



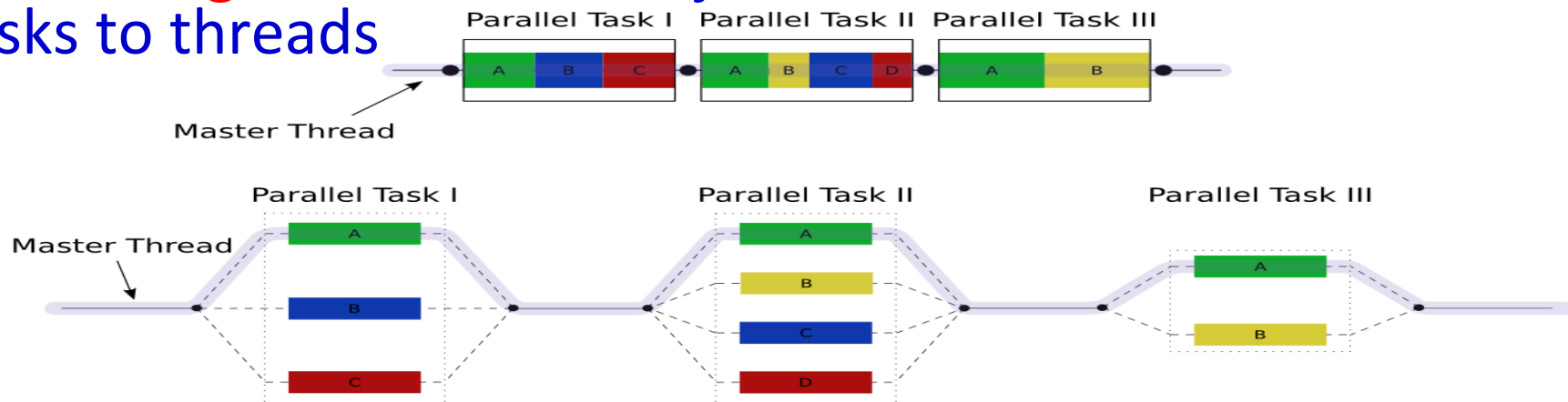
Shared-Memory Programming: OpenMP

National Tsing-Hua University
2019, Summer Semester

What's OpenMP

OpenMP == **Open** specification for **Multi-Processing**

- **An API** : multi-threaded, shared memory parallelism
- **Portable**: the API is specified for C/C++ and Fortran
- **Fork-Join model**: the master thread forks a specified number of slave threads and divides task among them
- **Compiler Directive Based**: **Compiler** takes care of generating code that forks/joins threads and divide tasks to threads



Example

- Add two data arrays in parallel by specifying **compiler directives**:
 - **Slave threads are forked** and each thread works on different iterations

```
#include <omp.h>
// Serial code
int A[10], B[10], C[10];

// Beginning of parallel section. Fork a team of threads.
#pragma omp parallel for num_threads(10)
{
    for (int i=0; i<10; i++)
        A[i] = B[i] + C[i];
} /* All threads join master thread and terminate */
```

OpenMP Directives

■ C/C++ Format:

#pragma omp	directive-name	[clause, ...]	newline
Required.	Valid OpenMP directive: parallel , do , for	Optional . Clauses can be in any order, and repeated as necessary.	Required.

■ Example:

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

↓ ↓ ↓

directive-name clause clause

■ General Rules:

- **Case sensitive**
- **Only one directive-name** may be specified per directive
- Each directive **applies to at most one succeeding statement**, which must be a **structured block**



OpenMP Outline

- Parallel Region Construct
 - Parallel Directive
- Working-Sharing Construct
 - DO/for Directive
 - SECTIONS Directive
 - SINGLE Directive
- Synchronization Construct
- Data Scope Attribute Clauses
- Run-Time Library Routines

Parallel Region Constructs --- Parallel Directive

- A parallel region is a block of code executed by multiple threads

```
#pragma omp parallel [clause .....]  
                if (scalar_expression)  
                num_threads (integer-expression)  
structured_block
```

- Overview:

- When **PARALLEL** is reached, a team of threads is created
- The parallel region code is duplicated and executed by all threads
- There is an implied barrier at the end of a parallel section.
- One thread terminates, all threads terminate

- Limitations:

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch (goto) into or out of a parallel region, but you could call other functions within a parallel region

