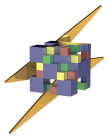
Lecture 12

OpenMP

이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee



THUNDER Research Group
Seoul National University
서울대학교 천둥 연구실

OpenMP

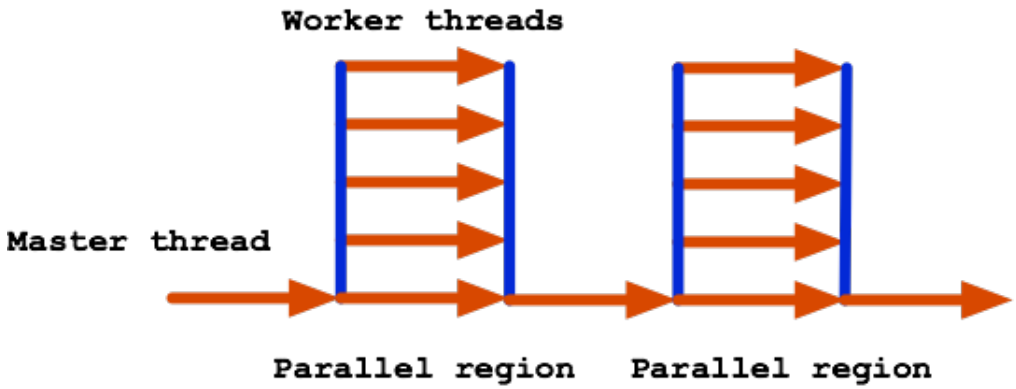
- An API for shared-memory parallelism in C, C++ and Fortran programs
  - A set of compiler directives, library routines, and environment variables for parallel application programmers
- Portable across shared-memory architectures
- Compiler generates thread program and synchronization
  - Will not parallelize automatically
- Accelerator support for GPUs
- OpenMP Application Program Interface version 5.1
  - http://www.openmp.org

Threading in OpenMP

- Thread
  - An execution entity with a stack and associated static memory, called thread-private memory
- OpenMP thread
  - A thread that is managed by the OpenMP runtime system
- Thread-safe routine
  - A routine that performs the intended function even when executed concurrently (by more than one thread)

OpenMP Execution Model

- Fork-join parallelism
  - Fork - master thread spawns a team of threads
    - The master thread always has thread ID 0
  - Join - when the team of threads complete the tasks in the parallel section, they terminate synchronously, leaving only the master thread
- Parallel region - a block of code executed by all threads simultaneously
- Implementation optimization
  - Worker threads spin waiting on next fork



Pragmas

- Special preprocessor instructions
  - To allow behaviors that are not part of the basic C specification
  - Compilers that do not support the pragmas ignore them

- Format
  - **`#pragma omp directive_name [ clause [ clause ] ... ] new-line`**
  - case sensitive

- Include file
  - **`#include <omp.h>`**

- Conditional compilation
  - **`#ifdef _OPENMP`**
  - **`#endif`**

Query Functions

- `int omp_get_num_threads(void);`
  - Returns the number of threads currently in the team executing the parallel region from which it is called

- `int omp_get_thread_num(void);`
  - Returns the thread number, within the team, that lies between 0 and `omp_get_num_threads() - 1`, inclusive
  - The master thread of the team is thread 0

```
double x[1000];
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int n = omp_get_num_threads();
    foo( id, x );
}
```

Parallel Region Construct

- OpenMP programs begin as a single process, the master thread, until they reach a parallel region, which then spawns a team of threads
  - Each thread executes the same code (SPMD)
  - The number of threads in the team remains constant for the duration of that parallel region
  - Within a parallel region, thread numbers uniquely identify each thread
  - By default, all variables are shared
- Format

```
#pragma omp parallel [ clause [[,] clause ] ... ] new-line
    structured-block
```

Hello World in OpenMP

- The program should be correct without the pragmas and library function calls
- gcc -fopenmp ...

```
#include <omp.h>
```

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

Hello World in OpenMP (cont'd)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void hello(void);

void main(int argc, char* argv[]) {
    int cnt_threads = strtol(arg[1], NULL, 10);

    #pragma omp parallel num_threads(cnt_threads)
    hello();

    return 0;
}

void hello(void) {
    int my_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    printf("Hello world! %d %d\n", my_id, num_threads);
}
```

Another Example for Parallel Region

```
#include <stdio.h>
#include <omp.h>

void main() {
    int num_threads, tid;
    num_threads = omp_get_num_threads();

    printf("Sequential section: # of threads = %d\n", num_threads);

#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Parallel section: Hello world from thread %d\n",
               tid);

        if (tid == 0) {
            num_threads = omp_get_num_threads();
            printf("Parallel section: # of threads = %d\n",
                   num_threads);
        }
    }
}
```

Setting the Number of Threads

```
#include <stdio.h>
#include <omp.h>

void main() {
    int num_threads, tid;
    omp_set_num_threads(2);
    num_threads = omp_get_num_threads();

    printf("Sequential section: # of threads = %d\n",num_threads);

#pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num();
        printf("Parallel section: Hello world from thread %d\n",
               tid);

        if (tid == 0) {
            num_threads = omp_get_num_threads();
            printf("Parallel section: # of threads = %d\n",
                   num_threads);
        }
    }
}
```

if/private/shared clauses

- if (scalar_expression)
  - Only execute in parallel if scalar_expression evaluates to true
  - Otherwise, execute serially
- private(list)
  - All references are to the local variable
  - Values are undefined on entry and exit
- shared(list)
  - Accessible by all threads in the team
- firstprivate
  - Private and copy initial value from global variable
- lastprivate
  - Private and copy back final value to global variable

Work Sharing Constructs

- A work-sharing construct distributes the execution of the associated region among the members of the team that encounters it
- If the team consists of only one thread then the work sharing region is not executed in parallel
- A work-sharing region has no barrier on entry
  - An *implied barrier exists* at the end of the work-sharing region
  - If a nowait clause is present, an implementation may omit the barrier

Loop Construct

- Specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team
- The iterations are distributed across threads that already exist in the team executing the parallel region

```
#pragma omp for [ clause [[,] clause ] ... ] new-line
    for-loops
```

Loop Construct (cont'd)

```
#include <math.h>
void nowait_example(int n, int m, float *a, float *b,
                    float *y, float *z)
{
    int i;

    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Thread Scheduling Clause

- **`schedule(static [, chunk_size])`**
  - Iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number

- **`schedule(dynamic [, chunk_size])`**
  - The iterations are distributed to threads in the team in chunks as the threads request them
    - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed

Thread Scheduling Clause (cont'd)

- **`schedule(guided [, chunk_size])`**
  - To reduce the overhead of dynamic
  - The iterations are assigned to threads in the team in chunks as the executing threads request them
    - Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned
  - For a chunk_size of k, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to k
    - With the restriction that the chunks do not contain fewer than k iterations
- **`auto`**
  - The decision regarding scheduling is delegated to the compiler and/or runtime system

Sections Construct

- A non-iterative work-sharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team

```
#pragma omp sections [clause[[,] clause] ...] new-line
    {[#pragma omp section new-line]
          structured-block
     [#pragma omp section new-line
          structured-block]
        ...
    }
```

- Each structured block is executed once by one of the threads in the team

Sections Construct (cont'd)

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {.A.}
        #pragma omp section
        {.B.}
        #pragma omp section
        {.C.}
        #pragma omp section
        {.D.}
    } /* omp end sections */
} /* omp end parallel */
```

Sections Construct (cont'd)

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        for (i = 0; i < n; i++)
            d[i] = 1.0/a[i];
        #pragma omp section
        for (i = 0; i < n-1; i++)
            b[i] = a[i] + c[i+1];
    }
}
```

Single Construct

- Specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread)
- The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the single construct unless a nowait clause is specified

```
#pragma omp single [clause[[,] clause] ...] new-line
    structured-block
```

Single Construct (cont'd)

```
#include <stdio.h>
void work1() {}
void work2() {}
void single_example()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");
        work1();
        #pragma omp single
        printf("Finishing work1.\n");
        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");
        work2();
    }
}
```

Combined Work-sharing Constructs

- Shortcuts for specifying a work-sharing construct nested immediately inside a parallel construct
- The semantics of these directives are identical to that of explicitly specifying a parallel construct containing one work-sharing construct and no other statements

```
#pragma omp parallel for [clause[[,] clause] ...] new-line
    for-loop

#pragma omp parallel sections [clause[[,] clause] ...] new-line
{
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block ]
...
}
```

Combined Work-sharing Constructs (cont'd)

```
void simple(int n, float *a, float *b)
{
    int i;
    #pragma omp parallel for
    for (i = 1; i < n; i++)
     /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```

Reduction

- reduction(op : list)
- A local copy of each list variable is made and initialized depending on the op
- Updates occur on the local copy
- Local copies are reduced to the original global variable
- Many different associative operators
  - `+, *, -, &, |, ^, &&, ||`

```
double sum = 0.0, A[MAX];
int i;

#pragma omp parallel for reduction (+:sum)
    for (i = 0; i < MAX; i++) {
        sum + = A[i];
    }
ave = sum/MAX;
```

* Task Construct (version 3.0)

- Defines an explicit task
- Use the task construct when you want to identify a block of code to be executed in parallel with the code outside the task region
  - Useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms for which other OpenMP work-sharing constructs are inadequate
- The data sharing default for tasks is firstprivate

```
#pragma omp task [clause[[,] clause] ...] new-line
    structured-block
```

* Task Construct (cont'd)

- The encountering thread may immediately execute the task, or defer its execution
  - In the latter case, any thread in the team may be assigned the task
  - Completion of the task can be guaranteed using task synchronization constructs

* Task Construct (cont'd)

```
struct node {
    struct node *left;
    struct node *right;
};

extern void process(struct node *);

void traverse( struct node *p )
{
    if (p->left)
    #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
    #pragma omp task // p is firstprivate by default
        traverse(p->right);

    #pragma omp taskwait
    process(p);
}
```

* Taskyield Construct

- Specifies that the current task can be suspended in favor of execution of a different task

```
#pragma omp taskyield new-line
```

* Taskyield Construct (cont'd)

