

12.950

Parallel Programming
for
Multicore Machines
Using
OpenMP and MPI

Dr. C. Evangelinos
MIT/EAPS

Course basics

- Web site: <http://stellar.mit.edu/S/course/12/ia10/12.950/>
- <https://wikis.mit.edu/confluence/display/12DOT950ia10/Home>
- Homeworks: One per day, incremental, finally due **Feb 1**.
- Discussion on homework problems during next class
- Grade: A/B/C etc.
- Textbook: none! But suggested books on Web site.
- Look at a lot of other support material on the Web site, including instructions about virtual machines.
- Please sign up with your name, Athena e-mail (if existing) and whether you want to be a listener or not. You can change this before the end of the course.

StarHPC

- A VMware Player/VirtualBox image with OpenMPI and the GNU and Sun compilers for OpenMP for development alongside Eclipse PTP and SunStudio 12/Netbeans for an IDE. Link to download the virtual machine will appear on the class website.
- <http://web.mit.edu/star/hpc/> contains detailed instructions on using the Virtual Machines.
- E-mail star@mit.edu for support and troubleshooting.

Course Syllabus

- Day 1 (Parallel Computing and OpenMP):
 - Fundamentals of Shared Memory Programming
 - Basic OpenMP concepts, PARALLEL directive
 - Data scoping rules
 - Basic OpenMP constructs/directives/calls
 - Examples
 - Parallelizing an existing code using OpenMP
 - More advanced OpenMP directives & functions
 - OpenMP Performance issues

Syllabus cont.

- Day 2 (Parallel Computing and MPI Pt2Pt):
 - OpenMP 3.0 enhancements
 - Fundamentals of Distributed Memory Programming
 - MPI concepts
 - Blocking Point to Point Communications
- Day 3 (More Pt2Pt & Collective communications):
 - Paired and Nonblocking Point to Point Communications
 - Other Point to Point routines
 - Collective Communications: One-with-All
 - Collective Communications: All-with-All

Syllabus cont.

- Day 4 (advanced MPI-1):
 - Collective Communications: All-with-All
 - Derived Datatypes
 - Groups, Contexts and Communicators
 - Topologies
 - Language Binding issues
 - The Runtime and Environment Management
 - The MPI profiling interface and tracing

Syllabus cont.

- Day 5 (more MPI-1 & Parallel Programming):
 - Hybrid MPI+OpenMP programming
 - MPI Performance Tuning & Portable Performance
 - Performance concepts and Scalability
 - Different modes of parallelism
 - Parallelizing an existing code using MPI
 - Using 3rd party libraries or writing your own library

Outline

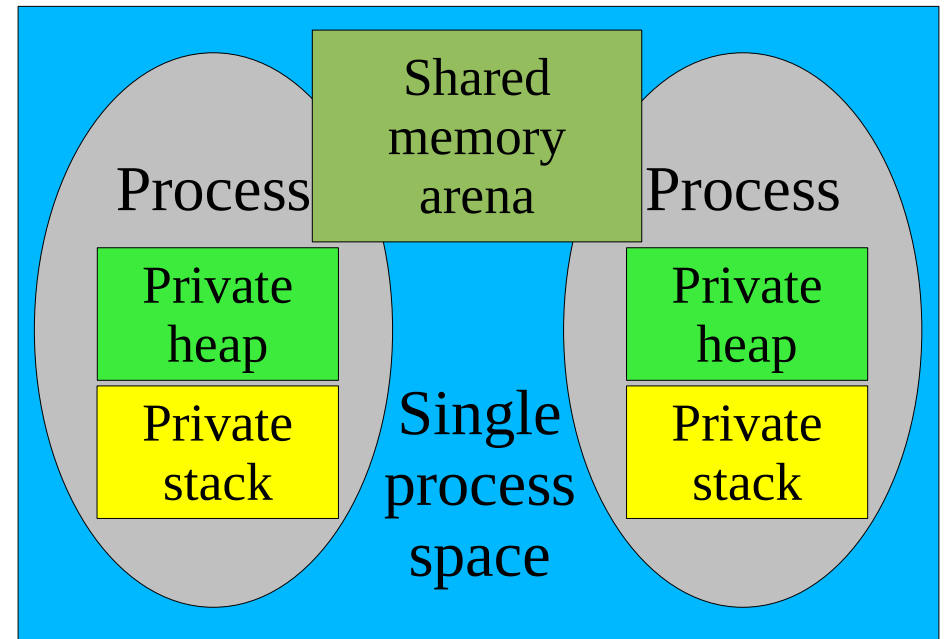
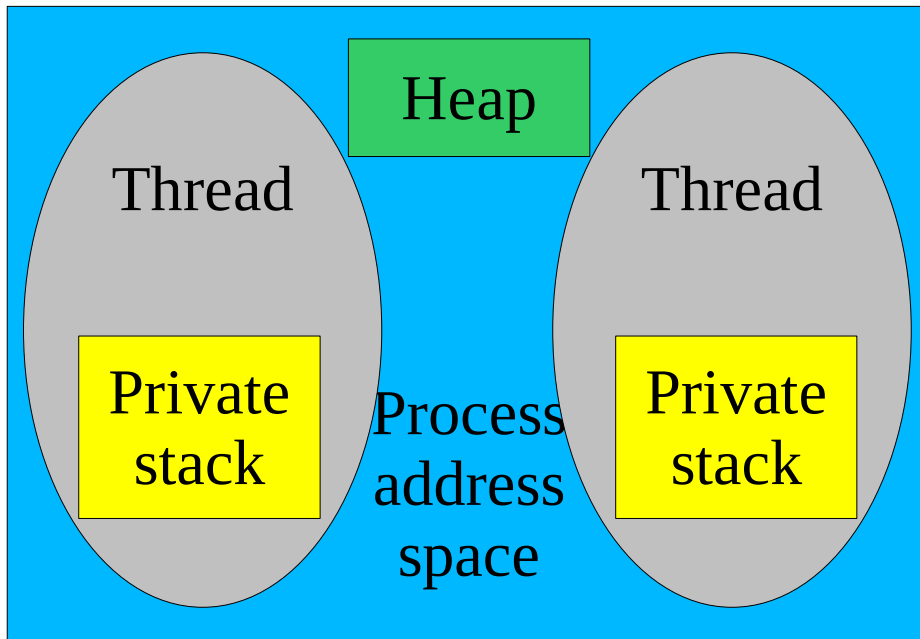
- Fundamentals of Shared Memory Programming
- Basic OpenMP concepts, PARALLEL directive
- Data scoping rules
- Basic OpenMP constructs/directives/calls
- Examples
- Parallelizing an existing code using OpenMP
- More advanced OpenMP directives & functions
- OpenMP Performance and Correctness issues

Acknowledgments

- The OpenMP ARB
- Tim Mattson (Intel) & Rudolf Eigenmann (Purdue)
- Miguel Hermanss (UP Madrid)
- Ruud van der Pas (Sun Micro)
- NERSC, LLNL

Shared Memory Programming

- Under the assumption of a single address space one uses multiple control streams (threads, processes) to operate on both private and shared data.
- Shared data: synchronization, communication, work
 - In shared arena/mmaped file (multiple processes)
 - In the heap of the process address space (multiple threads)



OpenMP programming model

- The OpenMP standard provides an API for shared memory programming using the fork-join model.
- Multiple threads within the same address space
- Code parallelization can be incremental
- Supports both coarse and fine level parallelization
- Fortran, C, C++ support

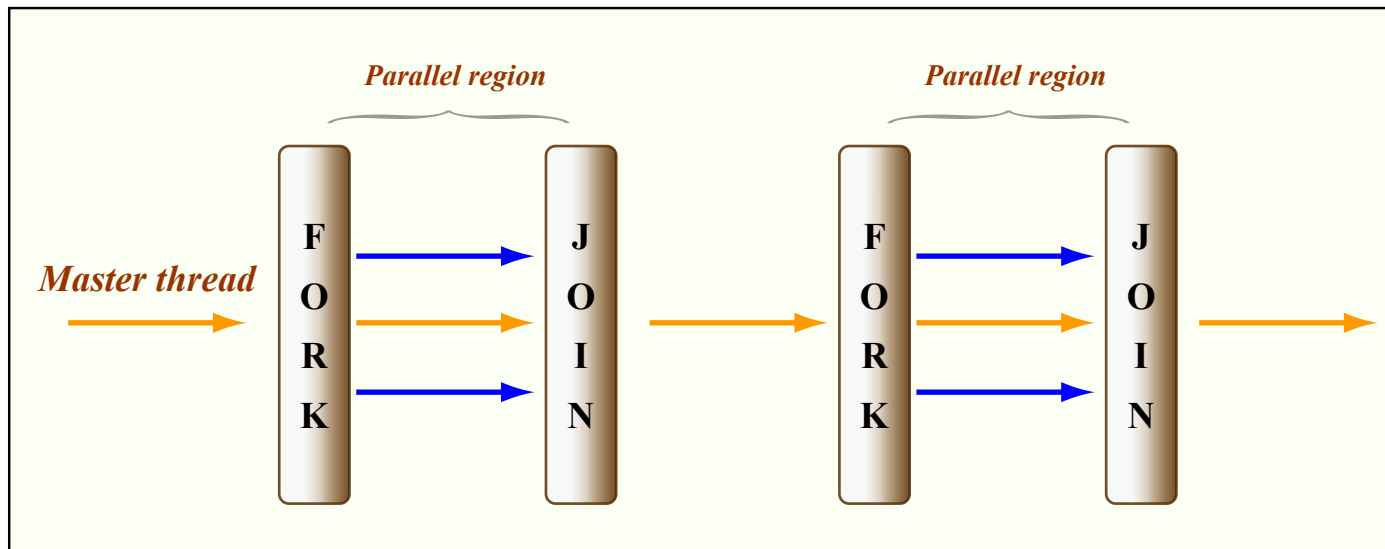


Figure by MIT OpenCourseWare.