Parallel Region --- How Many Threads

- The number of threads in a parallel region is determined in order of following precedence:
 - Evaluation of the **IF** clause
 - ◆ If **FALSE**, it is **executed serially** by the master thread
 - ◆ E.g: `#pragma omp parallel IF(para == true)`
 - Setting of the **num_threads** clause
 - ◆ E.g.: `#pragma omp parallel num_threads(10)`
 - Use of the **omp_set_num_threads()** library function
 - ◆ Called **BEFORE** the parallel region
 - Setting of the **OMP_NUM_THREADS** environment variable
 - ◆ Called **BEFORE** the parallel region
 - By default - usually the number of CPUs on a node

Nested Parallel Region

```
// A total of 6 "hello world!" is printed
#pragma omp parallel num_threads(2)
{
    #pragma omp parallel num_threads(3)
    {
        printf("hello world!");
    }
}
```

- check if nested parallel regions are enabled
 - **omp_get_nested ()**
- To disable/enable nested parallel regions:
 - **omp_set_nested (bool)**
 - Setting of the **OMP_NESTED** environment variable
- If nested is not supported or enabled:
 - **Only one thread is created** for the nested parallel region code

OpenMP Outline

- Parallel Region Construct
 - Parallel Directive
- Working-Sharing Construct
 - DO/for Directive
 - SECTIONS Directive
 - SINGLE Directive
- Synchronization Construct
- Data Scope Attribute Clauses
- Run-Time Library Routines

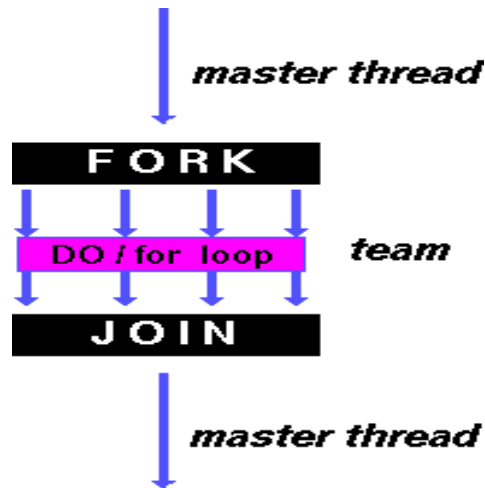
Work-Sharing Constructs

■ Definition:

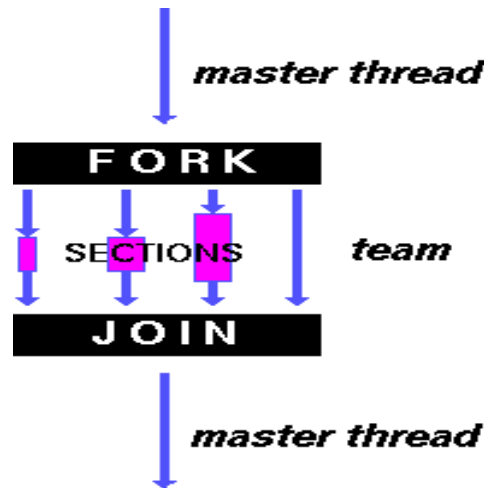
- A work-sharing construct divides the execution of the enclosed code region among the threads that encounter it
- Work-sharing constructs **DO NOT** launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied **barrier at the end** of a work sharing construct

Type of Work-Sharing Constructs

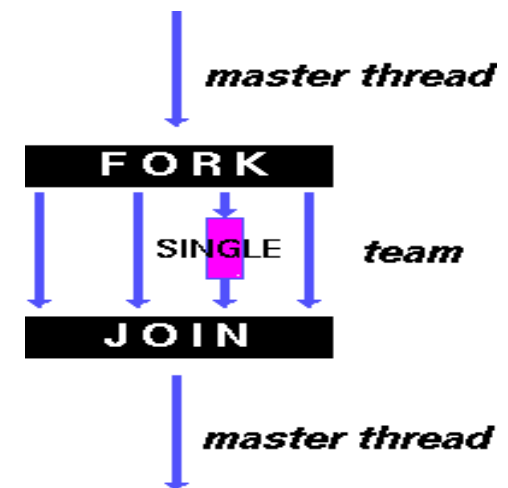
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections of code. Each section is executed by a thread.



