Shared Memory Programming

Ilhan Guner

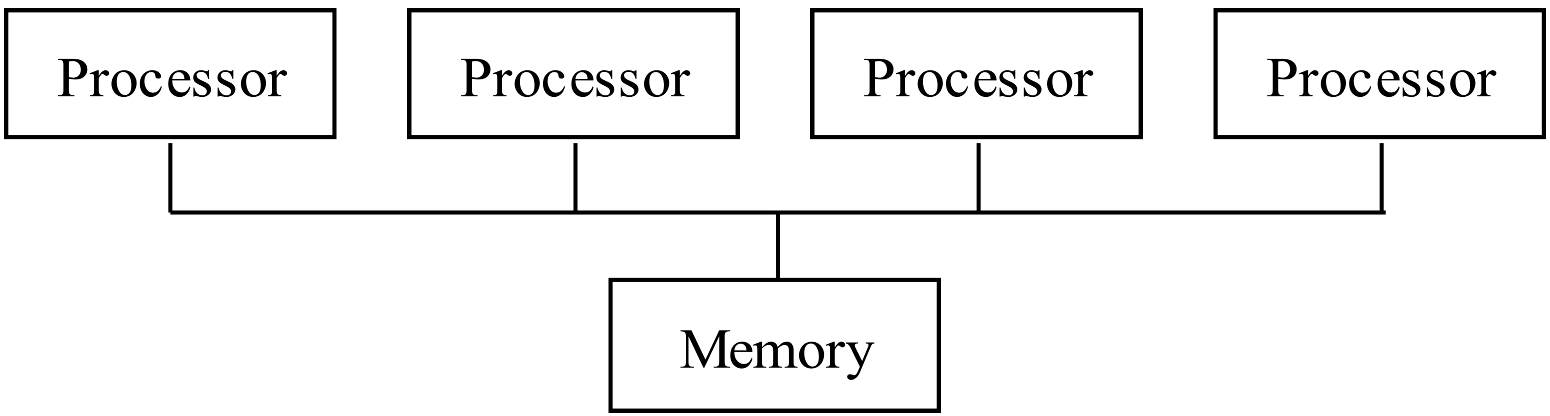
Disclosure

- Slides are adapted from Aaron Bloomfield's lecture slides on HPC Bootcamp

Parallel Computing

- Running a code in different computing units at the same time
- Computing units share memory
  - OpenMP
- Computing units do NOT share memory
  - MPI

Shared-memory Model



Processors interact and synchronize with each other through shared variables.