OpenMP conceptual overview

- Threads read and write shared variables
 - No need for explicit communications with messages
 - Use synchronization to protect against race conditions
 - Shared variable scope attributes help minimize the necessary synchronization
 - No sense to try and run in parallel loops with loop-carried dependencies
- Threads use their own private variables to do work that does not need to be globally visible outside the parallel region
- No support for proper parallel I/O

OpenMP history

- During the early multiprocessor days: vendor specific
 - Early standardization efforts: PCF, ANSI X3H5
 - SGI/Cray merger gave impetus to new efforts
- OpenMP: **Open Multi Processing**
 - ARB (Architecture Review Board):
 - Software/hardware vendors, ISVs & DOE/ASCI
- Evolving standard: currently common at 3.0, some compilers still implement just Fortran 1.1, C/C++ 1.0. Most Fortran 2.0 or 2.5 however. Version 3.0 support is becoming more common.
- JOMP for Java (academic project)
- OpenMP uses compiler directives & library routines

Directives

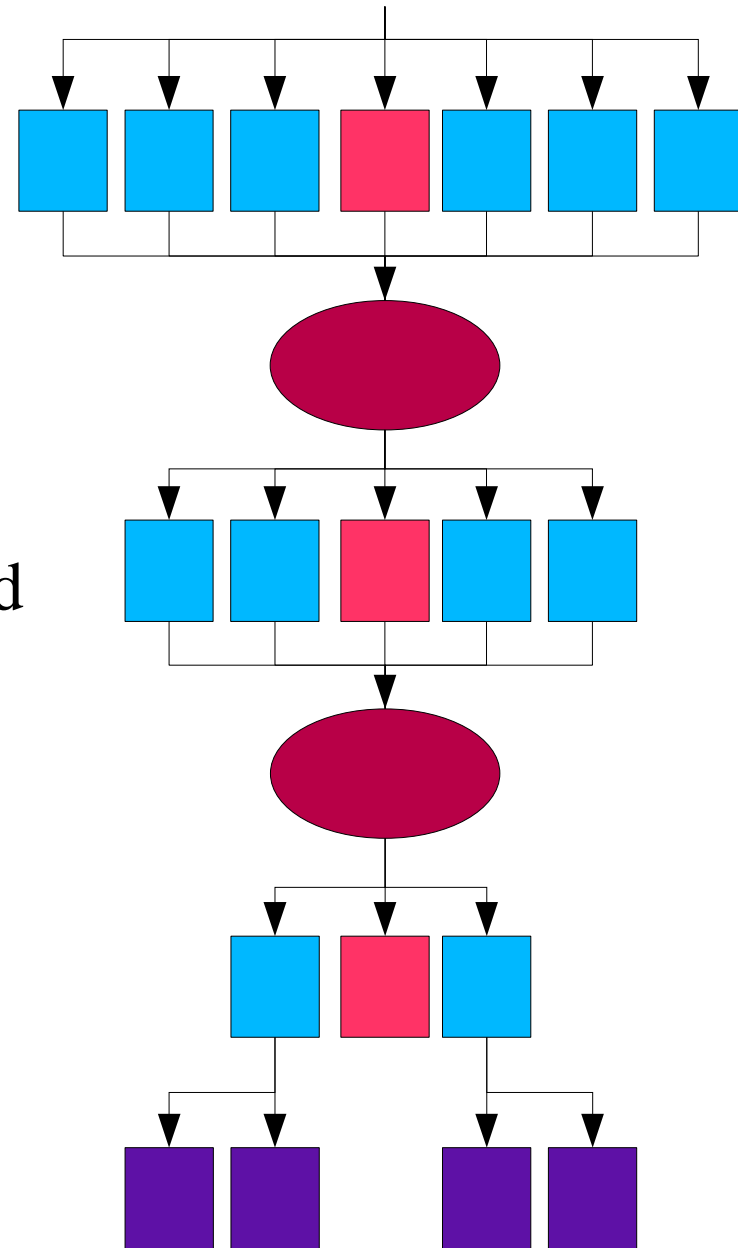
- Directives are additions to the source code that can be ignored by the compiler:
 - Appearing as comments (OpenMP Fortran)
 - Appearing as preprocessor macros (OpenMP C/C++)
 - Hence a serial and a parallel program can share the same source code - the serial compiler simply overlooks the parallel code additions
 - Addition of a directive does not break the serial code
 - However the wrong directive or combination of directives can give rise to parallel bugs!
 - Easier code maintenance, more compact code
 - New serial code enhancements outside parallel regions will not break the parallel program.

OpenMP Library and Environment

- Aside from directives, OpenMP runtime library:
 - Execution environment routines:
 - Who am I? How many of us are there? How are we running?
 - Lock routines
 - Timing routines
 - Environment variables are another, easy way to modify the parallel program's behavior from the command line, before execution.
 - Nested parallelism is allowed
 - A dynamic change in the number of threads is allowed before a parallel region is entered

Nested and Dynamic Parallelism

- By default, the original number of forked threads is used throughout
- If `omp_set_dynamic()` is used or `OMP_DYNAMIC` is `TRUE`, this number can be reset.
- Nested parallel constructs are run serialized unless `omp_set_nested()` is used or `OMP_NESTED` is `TRUE`.
- Implementations are not required to implement either: Use functions `omp_get_dynamic/omp_get_nested`



Basics of Directives

- Composed of: **sentinel** *construct* [**clauses**]
 - Fortran fixed form: C\$OMP, c\$OMP, *\$OMP, !\$OMP
 - Fortran free form: !\$OMP
 - Standard Fortran continuation characters allowed

```
C$OMP parallel default(none) shared(a,b) private(c,d)
```

```
C$OMP& reduction(a,+)
```

- C/C++: #pragma omp
- Preprocessor macros allowed after #pragma omp
- Continuation accomplished with \

```
#pragma omp parallel default(none) shared(a,b) \
```

```
reduction(a,+)
```

- Clause order is immaterial

Conditional Compilation

- `_OPENMP` macro defined by the compiler
- Conditional compilation of OpenMP library calls
- Enclose within:

```
#ifdef _OPENMP
```

```
/* C/C++ (or Fortran code) calling OpenMP runtime lib */
```

```
whoami = omp_get_thread_num() + 1;
```

```
#endif
```

- Precede by `!$/C$/c$/*$` in Fortran:

```
!$      whoami = omp_get_thread_num() + &
```

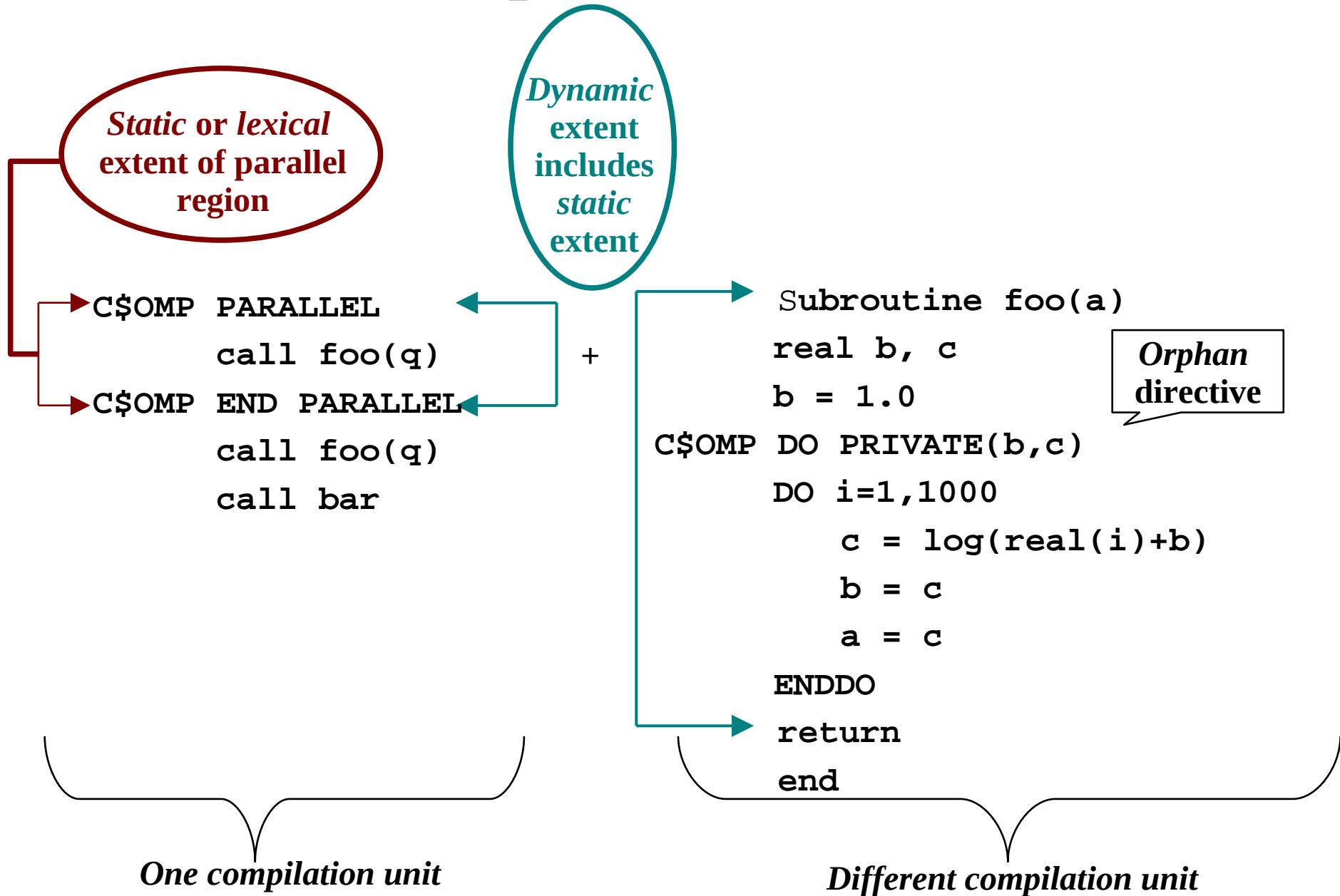
```
!$    & 1
```

- Fixed form formatting rules apply

Static and Dynamic Extent

- Directives apply to the structured block that follows:
 - Structured block:
 - 1 point of entry, 1 point of exit
 - Illegal to branch out of block, only `exit()`, `stop` allowed
 - For Fortran: next line or marked with an `!$OMP END`
 - For C/C++: next line or enclosed in `{ }`
 - Lexical scope forms the *static* extent.
- Any function/subroutine calls within a block give rise to a dynamic extent that the directive also applies to.
- *Dynamic* extent includes static extent + statements in call tree
- The called code can contain further OpenMP directives
 - A directive in a dynamic but not a static extent is called *orphan(ed)*