SINGLE - serializes a section of code by running with a single thread.



■ Notice:

- should be enclosed within a **parallel region** for parallelism

DO / for Directive

- Purpose: indicate the iterations of the **loop immediately following it must be executed in parallel** by the team of threads

```
#pragma omp for [clause .....]  
                                schedule (type [,chunk])  
                                ordered  
                                nowait  
                                collapse (n)  
  
for_loop
```

- Do/for Directive Specific Clauses:
 - **nowait**: Do not synchronize threads at the end of the loop
 - **schedule**: Describes how iterations are divided among threads
 - **ordered**: **Iterations** must be executed as in a serial program
 - **collapse**: Specifies how many loops in a **nested loop** should be collapsed into one large iteration space and divided according to the schedule clause

DO / for Directive --- Schedule Clause

■ **STATIC**

- Loop iterations are divided into **chunks**
- If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads
- Then statically assigned to threads

■ **DYNAMIC:** When a thread finishes one **chunk (default size: 1)**, it is **dynamically assigned another**

■ **GUIDED:** Similar to DYNAMIC except **chunk size decreases over time** (better load balancing)

■ **RUNTIME:** The scheduling **decision** is deferred until runtime by the environment variable **OMP_SCHEDULE**

■ **AUTO:** The scheduling decision is **delegated to the compiler and/or runtime system**

Scheduling Examples

■ A for loop with 100 iterations and 4 threads:

➤ `schedule(static, 10)`

- ◆ Thread0: Iter0-10, Iter40-50, Iter80-90
- ◆ Thread1: Iter10-20, Iter50-60, Iter90-100
- ◆ Thread2: Iter20-30, Iter60-70
- ◆ Thread3: Iter30-40, Iter70-80

➤ `schedule(dynamic, 10)`

- ◆ Thread0: Iter0-10, Iter70-80, Iter80-90, Iter90-100
- ◆ Thread1: Iter10-20, Iter50-60
- ◆ Thread2: Iter20-30, Iter60-70
- ◆ Thread3: Iter30-40, Iter40-50

Scheduling Examples

■ A for loop with 100 iterations and 4 threads:

➤ `schedule(guided, 10)`

- ◆ Thread0: Iter0-10, Iter40-50, Iter80-85
- ◆ Thread0: Iter10-20, Iter50-60, Iter85-90
- ◆ Thread0: Iter20-30, Iter60-70, Iter90-95
- ◆ Thread0: Iter30-40, Iter70-80, Iter95-100

DO / for Directive --- Example

```
#include <omp.h>
#define NUM_THREAD 2
#define CHUNKSIZE 100
#define N 1000
main () {
    int a[N], b[N], c[N];
    /* Some initializations */
    for (int i=0; i < N; i++) a[i] = b[i] = i;
    int chunk = CHUNKSIZE;
    int thread = NUM_THREAD;
```

Shared variables
among threads

Private variables
of each thread

```
#pragma omp parallel num_thread(thread) shared(a,b,c) private(i)
{
    #pragma omp for schedule(dynamic,chunk) nowait
    for (int i=0; i < N; i++) c[i] = a[i] + b[i];
} /* end of parallel section */
}
```


DO / for Directive --- Order

```
#pragma omp parallel for
for (int i = 0; i < 10; i++)
    printf("i=%d, thread = %d\n",
           i, omp_get_thread_num());
```

```
i=2, thread = 0
i=0, thread = 1
i=1, thread = 2
i=3, thread = 1
i=4, thread = 0
i=8, thread = 2
i=5, thread = 1
i=6, thread = 2
i=9, thread = 1
i=7, thread = 1
```

```
#pragma omp parallel for order
for (int i = 0; i < 3; i++)
    printf("i=%d, thread = %d\n",
           i, omp_get_thread_num());
```

```
i=0, thread = 0
i=1, thread = 1
i=2, thread = 2
i=3, thread = 1
i=4, thread = 0
i=5, thread = 2
i=6, thread = 1
i=7, thread = 2
i=8, thread = 1
i=9, thread = 1
```

DO / for Directive --- Collapse

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        printf("i=%d, j=%d, thread = %d\n",
               i, j, omp_get_thread_num());
```

