       call upade(…)
   else
       call ucrank(…)
   endif
30 continue
40 continue
!$OMP end parallel
```

- Schedule(dynamic) for load balancing

Weather Example - 4

```
!$OMP parallel !don't it
   slows down
!$OMP& default(shared)
!$OMP& private(i)
do 30 I=1,loop
   y2=f2(I)
   f2(i)=f0(i) +
   2.0*delta*f1(i)
   f0(i)=y2
30 continue
!$OMP end parallel do
```

- Don't over parallelize small loops
- Use if(<condition>) clause when loop is sometimes big, other times small

Weather Example - 5

```
!$OMP parallel do
   schedule(dynamic)
!$OMP& shared(…)
!$OMP& private(help,…)
!$OMP& firstprivate
   (savex,savey)
do 30 i=1,nztop
…
30 continue
!$OMP end parallel do
```

- First private (…) initializes private variables