Orphan directives



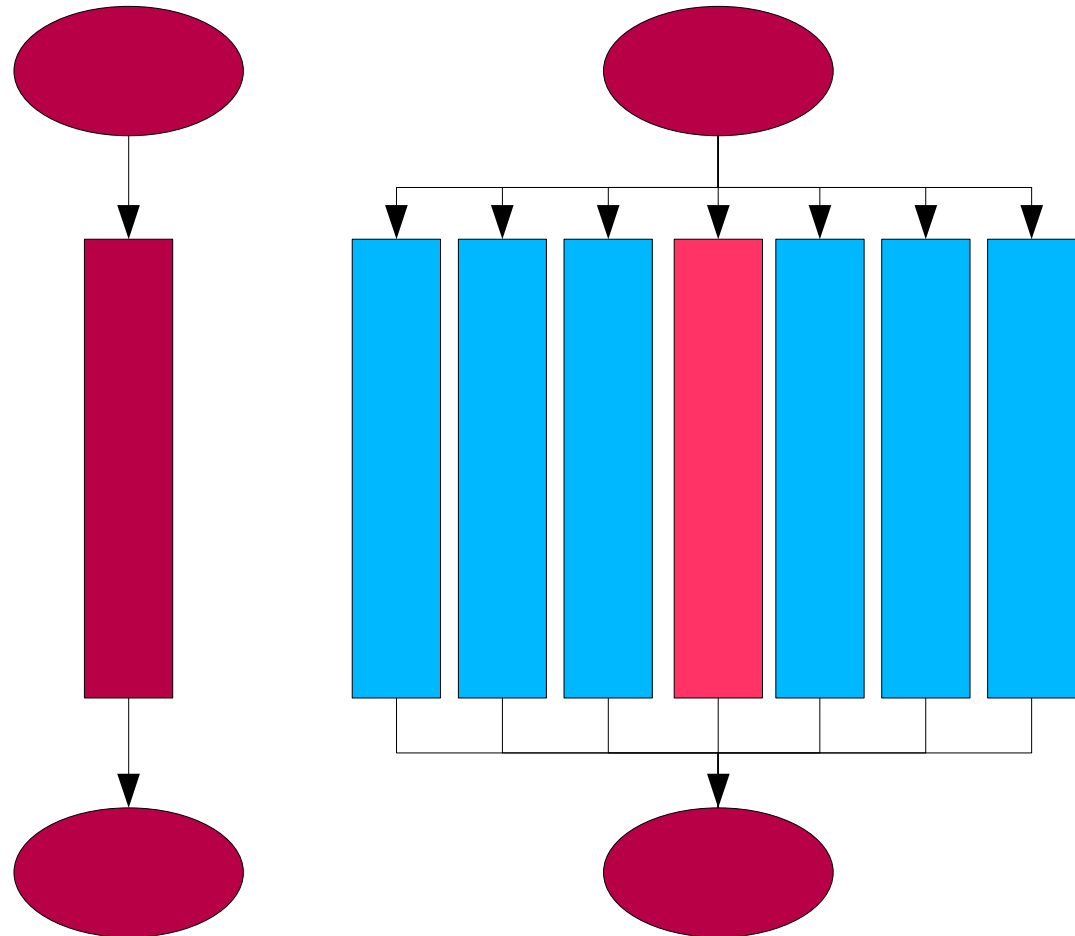
Parallel Directive

Most basic of all directives, it implies the creation of a team of extra threads to execute the structured block:

```
!$OMP PARALLEL [clauses]
!some Fortran structured block
!$OMP END PARALLEL
```

```
#pragma omp parallel [clauses]
{
/* some C/C++ structured block
*/
}
```

- Structured block:
- Implicit barrier at the end
- Whole block executed in parallel
- Master thread forks slaves, and participates in parallel computation
- At the end of the region, slaves go to sleep, spin or are destroyed



Parallel Directive Example

```
PROGRAM PARALLEL
```

```
IMPLICIT NONE
```

Look at hello_omp.f90

```
!$OMP PARALLEL
```

```
write (6,*) "hello world!"
```

```
!$OMP END PARALLEL
```

```
END PROGRAM PARALLEL
```

To run on a system:

```
% setenv OMP_NUM_THREADS 4
```

```
% sunf90 -xopenmp hello_omp.f90
```

```
% ./a.out
```

```
hello world!
```

```
hello world!
```

```
hello world!
```

```
hello world!
```

Parallel Directive Clauses

- Control clauses:
 - IF (scalar_logical_expression)
 - Conditional parallel execution (is there enough work to do?)
 - **NUMTHREADS**(scalar_logical_expression)
 - Hardcodes the number of threads (useful for sections)
 - Overrides runtime library call, environment variable
- Data sharing attribute clauses (lowercase for C/C++)
 - PRIVATE (list), SHARED (list)
 - DEFAULT (**PRIVATE** | SHARED | NONE)
 - FIRSTPRIVATE (list)
 - REDUCTION (operator: list)
 - COPYIN (list)

Example of IF clause

```
Program ompif
```

```
integer N
```

```
read (5,*) N
```

```
!$OMP PARALLEL IF(N > 1000)
```

```
write(6,*) "Here I am running"
```

```
!$OMP END PARALLEL
```

```
END
```

```
To run with 3 threads:
```

```
% setenv OMP_NUM_THREADS 3
```

```
% ./a.out
```

```
5
```

```
Here I am running
```

```
% ./a.out
```

```
1000000
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

Look at ompif.f90

```
To run with 9 threads:
```

```
% setenv OMP_NUM_THREADS 9
```

```
% ./a.out
```

```
5
```

```
Here I am running
```

```
% ./a.out
```

```
1000000
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```

```
Here I am running
```


Default Data Sharing Attributes

- Threads share global variables
 - Fortran: **COMMON** blocks, **SAVE** variables, **MODULE** variables
 - C: File scope variables, static, storage on the heap
- Stack (automatic) variables are private:
 - Automatic variables in a structured block
 - Local variables in subroutines called from parallel
 - Local pointers pointing to heap storage
- Loop index variables are by default private
- Defaults can be changed with the **DEFAULT** clause
 - Default cannot be private in C/C++

DEFAULT example

```
iwork = 1000
common /stuff/ a(iwork)
C$OMP PARALLEL PRIVATE(np, ieach)
np = omp_get_num_threads()
ieach = iwork/np
call foo(ieach, iwork, np)
C$OMP END PARALLEL
```

```
subroutine foo(ido, iwork, np)
integer ido, iwork, np
common /input/ A(iwork)
real temp(ido)
DO i = 1, ido
    temp(i) = A((np-1)*ido+i)
write(6,*) temp(i)
ENDDO
```

```
iwork = 1000
common /stuff/ a(iwork)
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(a, iwork)
np = omp_get_num_threads()
ieach = iwork/np
call foo(ieach, iwork, np)
C$OMP END PARALLEL
```

DEFAULT(NONE) serves as a way of forcing the user to specify the data attribute for each variable to avoid bugs.

PRIVATE clause

- A private variable is uninitialized upon entry to the parallel region
- There is no storage association with the variable outside the region
- However at the end of the parallel region the outside variable's value cannot be defined on the basis of its prior to the parallel region value.
- The example to the right contains

many problems:

- The value of A is uninitialized
- The value of A is undefined

Look at problem-private.f

```
program problem
  real A
  A = 10.0
  C$OMP PARALLEL PRIVATE(A)
  A = A + LOG(A)
  C$OMP END PARALLEL
  print *, A
end
```


→

→

FIRSTPRIVATE clause

- FIRSTPRIVATE is a variation on PRIVATE with the value of the variable on each thread initialized by the value of the variable outside the parallel region **Look at problem-firstprivate.f**
- This solves one of the two problems seen before but the final value is still undefined, could be 10.0
- This can be corrected in the case of DO/for or SECTION worksharing constructs through use of the LASTPRIVATE clause and for SINGLE through the **COPYPRIVATE clause**

```
program problem
  real A
  A = 10.0
  C$OMP PARALLEL FIRSTPRIVATE(A)
    A = A + LOG(A)
  C$OMP END PARALLEL
  print *, A
end
```



REDUCTION clause

- It enables reduction binary operations (arithmetic, logical and intrinsic procedures like MAX) in an optimal manner (as they require atomic updates).
- Scalar variables are initialized to relevant values and at the end of the loop the value of the variable before the parallel execution is also included in the reduction.

```
!$OMP PARALLEL DEFAULT(PRIVATE) REDUCTION(+:I) &
```

```
!$ & REDUCTION(*:J) REDUCTION(MAX:K)
```

```
    tnumber=OMP_GET_THREAD_NUM( )
```

```
    I = I + tnumber
```

```
    J = J * tnumber
```

