The Supercomputing People

OpenMP

(Open Specifications for multi processing)

Deepika H.V

SSDG,

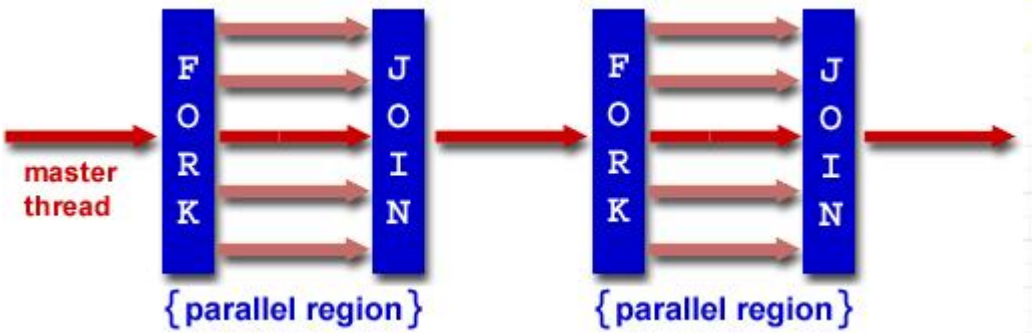
C-DAC,Bangalore

What is OpenMP?

- **An Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory parallelism***
- **Comprised of three primary API components**:
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- **Portable:**
  - The API is specified for C/C++ and Fortran
  - Multiple platforms have been implemented including most Unix platforms and Windows NT
- **Standardized:**
  - Jointly defined and endorsed by a group of major computer hardware and software vendors

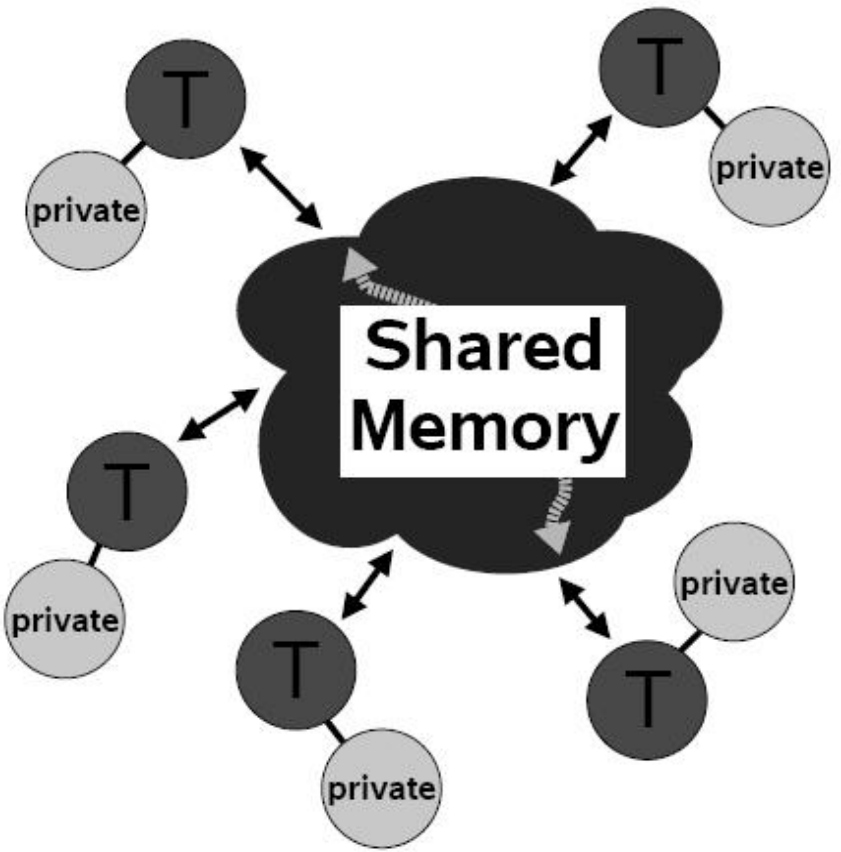
OpenMP Programming Model

- Shared Memory, Thread Based Parallelism
- Explicit Parallelism
- Fork - Join Model



- Compiler Directive Based
- Nested Parallelism Support
- Dynamic Threads
- Memory Model: Flush often

OpenMP ?



✓All threads have access to same globally shared memory.

✓ Data is private or shared.

✓Shared data accessible by all threads.

✓Private accessed only by owned threads.

✓Data transfer is transparent to programmer.

✓Synchronization takes place, but it is almost implicit.