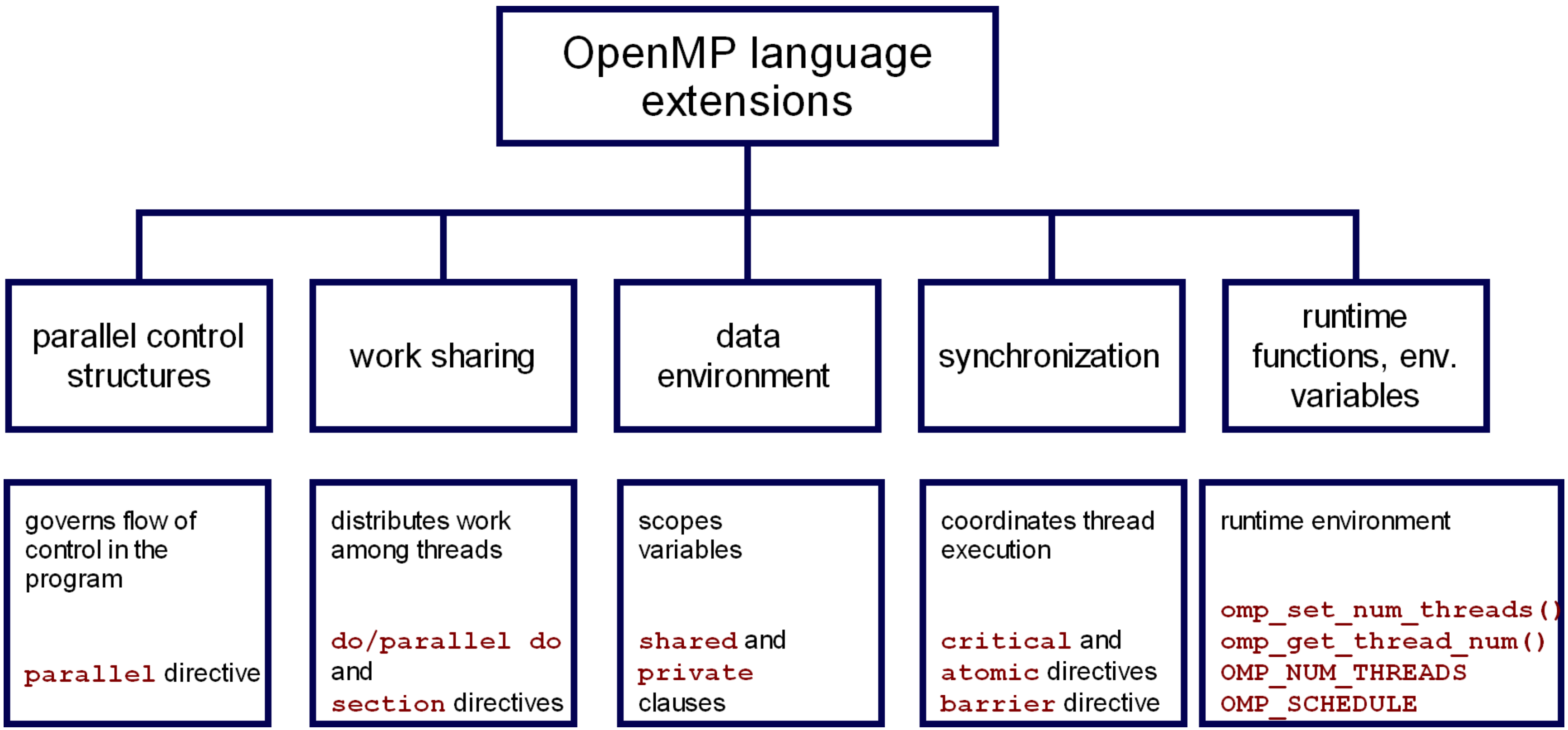
MPI vs OpenMP

Characteristic	*OpenMP*	*MPI*
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes

OpenMP

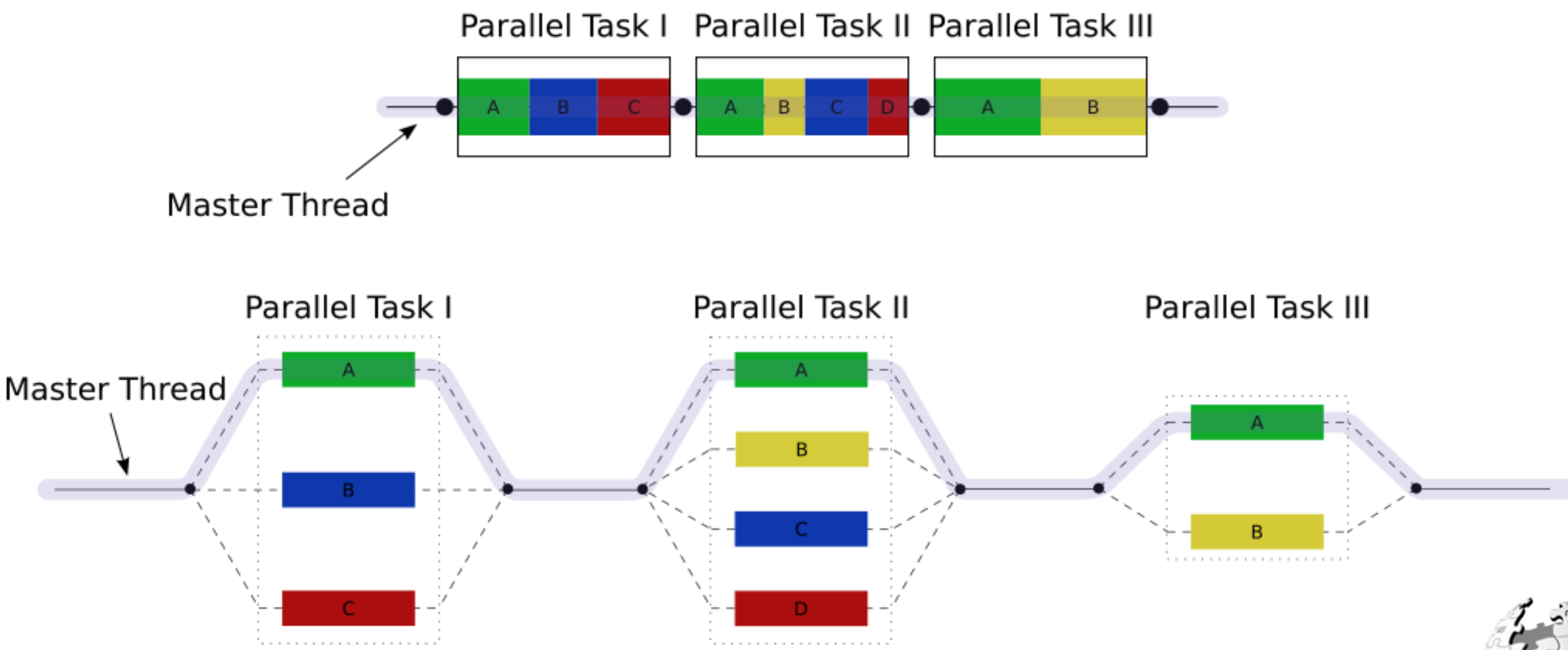
- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
- OpenMP works in conjunction with Fortran, C, or C++