Look at reduction.f90

```
    K = MAX(K, tnumber)
```

```
!$OMP END PARALLEL
```

```
PRINT *, " I=", I, " J=", J, " K=", K
```

Worksharing constructs

- Worksharing constructs allow us to distribute different work to threads in a parallel region:

Iterative worksharing:

```
!$OMP DO
    DO i=1,N
        ! some fortran work
    ENDDO
!$OMP END DO

!$OMP WORKSHARE
    FORALL (i=1,N)
        ! some Fortran 90/95 work
!$OMP END WORKSHARE
```

Non-iterative worksharing:

```
!$OMP SECTIONS
!$OMP SECTION
    ! some fortran work
!$OMP SECTION
    ! some other fortran work
!$OMP END SECTIONS
```

Serial work by any processor:

```
!$OMP SINGLE
    !some serial work
!$OMP END SINGLE
```

Iterative worksharing:

```
#pragma omp for
    for (i=0; i<N; i++) {
        /* some C/C++ work */
    }
```

Non-iterative worksharing:

```
#pragma omp sections
{
    #pragma omp section
        {
            /* some C/C++ work */
        }
    #pragma omp section
        {
            /* some other C/C++ work */
        }
}
```

Serial work by any processor:

```
#pragma omp single
{
    /* some serial work */
}
```

DO/for construct

```
C$OMP PARALLEL PRIVATE(mytid,many,ntill,i)
mytid = omp_get_thread_num()
many = N/omp_get_num_threads()
ntill = (mytid+1)*many -1
do i=mytid*many, ntill
    write(6,*) foo
enddo
```

```
C$OMP PARALLEL
C$OMP DO
do i=1,N
    write(6,*) foo
enddo
C$OMP END DO
C$OMP END PARALLEL
```

C\$OMP END PARALLEL
C/C++ for loops need to be in canonical shape:

- initialization
- comparison test (order)
- increment of loop index
- loop limits are invariant
- increment is invariant

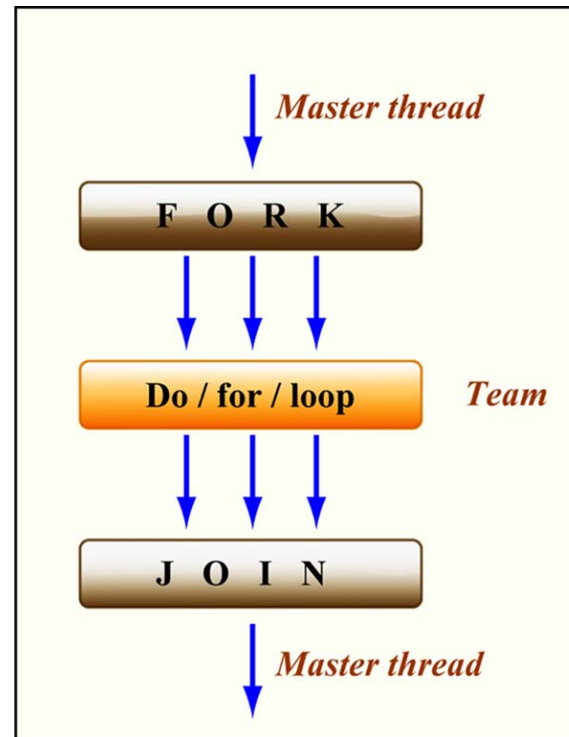


Figure by MIT OpenCourseWare.

DO/for clauses

- The DO/for construct has the following clauses:
 - Data scope attribute clauses (lowercase for C/C++)
 - PRIVATE (list), SHARED (list)
 - FIRSTPRIVATE (list), LASTPRIVATE (list)
 - REDUCTION (operator: list)
 - COPYIN (list)
 - Execution control clauses
 - SCHEDULE (type, chunk)
 - ORDERED
 - NOWAIT (*at the !\$OMP END DO for Fortran*)

WORKSHARE construct

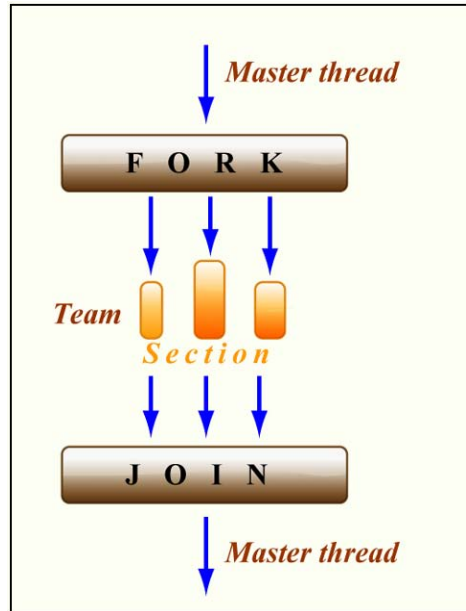
- Added in **OpenMP 2.0** to cover F90/F95
- Works on array notation operations, forall & where statements & constructs and transformational array intrinsics like matmul, dot_product, cshift etc.
- Can only include **ATOMIC** and **CRITICAL** directives
- Private variables cannot be modified inside the block
- Applies only to lexical scope
- No function/subroutine calls inside the block
- There is an implicit barrier
after every array statement

```
!$OMP WORKSHARE
  A = B + 1.0
  FORALL (i=1:100:2) B = A-1.5
  WHERE (A .NE. 1.5) A = B
!$OMP END WORKSHARE NOWAIT
```

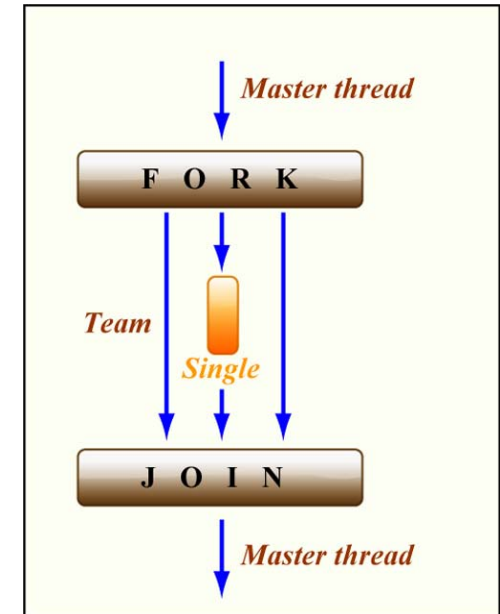
SECTIONS & SINGLE constructs

```
C$OMP PARALLEL PRIVATE(mytid)
mytid = omp_get_thread_num()
if (mytid .eq. 0) then
    call foo
endif
if (mytid .eq. 1) then
    call bar
endif
C$OMP END PARALLEL
```

The SINGLE construct allows code that is serial in nature to be executed inside a parallel region. The thread executing the code will be the first to reach the directive in the code. It doesn't have to be the master thread. All other threads proceed to the end of the structured block where there is an implicit synchronization.



```
C$OMP PARALLEL
C$OMP SECTIONS
call foo
C$OMP SECTION
call bar
C$OMP END PARALLEL
```



Figures by MIT OpenCourseWare.

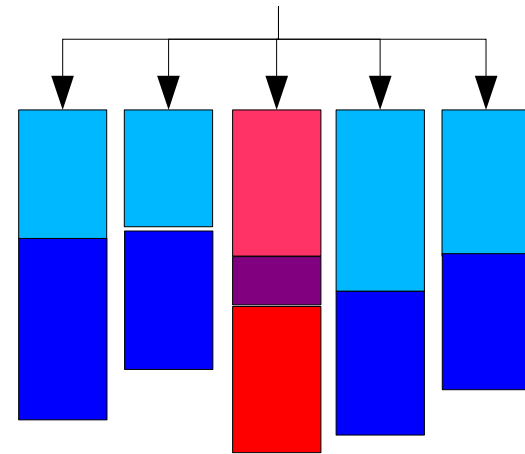
SECTION & SINGLE clauses

- The SECTION construct has the following clauses:
 - Data scope attribute clauses (lowercase for C/C++)
 - PRIVATE (list), FIRSTPRIVATE (list), LASTPRIVATE (list)
 - REDUCTION (operator: list)
 - COPYIN (list)
 - NOWAIT (*at the !\$OMP END SECTION for Fortran*)
- The SINGLE construct has the following clauses:
 - Data scope attribute clauses (lowercase for C/C++)
 - PRIVATE (list), FIRSTPRIVATE (list)
 - NOWAIT (*at the !\$OMP END SINGLE for Fortran*)
 - COPYPRIVATE(list) (*at the !\$OMP END SINGLE for Fortran, cannot coexist with NOWAIT*)

MASTER

- Only the master thread executes the section
 - The rest of the threads proceed to continue execution from the end of the master section
 - There is no barrier at the end of the master section
 - Same as SINGLE NOWAIT but only for master thread

```
C$OMP PARALLEL          #pragma omp parallel
  ..                    {
C$OMP MASTER            ..
  print*, "Init" #pragma omp master
C$OMP END MASTER       printf("Init\n");
  ..                    ..
C$OMP END PARALLEL     }
```



Not really a synchronization construct!