Components of OpenMP

Directives

- ✓Parallel regions
- ✓Work Sharing
- ✓Synchronization
- ✓Data scoping
  - private
  - first private
  - last private
  - shared
  - reduction
- ✓Orphaning

Environment Variables

- ✓Number of threads
- ✓Scheduling type
- ✓Dynamic thread adjustemnt
- ✓Nested parallelism

Runtime Library routines

- ✓Number of threads
- ✓Thread ID
- ✓Dynamic thread adjustment
- ✓Nested parallelism
- ✓Timers
- ✓Lock

OpenMP Directives

#pragma omp *directive-name [clause[[,] clause]...] new-line*

Eg: #pragma omp parallel default(shared) private(beta,pi)

General Rules:

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- 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" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

OpenMP Directives

1. Data Scoping Attribute Clauses

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped. They include:

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION
- COPYIN
- Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables

[A] private clause

This declares variables in its list to be private to each thread

Format

private (list)

Eg: int B = 10;
#pragma omp parallel private(B)
B = … ;

- ✓ A private uninitialised copy of B is created before the parallel region begins
- ✓ B is not the same within the parallel region as outside

[B] firstprivate clause

Format

first private (list)

Eg: int B;

```
B = 10;
#pragma omp parallel firstprivate(B)
B = B + … ;
```

- ✓ A private initialised copy of B is created before the parallel region begins
- ✓ The copy of each thread gets the same value

[C] SHARED Clause

A shared variable exists in only one memory location and all threads can read or write to that address

Format

shared *(list)*

[D] DEFAULT Clause

Specify defualt scope for all variables in the lexical extent.

Format

default (shared | none)

[E] LASTPRIVATE Clause

Value from the last loop iteration assigned the original variable object.

Format

lastprivate *(list)*

[F] COPYIN Clause

The team threads are initialized with its value from master thread upon entry into the parallel construct. They are used for threadprivate variables

Format

copyin (list)

[H] COPYPRIVATE Clause

- ✓ The COPYPRIVATE clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.
- ✓ Associated with the SINGLE directive

Format

copyprivate *(list)*

[G] REDUCTION Clause

private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Format

reduction (operator: list)

OpenMP Directives

2. PARALLEL Region Construct

- ✓ A parallel region is a block of code that will be executed by multiple threads
- ✓ #pragma omp parallel *[clause ...] newline*

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

structured_block

- ✓ On reaching PARALLEL directive, it creates a team of threads and becomes the master of the team.
- ✓ The master is a member of that team and has thread number 0 within that team.
- ✓ Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- ✓ There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- ✓ If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - ✓ **Evaluation of the IF clause**
  - ✓ **Setting of the NUM_THREADS clause**
  - ✓ **Use of the omp_set_num_threads() library function**
  - ✓ **Setting of the OMP_NUM_THREADS environment variable**
  - ✓ **Implementation default - usually the number of CPUs on a node, though it could be dynamic (see next bullet).**

- Threads are numbered from 0 (master thread) to N-1

Dynamic Threads

- **omp_get_dynamic()** determines if dynamic threads are enabled.
- If supported, the two methods available for enabling dynamic threads are
  - ✓ The **omp_set_dynamic()** library routine
  - ✓ Setting of the **OMP_DYNAMIC** environment variable to TRUE

Nested Parallel Regions

- **omp_get_nested()** library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
  - ✓ **The** omp_set_nested() **library routine**
  - ✓ **Setting of the** OMP_NESTED **environment variable to TRUE**

- If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

Restrictions:

