OpenMP

Jingke Li

Portland State University

OpenMP

An API for a set of compiler directives, library routines, and environment variables that can be used to specify *shared-memory* parallelism.

Main Characteristics:

- (Largely) *Non-intrusive* — OpenMP directives do not change a program's sequential semantics.
- *Explicit parallelism* — OpenMP offers programmer mechanisms for specifying multiple forms of parallelism.

A Brief History:

- Developed by a joint committee in the 90s, supports Fortran, C, and C++.
- We use version 3.1, since it is implemented in our GCC 4.8.4 compiler.
- Current version is 4.5, which will be partially supported in GCC 6.1.
- The official website is `www.openmp.org`.

Overview

- *Compiler Directives* — the main part of OpenMP, non-intrusive
  - for specifying every aspect of parallel execution: parallelism, work distribution, and synchronization
  - all except one are *executable* directives, they may be placed in an executable context
  - many accepts *clauses* for specifying data-sharing attributes of variables
- *Runtime Library Routines* — intrusive
  - allow dynamic information to be used, *e.g.* number of threads
  - allow more flexible control, *e.g.* nested locking
- *Environment Variables*
  - allow program-independent control over a set of parameters, *e.g.* number of threads

OpenMP Execution Model

OpenMP uses the *fork-join* model of parallel execution.

An OpenMP program always begins as a single *master* thread.

1. The master thread executes sequentially until the first *parallel region* is encountered.
2. The master thread then forks a team of parallel threads.
   - The default number of threads is specified outside of the program in an environment variable (if exists), or equals to the number of cores.
3. The statements in the parallel region are then executed by threads — each executes a copy of the statements by default.
4. When the team threads complete the execution, they synchronize and terminate, leaving only the master thread.

Parallel Regions

The programmer specifies parallel regions in a program with the `parallel` directive in the form of comment lines (with a special prefix).

Example:

C:

```c
int main () {
  #pragma omp parallel
  { work(); }
}
```

Fortran:

```fortran
   program example
!$omp parallel
   work();
!$omp end parallel
   end
```

- The `work()` routine will be executed by *all* threads.
- In the C version, the brackets {} are required, unless the parallel region has only one statement.

Disclaimer: Some examples in this lecture are from the official OpenMP release documents.

Parallel Regions (cont.)

- One can set the number of threads in the parallel directive.
- One can also differentiate multiple copies of the parallel region code by using thread ids.

```c
void example() {
  #pragma omp parallel num_threads(4) private(tid)
  {
    int tid = omp_get_thread_num();
    printf("Hello world from thread = %d\n", tid);
  }
}
```

Variants of Parallel Regions — `for` Loops

Distribute a loop among the threads using the `for` directive:

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

Note: `parallel` and `for` directives can be combined into one line:

```
  ...
  #pragma omp parallel for shared(a,b)
  for (i=1; i<n; i++) {  /* i is private by default */
  ...
```


Variants of Parallel Regions — Sections

Explicitly specify code for each individual thread to execute by using the `sections` directive:

```
void section_example() {
  int tid;
  #pragma omp parallel sections private(tid)
  {
    #pragma omp section
    { tid = omp_get_thread_num();
      printf("Sec 1 executed by thread %d\n", tid); }
    #pragma omp section
    { tid = omp_get_thread_num();
      printf("Sec 2 executed by thread %d\n", tid); }
    #pragma omp section
    { tid = omp_get_thread_num();
      printf("Sec 3 executed by thread %d\n", tid); }
  }
}
```


The Single Directive

For restricting a block inside a parallel region to a single thread of execution.

```
void single_example() {
  #pragma omp parallel
  {
    #pragma omp single
    printf("Beginning work1.\n");
    work1();
    #pragma omp single
    printf("Finishing work1.\n");
    work2();
  }
}
```

A related directive:

- `master` — For restricting a block to execution by the master thread.

The Task Directive

For creating child tasks.

```
int main() {
  int i;
  #pragma omp parallel
  #pragma omp single
  for (i=0; i<4; i++) {
    #pragma omp task firstprivate(i)
    printf("Hello world! from %d (i=%d)\n",
              omp_get_thread_num(), i);
  }
}
```

- A `single` directive is needed, otherwise multiple copies of the loop nest will be executed, one by each thread.
  - The `single` directive cannot be combined with the `parallel` directive.
- The `firstprivate` clause is to ensure each thread gets the right `i` value. (See later.)