Use of NOWAIT

- Implicit synchronizations at the end of worksharing constructs, even in the absence of an `!$OMP END` directive.
- Sometimes unnecessary - user can specify no synchronization using `NOWAIT` **judiciously**.
- Similar care needs to be taken when using shared variables inside a loop on both RHS & LHS.

```
!$OMP DO
    DO i=1, N
        A(i) = log(B(i))
    ENDDO
!$OMP END DO NOWAIT
```

```
!$OMP DO
    DO i=1, M
        C(i) = EXP(D(i))
    ENDDO
!$OMP END DO NOWAIT
```

```
!$OMP DO
    DO i=1, M
        D(i) = C(i)/D(i)
    ENDDO
!$OMP END DO
```

```
!$OMP DO
    DO i=1, N
        B(i) = 1.0*i+ 0.5
        A(i) = A(i)*B(i)
    ENDDO
!$OMP END DO
```

LASTPRIVATE clause

- `LASTPRIVATE(variable)` will make sure that what would have been the last value of the private variable if the loop had been executed sequentially gets assigned to the variable outside the scope
- **Look at the sequence of problem-*.f files in the homework.**

```
program problem
real A(4)
DO I=1,4
    A(I) = 10.0
ENDDO
C$OMP PARALLEL FIRSTPRIVATE(A)
C$OMP DO LASTPRIVATE(A)
DO I=1,4
    A(I) = A(I) + LOG(A(I))
ENDDO
C$OMP END DO
print *, "region result is ", A
C$OMP END PARALLEL
print *, "result is ", A
end
```

```
program problem
real A(4)
DO I=1,4
    A(I) = 10.0
ENDDO
C$OMP PARALLEL DO FIRSTPRIVATE(A) C$
&LASTPRIVATE(A)
DO I=1,4
    A(I) = A(I) + LOG(A(I))
ENDDO
C$OMP END PARALLEL DO
print *, "result is ", A
end
```

Combined Worksharing Directives

- For convenience combinations of PARALLEL with DO/for, SECTIONS and WORKSHARE are allowed, with reasonable combinations of allowed clauses
- NOWAIT does not make sense in this case
- !\$OMP PARALLEL DO
- #pragma omp parallel for
- !\$OMP PARALLEL SECTIONS
- #pragma omp parallel sections
- !\$OMP PARALLEL WORKSHARE

Allowed Combinations

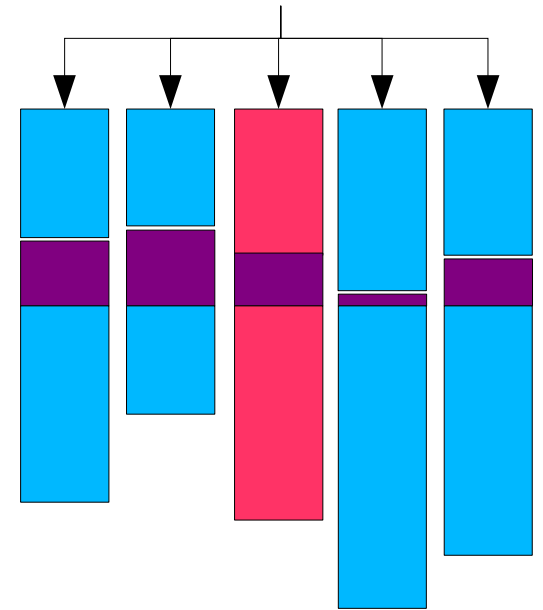
Clause	PARALLEL	DO/fo r	SECTIONS	SINGLE	WORKSHAR E	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE
IF	OK					OK	OK	OK
PRIVATE	OK	OK	OK	OK		OK	OK	OK
SHARED	OK	OK				OK	OK	OK
DEFAULT	OK					OK	OK	OK
FIRSTPRIVATE	OK	OK	OK	OK		OK	OK	OK
LASTPRIVATE		OK	OK			OK	OK	
REDUCTION	OK	OK	OK			OK	OK	OK
COPYIN	OK					OK	OK	OK
SCHEDULE		OK				OK		
ORDERED		OK				OK		
NOWAIT		OK	OK	OK	OK			

Synchronization

- Explicit synchronization is sometimes necessary in OpenMP programs. There's several constructs & directives handling it:
 - **CRITICAL: Mutual Exclusion**
 - `!$OMP CRITICAL [name]/!$OMP END CRITICAL [name]`
 - `#pragma omp critical [name]`
 - **ATOMIC: Atomic Update**
 - `!$OMP ATOMIC, #pragma omp atomic`
 - **BARRIER: Barrier Synchronization**
 - `!$OMP BARRIER, #pragma omp barrier`
 - **MASTER: Master Section**
 - `!$OMP MASTER/!$OMP END MASTER`
 - `#pragma omp master`

BARRIER

- Barrier Synchronization
- Threads wait until all threads reach this point
- Implicit barrier at the end of each parallel region
- Costly operation, to be used judiciously
- Be careful not to cause deadlock:
 - No barrier inside of CRITICAL, MASTER, SECTIONS, SINGLE!



```
C$OMP PARALLEL          #pragma omp parallel
  ..                    {
C$OMP BARRIER          ..
  ..                    #pragma omp barrier
C$OMP END PARALLEL     ..
                        }
```

CRITICAL

- Can be named (strongly suggested)
 - these names have a global scope and must not conflict with subroutine or common block names
- They ensure that only one thread at a time is in the critical section. The rest wait at the beginning of the section, or proceed after finishing their work.
- Essentially serializes work - beware of the **drop in performance** and *use when necessary!*

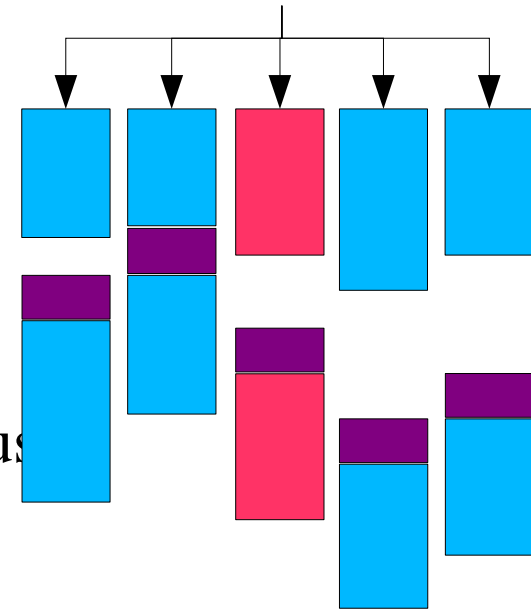
```
C$OMP PARALLEL                                #pragma omp parallel
  ..                                           {
C$OMP CRITICAL(mult)                          ..
  A(i) = A(i) * vlocal                        #pragma omp critical(mult)
C$OMP END CRITICAL(mult)                      A(i) *= vlocal
  ..                                           ..
C$OMP END PARALLEL                            }
```

ATOMIC

- Optimization of mutual exclusion for atomic updates
 - *Not* structured, applies to immediately following statement
 - At the heart of reduction operations (look at `density.f90`)

```
!$OMP ATOMIC
GRID_MASS(BIN(I)) = GRID_MASS(BIN(I)) + PARTICLE_MASS(I)
!$OMP ATOMIC
GRID_N(BIN(I)) = GRID_N(BIN(I)) + 1
```

- Enables fast implementation on some HW
 - In all other cases critical sections are actually used



```
C$OMP PARALLEL                                #pragma omp parallel
  ..                                          {
C$OMP ATOMIC                                  ..
  A(i) = A(i) - vlocal                        #pragma omp atomic
  ..                                          A(i) -= vlocal
C$OMP END PARALLEL                            ..
}
```

Synchronization Examples

- A case combining MASTER, CRITICAL and BARRIER. (look at barrier.f90)

```
!$OMP PARALLEL SHARED(L) PRIVATE(nthreads,tnumber)

nthreads = OMP_GET_NUM_THREADS()

tnumber  = OMP_GET_THREAD_NUM()

!$OMP MASTER

PRINT *, ' Enter a value for L'

READ(5,*)  L

!$OMP END MASTER

!$OMP BARRIER

!$OMP CRITICAL

PRINT *, ' My thread number          = ',tnumber

PRINT *, ' Value of L                = ',L

!$OMP END CRITICAL

!$OMP END PARALLEL
```

use of CRITICAL

- Look at the following piece of code.
- The correct result should be 10+ number of threads.
- Without a critical directive it has a problem in that updates to L are not controlled and a race condition develops.
- If you try to fix things by using an atomic directive to control the actual update to L inside the function, the reading of the inout argument is still not controlled.
- To correct the problem place the subroutine call inside a critical section.
- Look at **critical.f90**, **critical-atomic.f90** and **critical-fixed.f90**

Critical trouble

```
PROGRAM CRITICAL
```

```
    INTEGER:: L,I
```

```
    INTEGER:: nthreads, OMP_GET_NUM_THREADS
```

```
    L=10
```

```
!$OMP PARALLEL SHARED(L) PRIVATE(nthreads,I)
```

```
!$OMP MASTER
```

```
nthreads = OMP_GET_NUM_THREADS()
```

```
    PRINT *, "Number of threads:",nthreads
```

```
!$OMP END MASTER
```

```
!$OMP CRITICAL(adder)
```

```
CALL ADD_ONE(L)
```

```
!$OMP END CRITICAL(adder)
```

```
!$OMP END PARALLEL
```

```
PRINT *, "The final value of L is", L
```

```
END PROGRAM CRITICAL
```

```
SUBROUTINE ADD_ONE(I)
```

```
    IMPLICIT NONE
```

```
    INTEGER, INTENT(INOUT):: I
```

```
    INTEGER:: J
```

```
    J = I
```

```
!$OMP MASTER
```

```
OPEN(UNIT=26,FORM='FORMATTED',FILE='junk')
```

```
    DO I=1,40000
```

```
        WRITE(26,*) "Hi Mom!"
```

```
    END DO
```

```
    CLOSE(26)
```

```
!$OMP END MASTER
```

```
J = J + 1
```

```
I = J
```

```
END SUBROUTINE ADD_ONE
```

