Multicore Computing
Lecture06 - OpenMP Part II

남 범 석

bnam@skku.edu

Nesting `parallel` Directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
  - If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
  - In this case, each parallel directive creates a new team of threads.

Nesting **parallel** Directives

```
int i=0;
int cnt=0;

omp_set_nested(true);

#pragma omp parallel shared(cnt)
{
  printf("omp_get_thread_num()=%d\n", omp_get_thread_num());

  #pragma omp parallel shared(cnt)
  {
      printf("i=%d, thread-%d\n", i, omp_get_thread_num());
      #pragma omp atomic
      cnt++;
  }
}
printf("cnt=%d\n", cnt);
```

Synchronization Constructs in OpenMP

- OpenMP provides a variety of synchronization constructs:

```
#pragma omp barrier
#pragma omp single [clause list]
   structured block
#pragma omp master
   structured block
#pragma omp critical [(name)]
   structured block
#pragma omp ordered
   structured block
```

barrier Directive

- OpenMP
  - Barrier synchronization
    - Wait until all the threads in a team reach to the point
  - #pragma omp barrier

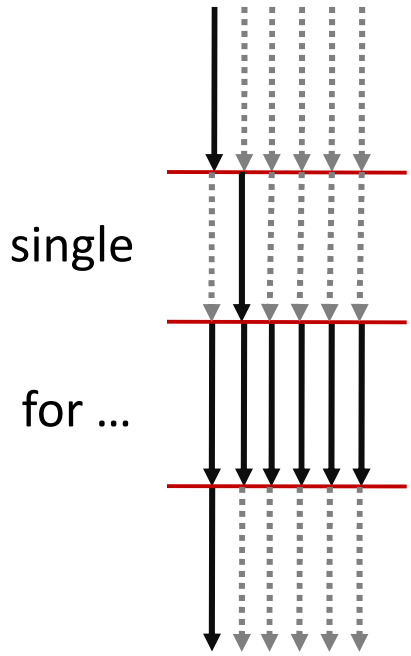
```
main() {
    #pragma omp parallel
    sub();
}
sub() {
    work1();
    #pragma omp barrier
    work2();
}
```

single Directive

- Executed by one thread within a parallel region
  - Any thread can execute the single region
  - Implicit barrier synchronization at the end

```
#pragma omp parallel
{
  #pragma omp single
  {
     a = 10;
  } /* implicit barrier */

  #pragma omp for
  for (i=0; i<N; i++)
     B[i] = a;
}
/* end of parallel region */
```

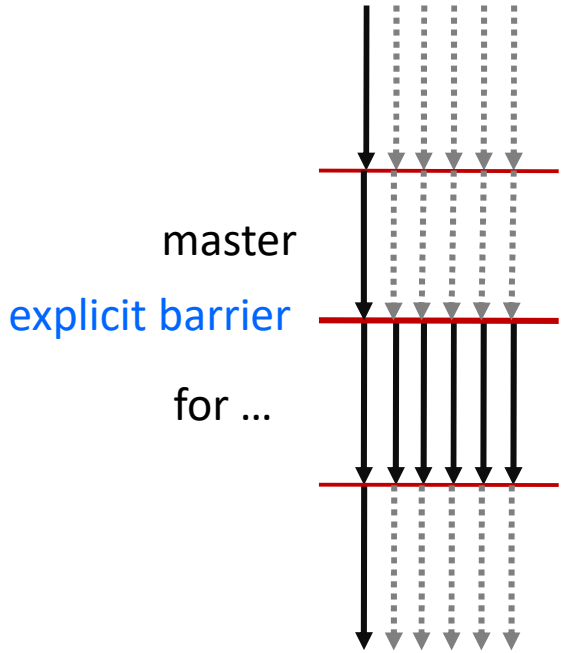


master Directive

- Executed by the master thread
  - No implicit barrier
  - If a barrier is needed for correctness, it must be specified

```
#pragma omp parallel
{
  #pragma omp master
  {
     a = 10;
  } /* no barrier */
  #pragma omp barrier

  #pragma omp for
  for (i=0; i<N; i++)
     B[i] = a;
}
/* end of parallel region */
```