OpenMP



OpenMP language extensions
parallel control structures
work sharing
data environment
synchronization
runtime functions, env. variables
governs flow of control in the program
parallel directive
distributes work among threads
do/parallel do and section directives
scopes variables
shared and private clauses
coordinates thread execution
critical and atomic directives
barrier directive
runtime environment
omp_set_num_threads()
omp_get_thread_num()
OMP_NUM_THREADS
OMP_SCHEDULE

How OpenMP works



Fork/Join Parallelism

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

Implementation

- Parallelism in do loops
  - parallel do
  - do
  - parallel
- Parallelism in sections of codes

Parallel do

- Format:

```
!$omp parallel do
 do i = 1,N
    a(i) = b(i) + c(i)
 enddo
!$omp end parallel do
```

- the loop iterations are completely independent

Restrictions

- do loops must have canonical form
- Cannot have statements that end the loop prematurely
  - break, returns, exits, or gotos
  - But can have continues

Shared and Private Variables

- Shared variable: has same address in execution context of every thread
  - All variables are shared by default
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```

Example Use of private Clause

```
!$omp parallel do private(j)
  do i = 1,N
     do j = 1,M
      a(i,j) = 2.0
     enddo
  enddo
!$omp end parallel do
```

Race Condition

```
a = 0.0
!$omp parallel do
 do i = 1,N
   a = a + 4.0
 enddo
!$omp end parallel do
```

May cause problems

critical Pragma

- Critical section: a portion of code that only thread at a time may execute
- We denote a critical section by putting the pragma

```
!$omp critical
```

in front of a block of Fortran code

Correct, But Inefficient, Code

```
a = 0.0
!$omp parallel do
 do i = 1,N
   !$omp critical
   a = a + 4.0
   !$omp end critical
 enddo
!$omp end parallel do
```

Source of Inefficiency

- Update to **`a`** inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop

reduction Clause

- The reduction clause has this syntax:
  `reduction (<op> :<variable>)`

- Operators

+	Sum
*	Product
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
&&	Logical and
||	Logical or
max	
min	

```
a = 0.0;
!$omp parallel do reduction(+:a)
 do i = 1,N
   a = a + 4.0
 enddo
!$omp end parallel do
```

Performance Improvement

- Too many fork/joins can lower performance
- Parallelism in the outer loop
- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- **`!$omp parallel for if(n > 5000)`**
- **`schedule`** clause to allocate iterations to threads

schedule Clause

- Syntax of schedule clause:

  `schedule (<type>[,<chunk> ])`
- Schedule type required, chunk size optional
- Allowable schedule types
  - static: static allocation
  - dynamic: dynamic allocation
  - guided: guided self-scheduling
  - runtime: type chosen at run-time based on value of environment variable OMP_SCHEDULE

parallel Pragma

- The **`parallel`** pragma precedes a block of code that should be executed by *all* of the threads
- Note: execution is replicated among all threads

do Pragma

- The **`parallel do`** pragma will fork the threads, and split the for loop into parts
- The **`parallel`** pragma will fork the threads, and execute the *same* for loop for each thread (i.e. not split any loops into parts)
- But if you have already split the threads (via a **`parallel`** pragma), and want to split a for loop among the *already existing* threads (as opposed to executing the entire loop in all threads), then use you a for pragma:

```
!$omp do
```

Example Use of do Pragma

```
!$omp parallel private(i,j)
do i = 1,m
   low = a(i);
   high = b(i);
   if (low > high) then
      print*,"Exiting"
      exit
   end if
enddo
!$omp do
   do j = low, high
      c(j) = (c(j) - a(i))/b(i)
   end do
!$omp end do
!$omp end parallel
```

single Pragma

- Suppose we only want to see the output once
- The **`single`** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```
!$omp single
```