OpenMP Runtime

- Library calls:
 - `omp_set_num_threads`, `omp_get_num_threads`
 - `omp_get_max_threads`, `omp_get_num_procs`
 - `omp_get_thread_num`
 - `omp_set_dynamic`, `omp_get_dynamic`
 - `omp_set_nested`, `omp_get_nested`
 - `omp_in_parallel`
- Environment variables:
 - `OMP_NUM_THREADS` (see `NUM_THREADS` clause)
 - `OMP_DYNAMIC`
 - `OMP_NESTED`

Example Programs

- Solution of Helmholtz eqn via Jacobi iterations (look at file `jacobi.f`)
 - Four parallel loops:
 - One for initialization
 - One for copying the previous state
 - One for evaluating the pointwise residual, updating the solution and computing the RMS residual
 - One for evaluating the error
- Molecular dynamics calculation (look at file `md.f`)
 - Two parallel loops:
 - Force and energy calculations for every particle pair
 - Two subroutine/function calls in the lexical extent
 - Time update of positions, velocities and accelerations

Jacobi initialization

```
!$omp parallel do private(xx,yy)
  do j = 1,m
    do i = 1,n
      xx = -1.0 + dx * dble(i-1)      ! -1 < x < 1
      yy = -1.0 + dy * dble(j-1)      ! -1 < y < 1
      u(i,j) = 0.0
      f(i,j) = -alpha *(1.0-xx*xx)*(1.0-yy*yy)
&      - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy)
    enddo
  enddo
!$omp end parallel do
```

Jacobi main loop

```
!$omp do private(resid) reduction(+:error)
  do j = 2,m-1
    do i = 2,n-1
*   Evaluate residual
      resid = (ax*(uold(i-1,j) + uold(i+1,j))
&         + ay*(uold(i,j-1) + uold(i,j+1))
&         + b * uold(i,j) - f(i,j))/b
*   Update solution
      u(i,j) = uold(i,j) - omega * resid
*   Accumulate residual error
      error = error + resid*resid
    end do
  enddo
!$omp enddo nowait
```

Jacobi error

```
!$omp parallel do private(xx,yy,temp) reduction(+:error)
  do j = 1,m
    do i = 1,n
      xx = -1.0d0 + dx * dble(i-1)
      yy = -1.0d0 + dy * dble(j-1)
      temp = u(i,j) - (1.0-xx*xx)*(1.0-yy*yy)
      error = error + temp*temp
    enddo
  enddo
!$omp end parallel do
  error = sqrt(error)/dble(n*m)
```

MD force and energy calculation

```
!$omp parallel do
!$omp& default(shared)
!$omp& private(i,j,k,rij,d)
!$omp& reduction(+ : pot, kin)
    do i=1,np
! compute potential energy
! and forces
        f(1:nd,i) = 0.0
        do j=1,np
            if (i .ne. j) then
                call dist(nd,box, &
                    & pos(1,i),pos(1,j),rij,d)
! attribute half of the potential
! energy to particle 'j'
                pot = pot + 0.5*v(d)
                do k=1,nd
                    f(k,i) = f(k,i) - &
                        & rij(k)*dv(d)/d
                enddo
            endif
        enddo
    enddo
! compute kinetic energy
    kin = kin + &
        & dotr8(nd,vel(1,i),vel(1,i))
enddo
!$omp end parallel do
    kin = kin*0.5*mass
```

MD pos/vel/acc integration

```
!$omp parallel do
!$omp& default(shared)
!$omp& private(i,j)
  do i = 1,np
    do j = 1,nd
      pos(j,i) = pos(j,i) + vel(j,i)*dt + 0.5*dt*dt*a(j,i)
      vel(j,i) = vel(j,i) + 0.5*dt*(f(j,i)*rmass + a(j,i))
      a(j,i) = f(j,i)*rmass
    enddo
  enddo
!$omp end parallel do
```

Parallelizing an existing application

- The really nice feature of OpenMP is that it allows for incremental parallelization of a code. You can start from the most expensive (time-wise) routine and work your way down to less important subroutines with a valid parallel program at any stage in the process.
- A few basic steps when starting from scratch:
 - 1) Get a baseline result (some form of output) that you consider “correct” from your default set of compiler optimization flags.
 - 2) Test for better performance using more aggressive compiler flags testing for correctness to within some tolerance. Set a new baseline result based on your final set of flags.

Preparing for parallelization

- 3) Profile your code: use the `-p/-pg` flags and `prof/gprof` or some of the more capable tools such as Sun Studio's Performance Analyzer. Locate the most costly parts of your code.
- 4) At this point you can opt for either a potentially better performing and scaling “top-level” whole program parallelization that tries to address more coarse grained parallel work or go for the easier solution of the incremental loop level parallelization.
- 5) If you choose the former you need to study the algorithm and try and understand where it offers opportunities for data parallelism.
- 6) For the latter do the same analysis at the “hottest” loop level.

Parallelizing

- 8) One can try and use the diagnostic information that the autoparallelizing engines of most compilers emit to aid in faster identifying the low hanging fruit for the fine level loop parallelization.
- 9) Once you have a functional parallel program test first for correctness vs. the baseline solution and then for speed and scaling at various processor counts. Debug if necessary.
- 10) You then proceed to optimize the code by hoisting and fusing parallel regions so as to get as few as possible, ideally only one for the whole code through the use of **SINGLE & MASTER**. Move code to subroutines with orphaned directives and privatize as many variables as possible. Iterate with (9).

THREADPRIVATE & friends

- \$OMP THREADPRIVATE(list)
- #pragma omp threadprivate(list)
 - Makes global data private to a thread
 - Fortran: COMMON blocks, **module or save variables**
 - C: File scope and static variables
- Different from making them PRIVATE
 - with PRIVATE global scope is lost
 - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN
- **After a SINGLE construct the value of a threadprivate or private variable can be broadcast to all threads using the COPYPRIVATE clause.**

THREADPRIVATE example

```
integer iam, nthr
real rnumber
common /identify/ iam, nthr
common /work/ rnumber
!$OMP THREADPRIVATE(/identify/ &
!$OMP& , /work/)
!$OMP PARALLEL
!$ iam = omp_get_thread_num()
!$ nthr = omp_get_num_threads()
!$OMP END PARALLEL
CALL DO_SERIAL_WORK
!OMP PARALLEL COPYIN(work)
print *, iam, nthr, rnumber
call DO_PARALLEL_WORK
!OMP END PARALLEL
```

```
subroutine DO_PARALLEL_WORK
integer i, iam, nthr
real rnumber
common /identify/ iam, nthr
common /work/ rnumber
!$OMP THREADPRIVATE(/identify/ &
!$OMP& , /work/)
!$OMP DO
DO i=1,nthr
print *, rnumber, iam
ENDDO
!$OMP END DO
end
```

Look at threadprivate.f90

THREADPRIVATE flow

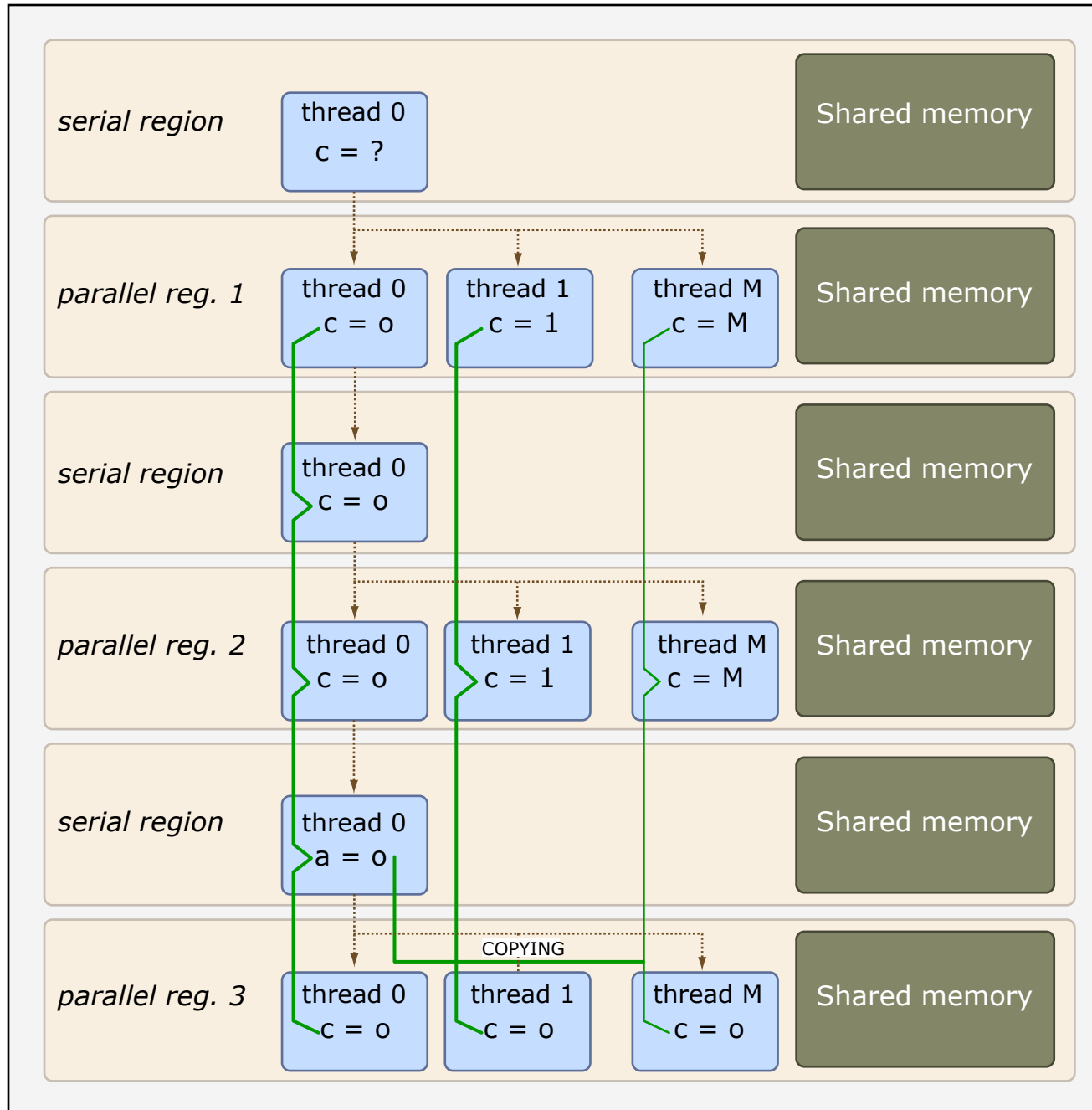


Figure by MIT OpenCourseWare.