`critical` Directive

- Format
  - `# pragma omp critical`
  - Provides mutual exclusion of the following structured block to all threads in a team

```
# pragma omp critical
Enqueue(&queue, &message)
```

```
# pragma omp critical
Dequeue(&queue, &message)
```

- Limitation
  - Distinct critical sections are treated as one composite critical section
  - Serialization of all threads

Critical sections for queue

```
# pragma omp critical
Enqueue(&queue, &message)
```

```
# pragma omp critical
Dequeue(&queue, &message)
```

Critical sections for stack

```
# pragma omp critical
Push(&stack, &message)
```

```
# pragma omp critical
Pop(&stack, &message)
```

Named `critical` Directive

- Format
  - `# pragma omp critical (name)`
  - Specifies the name of a critical section
  - OpenMP provides mutual exclusion to the critical sections having the same name

```
# pragma omp critical(queue)
Enqueue(&queue, &message)
```

```
# pragma omp critical(queue)
Dequeue(&queue, &message)
```

```
# pragma omp critical(stack)
Push(&stack, &message)
```

```
# pragma omp critical(stack)
Pop(&stack, &message)
```

- Limitation
  - Distinction of critical sections is made at compilation time
  - No critical section distinction between different data structures at runtime

Lock APIs in OpenMP

- Usage
  - `omp_lock_t lock;`
  - `omp_init_lock(&lock); omp_destroy_lock(&lock);`
  - `omp_set_lock(&lock);  omp_unset_lock(&lock);`

- Example

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

atomic Directive

- Format
  - `#pragma omp atomic`
- It only protects critical sections that consist of a single C assignment statement
- Valid statement format:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

- Supported operations:

```
+, *, -, /, &, ^, |, <<, or >>
```

ordered Directive

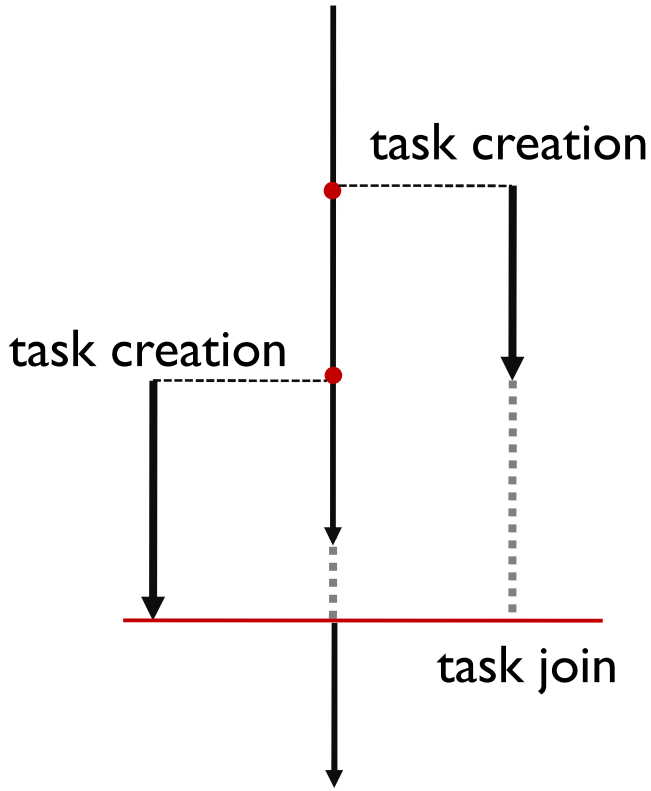
- Ensures loop-carried dependence does not cause a data race

```
#pragma omp parallel for ordered private(i) shared(a, b)
{
   for (i = 0; i < mmax; i++)
   {
       /* other processing on b[i] */
       #pragma omp ordered
       b[i] = b[i-1] + a[i]
   }
}
```

Data Handling Clauses

- `lastprivate`
  - The last value of a variable is kept after join of threads
- `threadprivate`
  - Each thread has a local copy of a variable similar to private
  - But, the variable is alive across different parallel constructs
- `copyin`
  - Initialize a threadprivate variable from the value of variable in a master thread