```
i=1, j=0, thread = 1
i=2, j=0, thread = 2
i=0, j=0, thread = 0
i=1, j=1, thread = 1
i=2, j=1, thread = 2
i=0, j=1, thread = 0
i=1, j=2, thread = 1
i=2, j=2, thread = 2
i=0, j=2, thread = 0
```

```
#pragma omp parallel num_thread(6)
#pragma omp for schedule(dynamic)
                        collapse(2)
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        printf("i=%d, j=%d, thread = %d\n",
               i, j, omp_get_thread_num());
```

```
i=0, j=0, thread = 0
i=0, j=2, thread = 1
i=1, j=0, thread = 2
i=2, j=0, thread = 4
i=0, j=1, thread = 0
i=1, j=2, thread = 3
i=2, j=2, thread = 5
i=1, j=1, thread = 2
i=2, j=1, thread = 4
```

SECTIONS Directive

- A non-iterative work-sharing construct
- It specifies that the enclosed **section(s)** of **CODE** are to be divided among the threads in the team
- Independent **SECTION** directives are nested within a **SECTIONS** directive
- Each SECTION is **executed ONCE** by **ONE** thread
- The mapping between threads and sections is **decided by the library implementation**

```
#pragma omp sections [clause .....]  
{  
    #pragma omp section  
        structured_block  
  
    #pragma omp section  
        structured_block  
}
```

SECTIONS Directive --- Example

```
int N = 1000
int a[N], b[N], c[N], d[N];

#pragma omp parallel num_thread(2) shared(a,b,c,d) private(i)
{
    #pragma omp sections          /* specify sections*/
    {
        #pragma omp section /* 1st section*/
        {
            for (int i=0; i < N; i++) c[i] = a[i] + b[i];
        }
        #pragma omp section /* 2nd section*/
        {
            for (int i=0; i < N; i++) d[i] = a[i] + b[i];
        }
    } /* end of section */
} /* end of parallel section */
```

SINGLE Directive

- The SINGLE directive specifies that 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)**
- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a **nowait** clause is specified
- Example:

```
int input;
#pragma omp parallel num_thread(10) shared(input)
{
    // computing code that can be prcessed in parallel
    #pragma omp single    /* specify section
    {
        scanf("%d", &input);
    } /* end of seralized I/O call */

    printf("input is %d", input);
} /* end of parallel section */
```

OpenMP Outline

- Parallel Region Construct
 - Parallel Directive
- Working-Sharing Construct
 - DO/for Directive
 - SECTIONS Directive
 - SINGLE Directive
- Synchronization Construct
- Data Scope Attribute Clauses
- Run-Time Library Routines

Synchronization Constructs

■ For synchronization purpose among threads

```
#pragma omp [synchronization_directive] [clause .....]  
structured_block
```

■ Synchronization Directives

- **master**: only executed by the **master** thread
 - ◆ No implicit barrier at the end
 - ◆ More efficient than SINGLE directive
- **critical**: must be executed by **only one** thread at a time
 - ◆ Threads will be blocked until the critical section is clear
- **barrier**: blocked until **all threads** reach the call
- **atomic**: **memory** location must be **updated** atomically
 - ◆ provide a mini-critical section

LOCK OpenMP Routine

- `void omp_init_lock(omp_lock_t *lock)`
 - Initializes a lock associated with the lock variable
- `void omp_destroy_lock(omp_lock_t *lock)`
 - Disassociates the given lock variable from any locks
- `void omp_set_lock(omp_lock_t *lock)`
 - Force the thread to wait until the specified lock is available
- `void omp_unset_lock(omp_lock_t *lock)`
 - Releases the lock from the executing subroutine
- `int omp_test_lock(omp_lock_t *lock)`
 - Attempts to set a lock, but does **NOT** block **if unavailable**

Example & Comparison

■ Advantage of using critical over lock:

- no need to declare, initialize and destroy a lock
- you always have explicit control over where your critical section ends
- Less overhead with compiler assist

```
#include <omp.h>
main () {
    int count=0;
    #pragma omp parallel
        #pragma omp critical
            count++;
}
```