SCHEDULE

- The user has finer control over the distribution of loop iterations onto threads in the DO/for construct:
 - SCHEDULE(static[,chunk])
 - Distribute work evenly or in *chunk* size units
 - SCHEDULE(dynamic[,chunk])
 - Distribute work on available threads in *chunk* sizes.
 - SCHEDULE(guided[,chunk])
 - Variation of *dynamic* starting from large chunks and exponentially going down to *chunk* size.
 - SCHEDULE(runtime)
 - The environment variable OMP_SCHEDULE which is one of *static*, *dynamic*, *guided* or an appropriate pair, say "static,500"

STATIC

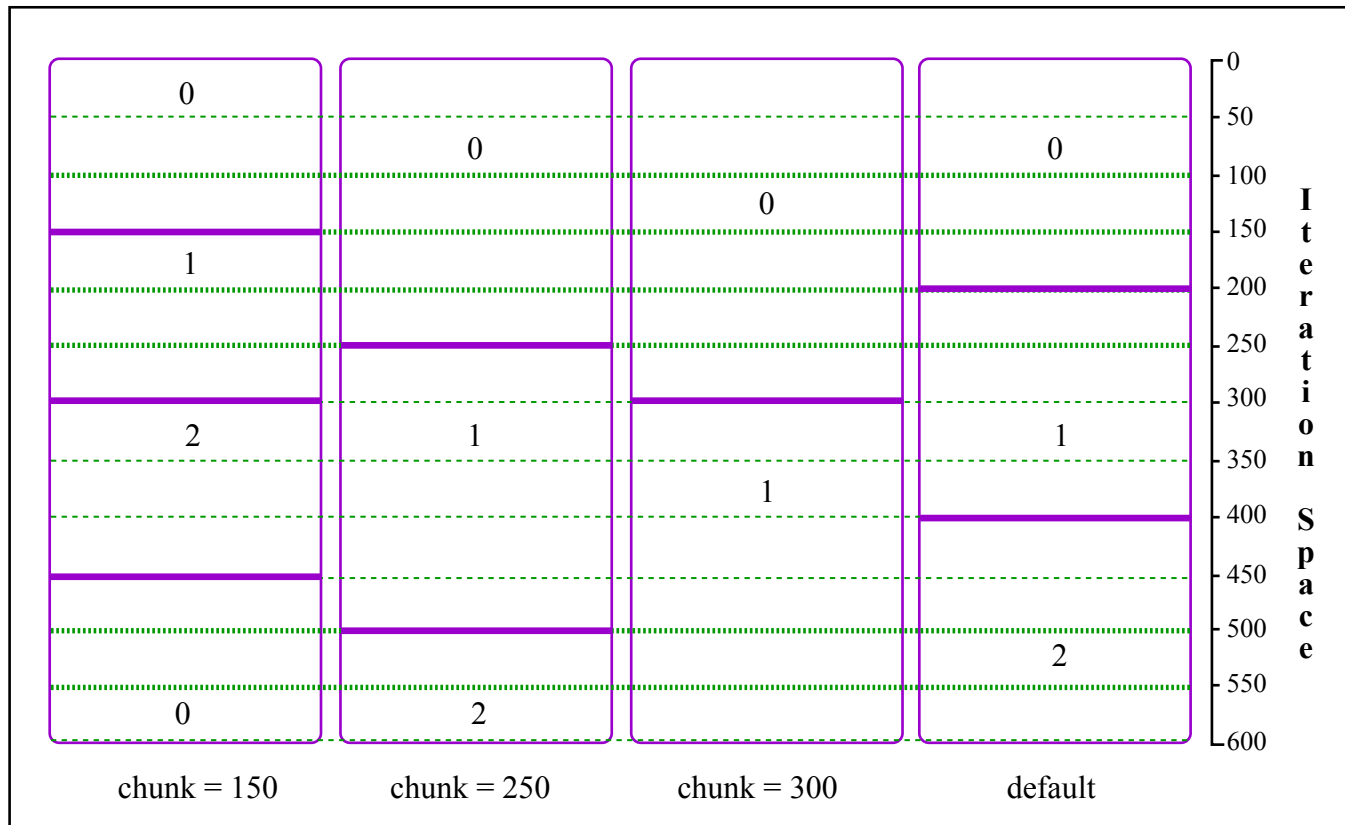


Figure by MIT OpenCourseWare.

DYNAMIC

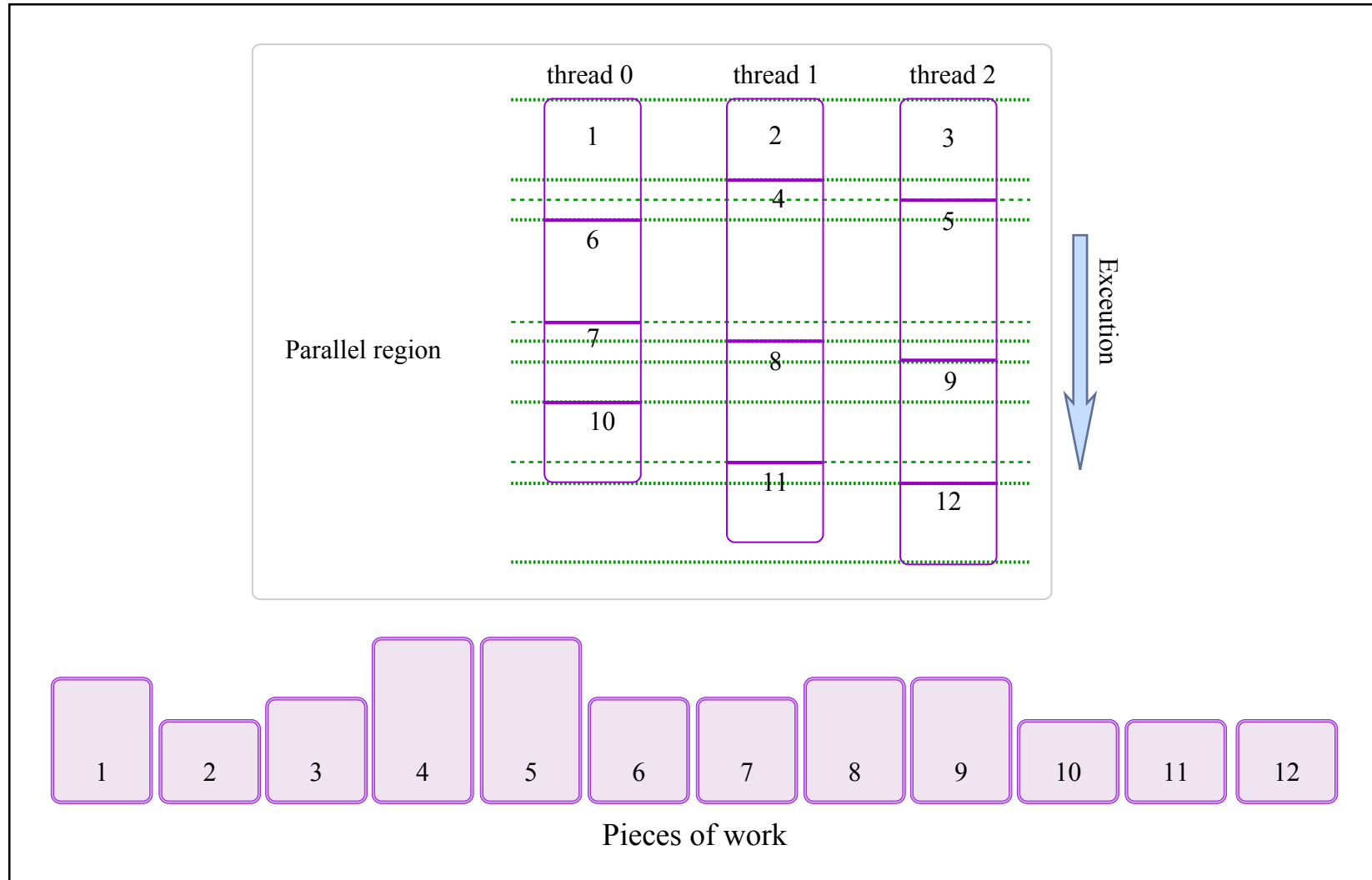


Figure by MIT OpenCourseWare.