```
!$omp parallel private(i,j)
do i = 1,m
   low = a(i);
   high = b(i);
   if (low > high) then
      !$omp single
      print*,"Exiting"
      !$omp end single
      exit
   end if
enddo
!$omp do
   do j = low, high
      c(j) = (c(j) - a(i))/b(i)
   end do
!$omp end do
!$omp end parallel
```

nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- In our example, this is necessary for the i for loop
  - If a thread leaves loop and changes **`low`** or **`high`**, it may affect behavior of another thread
  - But it's not necessary for the j for loop
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time

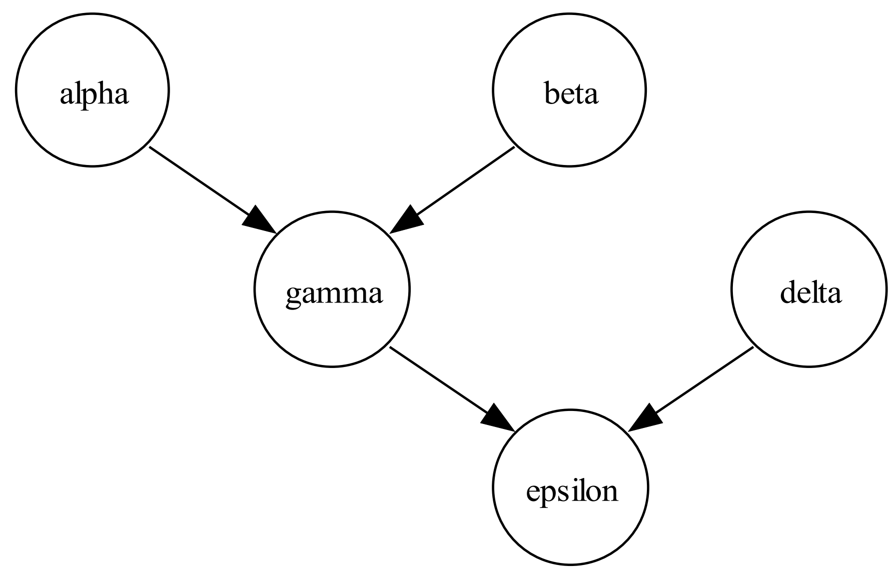
Functional Parallelism

- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)

Functional Parallelism Example

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

May execute alpha, beta, and delta in parallel



parallel sections Pragma

- Precedes a block of *k* blocks of code that may be executed concurrently by *k* threads
- Syntax:

```
!$omp parallel sections
```

section Pragma

- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```
!$omp section
```

Example of **`parallel sections`**

```
!$omp parallel sections
  !$omp section  /* Optional */
      v = alpha()
  !$omp section
      w = beta()
  !$omp section
      y = delta()
!$omp end parallel sections
      x = gamma(v, w)
   print*, epsilon(x,y)
```

All 10 OpenMP functions

- omp_get_dynamic()
- omp_get_max_threads()
- omp_get_nested()
- omp_get_num_procs()
- omp_get_num_threads()
- omp_get_thread_num()
- omp_in_parallel()
- omp_set_dynamic()
- omp_set_nested()
- omp_set_num_threads()

Problems Associated with Shared Data

- Cache coherence
  - Replicating data across multiple caches reduces contention
  - How to ensure different processors have same value for same address?
- Synchronization
  - Mutual exclusion
  - Barrier

Summary (1/3)

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause

Summary (2/3)

- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling

Resources

- http://www.openmp.org
  - Sample codes in Fortran
  - Specifications
- https://computing.llnl.gov/tutorials/openMP/

Compiling OpenMP programs

- Use Intel's compilers on the cluster
  - module load intel
  - ifort –openmp –o <executable> <f90 files>
  - Supports other compiling options as well
- Use gfortran on your PC
  - gfortran –fopenmp –o <executable> <f90 files>
- Write the program with the pragmas

Hello World in Fortran with OpenMP

```
program hello90
    use omp_lib
    integer:: id, nthreads
        !$omp parallel private(id)
        id = omp_get_thread_num()
        write (*,*) 'Hello World from thread', id
        !$omp barrier
        if ( id == 0 ) then
                nthreads = omp_get_num_threads()
                write (*,*) 'There are', nthreads, 'threads‘
        end if
    !$omp end parallel
end program
```

WIKIPEDIA
The Free Encyclopedia