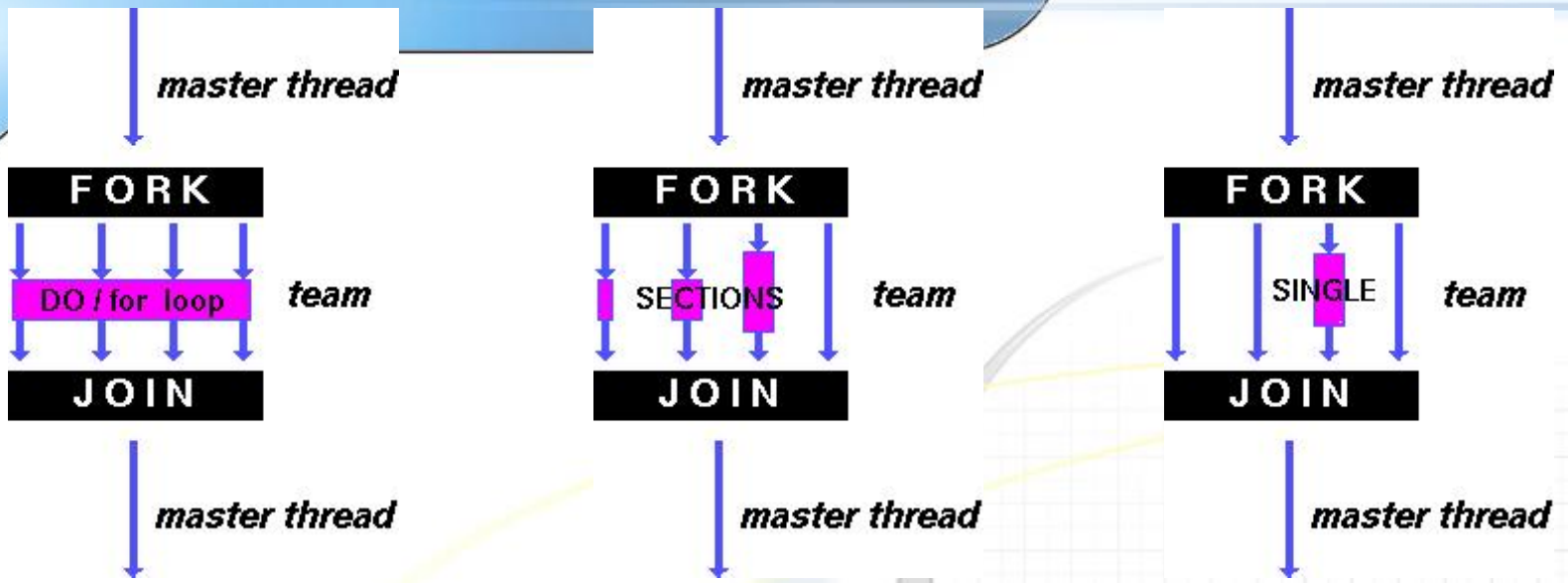
- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted
- Only a single NUM_THREADS clause is permitted

3. Work-Sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads

Types of Work-Sharing Constructs

- ✓ **for** - shares iterations of a loop across the team. Represents a type of "data parallelism".
- ✓ **SECTIONS** - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".
- ✓ **SINGLE** - serializes a section of code



Restrictions

✓A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

✓Work-sharing constructs must be encountered by all members of a team or none at all

✓Successive work-sharing constructs must be encountered in the same order by all members of a team

[A] for Directive

for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team.

- **Format**

#pragma omp for *[clause ...] newline*
schedule *(type [,chunk])*
ordered
private *(list)*
firstprivate *(list)*
lastprivate *(list)*
shared *(list)*
reduction *(operator: list)*
nowait
for_loop

Clauses

- ✓ **SCHEDULE**: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.
  - -- STATIC
  - -- DYNAMIC
  - -- GUIDED
- ✓ **NO WAIT / nowait**: Theads do not synchronize at the end of the parallel loop.
- ✓ **ORDERED**: Specifies that the iterations of the loop must be executed as they would be in a serial program.

[B] SECTION Directive

Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team.

- **Format**

```
#pragma omp sections [clause ...] newline
                        private (list)
                       firstprivate (list)
                       lastprivate (list)
                       reduction (operator: list)
                       nowait
 {
  #pragma omp section newline
          structured_block
  #pragma omp section newline
         structured_block
 }
```

Clauses

- ✓ **NOWAIT:** implied barrier exists at the end of a SECTIONS directive, unless this clause is used.

Restrictions:

- It is illegal to branch into or out of section blocks.
- SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

[C] SINGLE Directive

Purpose

- ✓ The enclosed code is to be executed by only one thread in the team.
- ✓ May be useful when dealing with sections of code that are not thread safe (such as I/O)

Format:

#pragma omp single *[clause ...] newline*

private *(list)*

firstprivate *(list)*

nowait

structured_block

4. Combined Parallel Work-Sharing Constructs

- These directives behave identical to individual parallel directives

Types of Directives

- ✓ **parallel for**
- ✓ **parallel sections**

For ex:

```
#pragma omp parallel sections
{
   #pragma omp section
      xaxis();
   #pragma omp section
      yaxis();
   #pragma omp section
      zaxis();
}
```

5. Synchronization Constructs

- Two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0):

THREAD 1:	**THREAD 2:**
increment(x) {	increment(x) {
x = x + 1;	x = x + 1;
}	}
THREAD 1:	**THREAD 2:**
10 LOAD A, (x address)	10 LOAD B, (x address)
20 ADD A, 1	20 ADD B, 1
30 STORE A, (x address)	30 STORE B, (x address)

[A] MASTER Directive

- To be executed only by the master thread of the team. All other threads on the team skip this section of code
- There is no implied barrier associated with this directive

Format

#pragma omp master *newline*

structured_block

[B] CRITICAL Directive

Purpose

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

Format

- #pragma omp critical *[name] newline structured_block*

✓ If a thread is executing a CRITICAL region and another thread tries to execute CRITICAL region, it will block until the first thread exits that CRITICAL region.