GUIDED

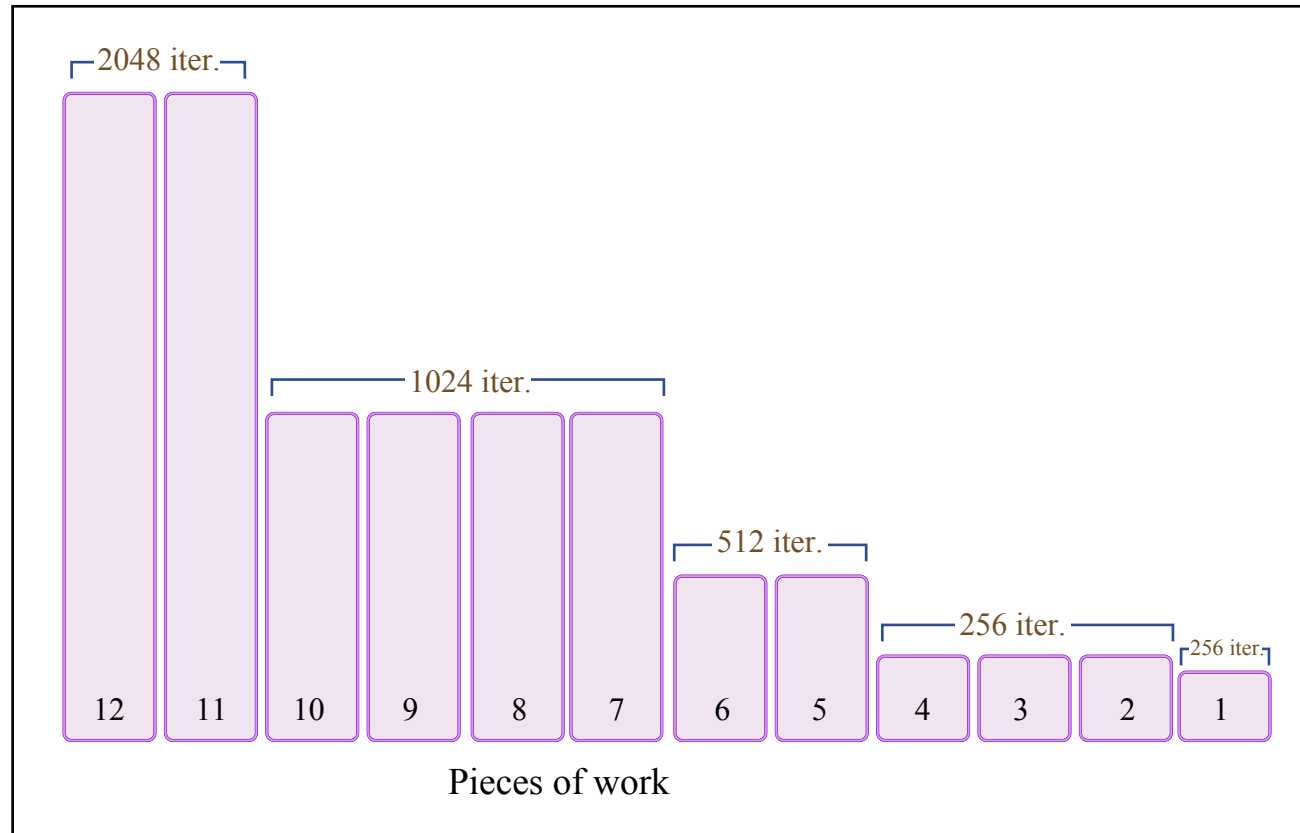


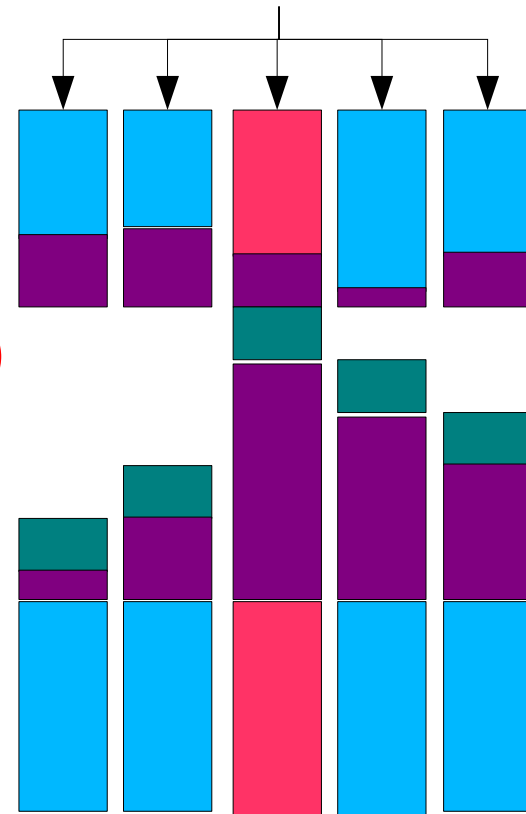
Figure by MIT OpenCourseWare.

ORDERED clause

- Enforces sequential order in a part of a parallel loop:
- Requires the ordered clause to the DO/for construct
- No more than one ordered directive can be executed per iteration

```
!$OMP PARALLEL DEFAULT(SHARED) &  
!$OMP& PRIVATE(I,J)  
!$OMP DO SCHEDULE(DYNAMIC,4) &  
!$OMP& ORDERED  
DO I=1,N  
DO J=1,M  
Z(I) = Z(I) + X(I,J)*Y(J,I)  
END DO  
!$OMP ORDERED  
IF(I<21) THEN  
PRINT *, 'Z(',I,') = ',Z(I)  
END IF  
!$OMP END ORDERED  
END DO  
!$OMP END DO  
!$OMP END PARALLEL
```

Look at ordered.f90



FLUSH directive

- At the heart of BARRIER, it enforces a consistent view of memory.
- !\$OMP FLUSH [(list)]
- #pragma omp flush [(list)]
- Restrictions on the use of flush in C/C++, same restrictions applying to barrier.

```
if (x != 0)
    #pragma omp flush
```

```
if (x != 0)
{
    #pragma omp flush
}
```

lock API

- `omp_init_lock`, `omp_destroy_lock`
- `omp_set_lock`, `omp_unset_lock`, `omp_test_lock`
- `omp_init_nest_lock`, `omp_destroy_nest_lock`
- `omp_set_nest_lock`, `omp_unset_nest_lock`,
`omp_test_nest_lock`

timing API

- `omp_get_wtime`
- `omp_get_wtick`

OpenMP Performance issues

Even when running using only one thread performance can be lower than the scalar code.

Several performance problems to consider

Parallelization

- thread management costs
- startup costs
 - creation & destruction
- small loop overhead
- additional code costs

Runtime

- load imbalance
- synchronization costs
 - excessive barriers
- false sharing
- processor affinity

OpenMP bugs

Race conditions

- Different results at different times due to:
 - unprivitized variables
 - unprotected updates
 - unfinished updates

```
real a, tmp
a = 0.0
C$OMP PARALLEL
C$OMP DO REDUCTION(+:a)
do i=1, n
    tmp = sin(0.1*i)
    a = a + tmp
    b(i) = tmp
enddo
C$OMP END DO NOWAIT
print *, b(1)
C$OMP END PARALLEL DO
print *, a
```

Deadlocks

- Not all threads enter a barrier
- A thread never releases a lock
- Two threads get the same nested locks in different succession

Consistency

- private variables mask global variables

Sequential Equivalence

- Using a subset of OpenMP produce parallel code that gives the same results with the serial code.
 - Only temporary and loop variables are private
 - Updates of shared variables are protected
- Strong Sequential Equivalence: bitwise identical results
 - Sequential ordered updates of variables
 - Serialization of reduction operations (ordered loops)
- Weak Sequential Equivalence: equivalent to within floating point math nuances
 - Sequential (critical section) updates

OpenMP on Linux platforms

- Sun Studio compilers: suncc/CC and sunf77/f90/f95
 - -xopenmp, -xopenmp=noopt
 - Autoscopying extensions! DEFAULT(__AUTO)
 - Free (without support) use
- Intel compilers: icc/icpc and ifort
 - **-openmp** -openmp_report[0,1,2]
 - Try KMP_SCHEDULE="static,balanced"
 - Can be used with a free personal license.
- GNU C/C++ and Fortran version 4.3 (and distro backports)
 - -fopenmp

OpenMP on Linux

IA32/IA64/AMD64 (commercial)

- Compilers
 - Portland Group compilers (PGI) with debugger
 - Absoft Fortran compiler
 - Pathscale compilers
 - Lahey/Fujitsu Fortran compiler
- Debuggers
 - Totalview
 - Allinea DDT

Other compilers and tools

- Research compilers
 - Omni
 - OdinMP/OdinMP2
 - OMPI
 - OpenUH
 - Intone
 - Nanos Mercurium
- Tools
 - Intel Thread Checker (free for personal use)
 - Sun Studio Thread Analyzer (free)
 - Sun Studio Performance Analyzer (free)

Cluster/DM Extensions

- OpenMP extensions to distributed memory machines using software shared memory (usually page-based coherence) or some other mechanism
 - Intel Cluster OpenMP (commercial enhancement of Rice's Treadmarks SDSM package)
 - Omni/SCASH combination
 - Good for a limited set of problems that exhibit very good spatial locality (so that fetching a memory page of 4KB does not result in a lot of wasted traffic). Otherwise the scalability is very limited – only memory expansion.
 - Lots of research alternatives

Summary

- OpenMP provides a portable high performance path towards parallelization - allowing for incremental parallelization and both coarse and fine grained parallelism targeting moderate numbers of processors.
- Using worksharing constructs one can describe parallelism in a very compact manner.
- Care needs to be taken with the data scope attributes to avoid bugs as well as performance issues
- Programming in a coarse-grained, SPMD format is the key to high OpenMP performance.
- Ideal for automatic load balancing

Further information

- The OpenMP ARB <http://www.openmp.org>
- The OpenMP users group <http://www.compunity.org>
- The Sun compilers
<http://developers.sun.com/sunstudio>
- OpenMP and the GNU compilers
<http://gcc.gnu.org/projects/gomp>
- The Intel compilers
<http://www.intel.com/software/products/compilers>
- A lot more information on the class Stellar website

MIT OpenCourseWare
<http://ocw.mit.edu>

12.950 Parallel Programming for Multicore Machines Using OpenMP and MPI
IAP 2010

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.