```
#include <omp.h>

void something_useful ( void );
void something_critical ( void );

void foo ( omp_lock_t * lock, int n )
{
    int i;

    for ( i = 0; i < n; i++ )
        #pragma omp task
        {
            something_useful();
            while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

Master Construct

- Specifies a structured block that is executed by the master thread of the team

```
#pragma omp master new-line
    structured-block
```

Master Construct (cont'd)

```
#include <stdio.h>
extern float average(float,float,float);

void master_example( float* x, float* xold, int n, float tol )
{
    int c, i, toobig;
    float error, y;

    c = 0;
    #pragma omp parallel
    {
        do{
            #pragma omp for private(i)
                for( i = 1; i < n-1; ++i ){
                     xold[i] = x[i];
                }
            #pragma omp single
                {
                    toobig = 0;
                }
            #pragma omp for private(i,y,error) reduction(+:toobig)
                for( i = 1; i < n-1; ++i ){
                    y = x[i];
                    x[i] = average( xold[i-1], x[i], xold[i+1] );
                    error = y - x[i];
                    if( error > tol || error < -tol ) ++toobig;
                }
            #pragma omp master
                {
                    ++c;
                    printf( "iteration %d, toobig=%d\n", c, toobig );
                }
         }while( toobig > 0 );
    }
}
```

Critical Construct

- Restricts execution of the associated structured block to a single thread at a time

```
#pragma omp critical [(name)] new-line
    structured-block
```

Critical Construct (cont'd)

```
int dequeue(float *a);

void work(int i, float *a);

void critical_example(float *x, float *y)
{
    int ix_next, iy_next;
    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
            ix_next = dequeue(x);

        work(ix_next, x);

        #pragma omp critical (yaxis)
            iy_next = dequeue(y);

        work(iy_next, y);
    }
}
```

Barrier Construct

- Specifies an explicit barrier at the point at which the construct appears

```
#pragma omp barrier new-line
```

* Taskwait Construct

- Specifies a wait on the completion of child tasks of the current task

```
#pragma omp taskwait new-line
```

Atomic Construct

- Ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values

```
#pragma omp atomic [read | write | update | capture ] new-line
    expression-stmt
```

```
#pragma omp atomic capture new-line
    structured-block
```

Atomic Construct (cont'd)

```
float work1(int i) {
    return 1.0 * i;
}

float work2(int i) {
    return 2.0 * i;
}

void atomic_example(float *x, float *y, int *index, int n)
{
    int i;
    #pragma omp parallel for shared(x, y, index, n)
        for (i=0; i<n; i++) {
            #pragma omp atomic update
                x[index[i]] += work1(i);
            y[i] += work2(i);
        }
}
```

* Ordered Construct

- Specifies a structured block in a loop region that will be executed in the order of the loop iterations
- This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel
- The ordered clause must be present on the loop construct if any ordered region ever binds to a loop region arising from the loop construct

```
#pragma omp ordered new-line
    structured-block
```

* Ordered Construct (cont'd)

```
#include <stdio.h>
void work(int k) {
    #pragma omp ordered
        printf(" %d\n", k);
}

void ordered_example(int lb, int ub, int stride) {
    int i;

    #pragma omp parallel for ordered schedule(dynamic)
        for (i=lb; i<ub; i+=stride)
            work(i);
}

int main() {
    ordered_example(0, 100, 5);
    return 0;
}
```

* Ordered Construct (cont'd)

```
#pragma omp parallel for private(myval) ordered
{
   for(i=1; i<=n; i++){
      myval = do_lots_of_work(i);
      #pragma omp ordered
      {
         printf("%d %d\n", i, myval);
      }
   }
}
```

OpenMP Memory Consistency

- All OpenMP threads have access to a place to store and to retrieve variables, called the memory

- In addition, each thread is allowed to have its own temporary view of the memory

OpenMP Memory Consistency (cont'd)

- Relaxed consistency
  - Similar to weak ordering
- A thread's temporary view of memory is not required to be consistent with memory at all times
  - A value written to a variable can remain in the thread's temporary view until it is forced to memory at a later time
  - Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory
- The OpenMP flush operation enforces consistency between the temporary view and memory
  - Synchronization operation in weak ordering

Flush Construct

- Executes the OpenMP flush operation
- This operation makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables explicitly specified or implied

```
#pragma omp flush [(list)] new-line
```

Producer and Consumer

```
flag = 0;
#pragma omp parallel
{
    #pragma omp section
    {
        produce();
        #pragma omp flush

        flag = 1;
        #pragma omp flush(flag)
    }

    #pragma omp section
    {
        while (!flag) {
            #pragma omp flush(flag)
        }

        #pragma omp flush
        consume();
    }
}
```

Environment Variables

- **`OMP_NUM_THREADS`**
  - Sets the number of threads to use during execution
  - When dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
  - **`export OMP_NUM_THREADS=16`**

- **`OMP_SCHEDULE`**
  - Applies only to for and parallel for directives that have the schedule type auto
  - Sets schedule type and chunk size for all such loops
  - **`export OMP_SCHEDULE=GUIDED, 4`**