- Task model (OpenMP 3.0 – released, May 2008)
  - Task creation and join
  - Can handle
    - Unbounded loops
    - Recursive algorithms
    - Producer/consumer
  - #pragma omp task [clause list]
    - task
    - taskwait



Example: OpenMP Task

- Task level parallelism

```
void traverse (NODE *p) {
  if (p->left)
     traverse(p->left);
  if (p->right)
     traverse(p->right);

  process(p);
}
```

```
void traverse (NODE *p) {
  if (p->left)
     #pragma omp task
     traverse(p->left);
  if (p->right)
     #pragma omp task
     traverse(p->right);
  #pragma omp taskwait
  process(p);
}
```

- Post-order visit
- Individual join in taskwait
  - Children of a task are joined at taskwait
- To wait for all the descendant tasks
  - Join all the task created so far
  - Taskgroup is needed (Not defined in OpenMP 3.0)

Example: Linked List Traversal

```
while(my_pointer) {
  do_independent_work (my_pointer);
  my_pointer = my_pointer->next ;
} // End of while loop
```

Example: Linked List Traversal

```
my_pointer = listhead;
#pragma omp parallel
{
  #pragma omp single
  {
    while(my_pointer) {
      #pragma omp task firstprivate(my_pointer)
      {
        do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    }
  } // End of single
} // End of parallel region
```

Example: Linked List Traversal

```
my_pointer = listhead;
#pragma omp parallel
{
  #pragma omp single nowait
  {
    while(my_pointer) {
      #pragma omp task firstprivate(my_pointer)
      {
        do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    }
  } // End of single - no implied barrier
} // End of parallel region - implicit barrier
```

OpenMP Library Functions

- Control the execution of threaded programs.
  - `void omp_set_num_threads (int num_threads);`
    - Set max # of threads for next parallel construct
  - `int omp_get_num_threads ();`
    - Get active # of threads
  - `int omp_get_max_threads ();`
    - Get maximum # of threads
  - `int omp_get_thread_num ();`
    - Return thread ID (from 0 to MAX-1)
  - `int omp_get_num_procs ();`
    - Get # of processors available
  - `int omp_in_parallel();`
    - Determines whether running in parallel construct

OpenMP Library Functions

- Controlling and monitoring thread creation
  - `void omp_set_dynamic (int dynamic_threads);`
    - Enable/disable dynamic change of # of threads for parallel construct
  - `int omp_get_dynamic ();`
    - Query whether dynamic change of # of threads for parallel construct is enabled or not
  - `void omp_set_nested (int nested);`
    - Enable nested parallel directive
  - `int omp_get_nested ();`
    - Query whether nested parallel directive is enabled or not

Environment Variables in OpenMP

- `OMP_NUM_THREADS`
  - Specifies the default number of threads created upon entering a parallel region.
- `OMP_SET_DYNAMIC`
  - Determines if the number of threads can be dynamically changed.
- `OMP_NESTED`
  - Turns on nested parallelism.
- `OMP_SCHEDULE`
  - Scheduling of for-loops if the clause specifies runtime
  - Example

```
$ export OMP_SCHEDULE="static, 1"
$ ./omp_program 4
```

→ static scheduling with chunksize of 1

OpenMP Programming Practice

- OpenMP
  - Start with a parallelizable algorithm
  - Implement serially, mostly ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Annotation with directives for parallelization & synchronization
  - Test & Debug

- Ideal way
  - Start with some algorithm
  - Implement serially, ignoring
    - Data races
    - Synchronization
    - Threading syntax
  - Test & Debug
  - Auto-magically parallelize

OpenMP Summary

- OpenMP is:
  - An API that may be used to explicitly direct multi-threaded, shared memory parallelism
  - Portable
    - C/C++ and Fortran support
    - Implemented on most Unix variants and Windows
  - Standardized
    - Major computer HW and SW vendors jointly defines (OpenMP.org)

- OpenMP does NOT:
  - Support distributed memory systems
    - but Cluster OpenMP does
  - Automatically parallelize
  - Have data distribution controls
  - Guarantee efficiency, freedom from data races, …