The Taskwait Directive

Causes a task to wait for all its child tasks to complete.

```
int fib(int n) {
  int a, b;
  if (n<2) return n;
  #pragma omp task shared(a) firstprivate(n)
  a = fib(n-1);
  #pragma omp task shared(b) firstprivate(n)
  b = fib(n-2);
  #pragma omp taskwait
  return a + b;
}
```

Note: A `parallel` directive is still needed in the main program:

```
int main() {
  #pragma omp parallel
  #pragma omp single
  printf("fib(10) = %d\n", fib(10));
}
```


OpenMP's Variable Clauses

- `private(`*varlist*`)` — private variables, one copy per thread
- `shared(`*varlist*`)` — shared variables, a single copy for all threads
- `default(shared|none)` — default declarations
- `firstprivate(`*varlist*`)` — private variables with inherited init values
- `lastprivate(`*varlist*`)` — private variables with values exported
- `reduction(`*op:varlist*`)` — a combination of both private and shared, for reduction operations

The lastprivate Clause

```
void a34 (int n, float *a, float *b) {
  int i;
  #pragma omp parallel for lastprivate(i)
  for (i=0; i<n-1; i++)
    a[i] = b[i] + b[i+1];
  a[i]=b[i]; /* i == n-1 here */
}
```


The reduction Clause

```
void a35(float *x, int *y, int n) {
  int i, b = 0;
  float a = 0.0;
  #pragma omp parallel for private(i) shared(x, y, n) \
                       reduction(+:a) reduction(^:b)
  for (i=0; i<n; i++) {
    a += x[i];
    b ^= y[i];
  }
}
```

Note: Long directives can be written in multiple lines.

The reduction Clause (cont.)

```
void a35_explained(float *x, int *y, int n) {
  int i, b = 0;
  float a = 0.0;
  #pragma omp parallel shared(a, b, x, y, n)
  {
    int b_p = 0;
    float a_p = 0.0;
    #pragma omp for private(i, a_p, b_p)
    for (i=0; i<n; i++) {
      a_p += x[i];
      b_p ^= y[i];
    }
    #pragma omp critical
    {
      a += a_p;
      b ^= b_p;
    }
  }
}
```


What's Wrong with This Code?

```
int main (void) {
  int a, i;
  #pragma omp parallel shared(a) private(i)
  {
    a = 0;
    #pragma omp for reduction(+:a)
    for (i = 0; i < 10; i++) {
      a += i;
    }
    printf ("Sum is %d\n", a);
  }
}
```


What's Wrong with This Code? (cont.)

```
int main (void) {
  int a, i;
  #pragma omp parallel shared(a) private(i)
  {
    #pragma omp single
    a = 0;
    #pragma omp for reduction(+:a)
    for (i = 0; i < 10; i++) {
      a += i;
    }
    #pragma omp single
    printf ("Sum is %d\n", a);
  }
}
```


OpenMP's Synchronization Directives

- `critical` — restricting a block to execution by one thread at a time
- `barrier` — synchronizing all threads
- `flush` — forcing a consistent view of memory for a thread
- `atomic` — avoiding conflicts when writing to memory
- `ordered` — specifying a block for serial execution

OpenMP Memory Model

OpenMP provides a relaxed memory consistency model, similar to *weak ordering*. It specifically allows the reordering of accesses within a thread to different variables unless they are separated by a `flush` (see below) that includes the variables.

- All OpenMP threads have access to the (shared) memory, but each thread is allowed to have its own temporary view of the memory. (Each thread also has access to a *threadprivate* memory, which is not accessible by other threads.)
- 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 Flush Directive

The OpenMP `flush` directive enforces consistency between the temporary view and memory.

```
x = 24;
...
x = 42;
/* x's value changes all happen within the thread's
   temporary view */
#pragma omp flush(x)
/* now, the shared memory has x's new value */
```

A flush of all visible variables is implied

- in a `barrier` directive
- at entry and exit from `parallel`, `critical` and `ordered` regions
- at entry and exit from combined parallel work-sharing regions
- during lock API routines

How to Write Producer-Consumer Code?

The flush operation is the only way in an OpenMP program to guarantee that a value will move between two threads.

In order to move a value from one thread to a second thread, OpenMP requires these four actions in exactly the following order:

1. the producer thread writes the value to a shared variable
2. the producer thread flushes the variable
3. the consumer thread flushes the variable
4. the consumer thread reads the variable

In short, the required sequence of actions on the shared variable is

$$\text{write}_0 - \text{flush}_0 - \text{flush}_1 - \text{read}_1$$

The order between the two flushes is the key, but it is not easy to guarantee.

Producer-Consumer (Ver. 1)

P_1:

```
A = 1;
#pragma omp flush(A)
...
```

P_2:

```
...
#pragma omp flush(A)
B = A;
```

This code does not work:

- The two flush operations are independent. There is no guarantee that the `flush(A)` op in P_2 happens *after* the `flush(A)` op in P_1.

Producer-Consumer (Ver. 2)

P_1:

```
A = 1;
#pragma omp flush(A)
...
```

P_2:

```
...
while (A == 0) {
  #pragma omp flush(A)
}
B = A;
```

This code *still* won't work:

- It assumes that P_2 knows what `A`'s old and new values are.
- The write operation in OpenMP is *not* atomic —
  - While `A` is being written to by P_1, there could be a small window during which `A`'s value is neither 0 nor 1, *i.e.* it is *undefined*.
  - If P_2 reads `A` during this window, the `while` test will fail, and `B` will get the undefined value.

Producer-Consumer (Ver. 3)

Use a separate variable `flag`, and assume its initial value is 0.