✓ The optional name enables multiple different CRITICAL regions to exist:

-- Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.

-- All CRITICAL sections which are unnamed, are treated as the same section.

[C] BARRIER Directive

When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Format

#pragma omp barrier newline

Eg: Suppose we run each of these two loops in parallel over i:

```
for (i=0; i < N; i++)
    a[i] = b[i] + c[i];

for (i=0; i < M; i++) // M < N
    d[i] = a[i] + b[i];
```

We need to have updated all of a[ ] first, before using a[ ]

```
for (i=0; i < N; i++)
a[i] = b[i] + c[i];
```

```
for (i=0; i < M; i++) // M < N
d[i] = a[i] + b[i];
```

All threads wait at the barrier point and only continue when all threads have reached the barrier point

[E] FLUSH Directive

The FLUSH directive identifies a synchronization point at which the implementation must provide a consistent view of memory

Format

#pragma omp flush (list) newline

-- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables.

-- The FLUSH directive is implied for the directives shown in the table below.

barrier
parallel - upon entry and exit
critical - upon entry and exit
ordered - upon entry and exit
for - upon exit
sections - upon exit
single - upon exit

[F] ORDERED Directive

Purpose

- Loop iterations will be executed in the same order as if they were executed on a serial processor.

Format:

#pragma omp for ordered *[clauses...]*

(loop region)

#pragma omp ordered *newline*

structured_block

(end of loop region)

THREADPRIVATE Directive

THREADPRIVATE variables differ from PRIVATE variables because they are able to persist between different parallel sections of a code.

Format

```
#pragma omp threadprivate (list)
```

Run-Time Library Routines

- **OMP_SET_NUM_THREADS**

  Sets the number of threads that will be used in the next parallel region

```
#include <omp.h>
void omp_set_num_threads(int num_threads)
```

- **OMP_GET_NUM_THREADS**

  Returns the number of threads that are currently in the team executing the parallel region

```
#include <omp.h>
int omp_get_num_threads(void)
```

- **OMP_GET_MAX_THREADS**

  Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function.

```
#include <omp.h>
int omp_get_max_threads(void)
```

Run-Time Library Routines

- **OMP_GET_THREAD_NUM**

  Returns the thread number of the thread

```
#include <omp.h>
int omp_get_thread_num(void)
```

- **OMP_GET_NUM_PROCS**

  Returns the number of processors that are available to the program.

```
#include <omp.h>
int omp_get_num_procs(void)
```

- **OMP_IN_PARALLEL**

  determine if the section of code which is executing is parallel or not.

```
#include <omp.h>
int omp_in_parallel(void)
```

Run-Time Library Routines

- ✓ **OMP_SET_DYNAMIC**

  Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions.

  ```
  #include <omp.h>
  void omp_set_dynamic(int dynamic_threads
  ```

- ✓ **OMP_GET_DYNAMIC**

  Determine if dynamic thread adjustment is enabled or not.

  ```
  #include <omp.h>
  int omp_get_dynamic(void)
  ```

Run-Time Library Routines

- ✓ **OMP_SET_NESTED**

  Used to enable or disable nested parallelism

  ```
  #include <omp.h>
  void omp_set_nested(int nested)
  ```

- ✓ **OMP_GET_NESTED**

  ```
  #include <omp.h>
  int omp_get_nested (void
  ```

Environment Variables

- **OMP_SCHEDULE**

  setenv OMP_SCHEDULE "guided, 4"

  setenv OMP_SCHEDULE "dynamic"

- **OMP_NUM_THREADS**

  setenv OMP_NUM_THREADS 8

- **OMP_DYNAMIC**

  setenv OMP_DYNAMIC TRUE

- **OMP_NESTED**

  Enables or disables nested parallelism.

  setenv OMP_NESTED TRUE

Directive Binding:

- ✓ The DO/for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists. If no parallel region is currently being executed, the directives have no effect.
- ✓ The ORDERED directive binds to the dynamically enclosing DO/for.
- ✓ A directive can never bind to any directive outside the closest enclosing PARALLEL.

Directive Nesting:

- ✓ A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- ✓ DO/for, SECTIONS, and SINGLE directives that bind to the same PARALLEL are not allowed to be nested inside of each other.
- ✓ DO/for, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL, ORDERED and MASTER regions.
- ✓ CRITICAL directives with the same name cannot be nested inside of each other.
- ✓ BARRIER directives are not permitted in the dynamic extent of DO/for, ORDERED, SECTIONS, SINGLE, MASTER and CRITICAL regions.
- ✓ MASTER directives are not permitted in the dynamic extent of DO/for, SECTIONS and SINGLE directives.
- ✓ ORDERED directives are not permitted in the dynamic extent of CRITICAL regions.

Thank You