```
#include <omp.h>
main () {
    int count=0;
    omp_lock_t *lock;
    omp_init_lock(lock)
    #pragma omp parallel
    {
        omp_set_lock(lock);
        count++;
        omp_unset_lock(lock);
    }
    omp_destroy_lock(lock)
}
```

OpenMP Outline

- Parallel Region Construct
 - Parallel Directive
- Working-Sharing Construct
 - DO/for Directive
 - SECTIONS Directive
 - SINGLE Directive
- Synchronization Construct
- **Date Scope Attribute Clauses**
- Run-Time Library Routines

OpenMP Data Scope

- This is critical to understand the scope of each data
 - OpenMP is based on **shared memory programming model**
 - Most variables are shared by default
- Global shared variables:
 - File scope variables, static
- Private non-shared variables:
 - Loop index variables
 - Stack variables in subroutines called from **parallel regions**
- Data scope can be **explicitly defined by clauses...**
 - **PRIVATE** , **SHARED**, FIRSTPRIVATE, LASTPRIVATE
 - DEFAULT, REDUCTION, COPYIN

Date Scope Attribute Clauses

■ **PRIVATE** (var_list):

- Declares variables in its list to be **private** to each thread; variable value is **NOT initialized & will not be maintained outside the parallel region**

■ **SHARED** (var_list):

- Declares variables in its list to be **shared among all threads**
- By default, all variables in the work sharing region are shared except the loop iteration counter.

■ **FIRSTPRIVATE** (var_list):

- Same as **PRIVATE** clause, but the **variable is INITIALIZED** according to the value of their original objects prior to entry into the parallel region

■ **LASTPRIVATE** (var_list)

- Same as **PRIVATE** clause, with a **copy from the LAST loop iteration or section to the original variable object**

Examples

■ firstprivate (var_list)

```
int var1 = 10;
#pragma omp parallel firstprivate (var1)
{
    printf("var1:%d" var1);
}
```

■ lastprivate (var_list)

```
int var1 = 10;
#pragma omp parallel lastprivate (var1) num_thread(10)
{
    int id = omp_get_thread_num();
    sleep(id);
    var1=id;
}
printf("var1:%d", var1);
```

Date Scope Attribute Clauses

- **DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)**
 - Allows the user to **specify a default scope for ALL variables** in the parallel region
- **COPYIN (var_list)**
 - **Assigning the same variable value** based on the instance from the **master thread**
- **COPYPRIVATE (var_list)**
 - **Broadcast values** acquired by a single thread directly to all instances in the other thread
 - Associated with the **SINGLE** directive
- **REDUCTION (operator: var_list)**
 - **A private copy** for each list variable is **created for each thread**
 - Performs a **reduction on all variable instances**
 - Write the **final result to the global shared copy**

Reduction Clause Example

```
#include <omp.h>
main () {
    int i, n, chunk, a[100], b[100], result;
    n = 10; chunk = 2; result = 0;
    for (i=0; i < n; i++) a[i] = b[i] = 1;

    #pragma omp parallel for default(shared) private(i) \
                schedule(static,chunk) reduction(+:result)
    {
        for (i=0; i < n; i++) result = result + (a[i] * b[i]);
    }
    printf("Final result= %f\n",result);
}
```

■ Reduction operators:

➤ +, *, &, |, ^, &&, ||

OpenMP Clause Summary

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

■ **Synchronization Directives DO NOT** accept clauses

OpenMP Outline

- Parallel Region Construct
 - Parallel Directive
- Working-Sharing Construct
 - DO/for Directive
 - SECTIONS Directive
 - SINGLE Directive
- Synchronization Construct
- Data Scope Attribute Clauses
- Run-Time Library Routines

Run-Time Library Routines

- `void omp_set_num_threads(int num_threads)`
 - Sets the number of threads that will be used in the next parallel region
- `int omp_get_num_threads(void)`
 - Returns the number of threads currently executing for the parallel region
- `int omp_get_thread_num(void)`
 - Returns the thread number of the thread, within the team, making this call
 - The master thread of the team is thread 0
- `int omp_get_thread_limit(void)`
 - Returns the maximum number of OpenMP threads available to a program
- `int omp_get_num_procs(void)`
 - Returns the number of processors that are available to the program
- `int omp_in_parallel(void)`
 - determine if the section of code which is executing is parallel or not

Many others are available for more complicated usage

Reference

- Textbook:

- Parallel Computing Chap8

- openMP Tutorial

- <https://computing.llnl.gov/tutorials/openMP/>

- openMP API

- <http://gcc.gnu.org/onlinedocs/libgomp.pdf>