P_1:

```
A = 1;
flag = 1;
#pragma omp flush(A, flag)
...
```

P_2:

```
...
while (flag != 1) {
  #pragma omp flush(A, flag)
}
B = A;
```

This code *still* won't work:

A possible scenario:

- P_1 executes `A = 1` and `flag = 1`. (In fact, these two operations can happen in either order.)
- P_2's `while (flag != 1)` test fails; it executes `B = A`, and `B` will get an undefined value, since P_1 has not executed its `flush` operation yet.

Producer-Consumer (Ver. 4)

Finally, a correct version.

- P_1: Enforce operation order by using separate flush operations.
- P_2: Add an extra `flush(A)` after flag-checking.

P_1:

```
A = 1;
#pragma omp flush(A)
flag = 1;
#pragma omp flush(flag)
...
```

P_2:

```
...
while (flag != 1) {
  #pragma omp flush(flag)
}
#pragma omp flush(A)
B = A;
```


The Barrier Directive

A barrier operation synchronizes *all* OpenMP threads. It implies a flush operation from every thread for all shared variables.

- It is expensive!

Back to the Producer/Consumer Example:

P_1:

```
A = 1;
#pragma omp barrier
...
```

P_2:

```
...
#pragma omp barrier
B = A;
```

This code will work, but with a high cost.

OpenMP Locking Routines

```
omp_lock_t lck;           // lock var decl
omp_init_lock(&lck)       // init
omp_set_lock(&lck)        // lock
omp_unset_lock(&lck)      // unlock
omp_test_lock(&lck)       // test
omp_destroy_lock(&lck)    // destroy
```

Note: A `flush` is included in each locking routine (since v3.0).

Question: Why?

Answer: Consider the following pre-v3.0 code:

```
omp_set_lock(&lck);
local_idx = global_idx;
global_idx++;
omp_unset_lock(&lck);
```

It's incorrect:

Without a flush, memory reads and writes are performed only in a thread's local view of the memory.

OpenMP Locking Routines (cont.)

Here is an attempt to fix (in pre-v3.0):

```
omp_set_lock(&lck);
#pragma omp flush(global_idx)
local_idx = global_idx;
global_idx++;
#pragma omp flush(global_idx)
omp_unset_lock(&lck);
```

It's still incorrect:
OpenMP's lock variables are ordinary variables, not special synchronization variables as in Pthreads.

The correct version:

```
omp_set_lock(&lck);
#pragma omp flush(global_idx,lck)
local_idx = global_idx;
global_idx++;
#pragma omp flush(global_idx,lck)
omp_unset_lock(&lck);
```

The lock variable `lck` must be included in the flush set, to both prevent reordering with respect to the lock calls, and to keep the value of `global_idx` up to date.

Since v3.0: By having locking routines imply a flush, OpenMP's lock variables now behave more like special synchronization variables.

Who Owns the Lock?

In Pthreads, a lock is owned by the thread which creates it. Only the owner thread can lock and unlock the lock.

Pre-v3.0 versions of OpenMP did the same. But there are issues:

```
int main() {
  omp_lock_t lck;
  omp_init_lock(&lck);
  // a parallel region, with a set of threads
  #pragma omp parallel
  { if ((omp_get_thread_num()==XX)) // thread XX sets a lock
      omp_set_lock(&lck);
    ... }
  ...
  // a new parallel region, with a new set of threads
  #pragma omp parallel
  { if ((omp_get_thread_num()==XX)) // thread XX unlocks
       omp_unset_lock(&lck);
    ... }
}
```


Who Owns the Lock? (cont.)

Since v3.0, OpenMP switched to have locks owned by task regions.

Consequence: A lock set by one thread can be unlocked by a different thread!

```
int main() {
  omp_lock_t lck;
  omp_init_lock(&lck);
  #pragma omp parallel num_threads(2)
  {
    if ((omp_get_thread_num()) == 0) {
      omp_set_lock(&lck);
      printf("Lock set by thread 0\n");
    } else {
      omp_unset_lock(&lck);  // !!!
      omp_set_lock(&lck);
      printf("Lock re-set by thread 1\n");
    }
  }
}
```

Other Library Routines

OpenMP library routines perform the following functions:

- Affect or monitor threads and processors:

```
omp_set_num_threads(n)
omp_get_num_threads()
omp_get_max_threads()
omp_get_num_procs()
omp_get_thread_num()
omp_in_parallel()
```

- Set execution environment functions: nested parallelism, dynamic adjustment of threads.

```
omp_set_dynamic(truefalse)
omp_get_dynamic()
omp_set_nested(true|false)
omp_get_nested()
```


Runtime Environment

OpenMP provides four environment variables for controlling the execution of parallel code.

- `OMP_SCHEDULE` — Applies only to loop-related directives which have their schedule clause set to `runtime`. Its value determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

- `OMP_NUM_THREADS` — Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

- `OMP_DYNAMIC` — Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are `TRUE` or `FALSE`.
- `OMP_NESTED` — Enables or disables nested parallelism. Valid values are `TRUE` or `FALSE`.