

# COMP4300/6430: The OpenMP Programming Model

Alistair Rendell

See: [www.openmp.org](http://www.openmp.org)

*Principles of Parallel Programming*, C. Lin and L. Synder, Chapter 6

*Introduction to Parallel Computing, 2nd Ed*, A. Grama, A. Gupta, G. Karypis, V. Kumar

*Using OpenMP: Portable Shared Memory Parallel Programming*, B. Chapman, G. Jost, R. van der Pas

*High Performance Computing*, Dowd and Severance, Chapter 11

## 13.1 Shared Memory Parallel Programming

- Notes for this lecture are essentially the same as those used by COMP3320 in 2008. The discussion may be different!

## 13.2 Shared Memory Parallel Programming

- Explicit thread programming is messy
  - low-level primitives
  - originally non-standard, although better since pthreads
  - used by system programmers, but ...
    - ... application programmers have better things to do!
- Many application codes can be usefully supported by higher level constructs
  - e.g proprietary *directive* based approaches of Cray, SGI, Sun etc
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
  - standardizes the form of the proprietary directives
  - avoids the need for explicitly setting up mutexes, condition variables, data scope, and initialization

## 13.3 OpenMP

- Specifications maintained by OpenMP Architecture Review Board (ARB)
  - members include DoE, Intel, HP, Sun, IBM ... cOMPunity
- Annual user group meetings in USA (Wompat), Japan, Europe
- 1.0 (Fortran '97, C '98), 1.1, 2.0, 2.5 and 3.0 (Fortran/C/C++ May '08)
- Comprises compiler directives, library routines and environment variables
  - C directives (case sensitive)

```
#pragma omp directive_name [clause-list]
```
  - library calls begin with `omp_`

```
void omp_set_num_threads(nthreads)
```
  - environment variables begin with `OMP_`

```
setenv OMP_NUM_THREADS 4
```
- OpenMP requires compiler support
  - activated in gcc via `-fopenmp` flag

### 13.4 The Parallel Directive

- OpenMP uses a fork/join model, i.e. programs execute serially until they encounter a parallel directive:

- this creates a group of threads
- number of threads dependent on environment variable or set via function call
- main thread becomes master with thread id 0

```
1 #pragma omp parallel [clause-list]
2     /* structured block */
3
```

- Each thread executes a *structured block*

### 13.5 Parallel Clauses

Clauses are used to specify

- **Conditional Parallelization:** to determine if parallel construct results in creation of threads

```
if (scalar expression)
```

- **Degree of concurrency:** explicit specification of the number of threads created

```
num_threads (integer_expression)
```

- **Data handling:** to indicate if specific variables are local to thread (allocated on the stack), global, or "special"

```
private (variable_list)
shared (variable_list)
firstprivate (variable_list)
```

### 13.6 Compiler Translation: OpenMP to Pthreads

- OpenMP code

```
int a,b;
main(){
    // serial segment
    #pragma omp parallel num_threads(8) private(a) shared(b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

- Pthread equivalent (structured block is *outlined*)

```
int a, b;
main(){
    //serial segment
    for (i=0; i<8; i++)pthread_create(....., internal_thunk,...);
    for (i=0; i<8; i++)pthread_join(.....);
    //rest of serial segment
}
void *internal_thunk(void *packaged_argument){
    int a;
    // parallel segment
}
```

### 13.7 Parallel Directive Examples

```
#pragma omp parallel if (is_parallel == 1) num_threads(8) \
    private(a) shared(b) firstprivate(c)
```

- If value of variable `is_parallel` is one, eight threads are used
- Each thread has private copy of `a` and `c`, but shares a single copy of `b`
- Value of each private copy of `c` is initialized to value of `c` before parallel region

```
#pragma omp parallel reduction(+:sum) num_threads(8) default(private)
```

- Eight threads get a copy of variable `sum`
- When threads exit the values of these local copies are accumulated into the `sum` variable on the master thread
  - other reduction operations include `*`, `-`, `&`, `|`, `^`, `&&` and `||`
- All variables are private unless otherwise specified

### 13.8 Example: Computing Pi

- Compute  $\pi$  by generating random numbers in square with side length of 2 centered at (0,0) and counting numbers that fall within a circle of radius 1
  - area of square = 4, area of circle =  $\pi r^2 = \pi$
  - ratio of points in circle to outside approaches  $\pi/4$

```
#pragma omp parallel default(private) shared(npoints) \
    reduction(+: sum) sum_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints/num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++){
        rand_x = (double) (rand_range(&seed,-1,1));
        rand_y = (double) (rand_range(&seed,-1,1));
        if ((rand_x * rand_x + rand_y * rand_y) <= 1.0) sum++;
    }
}
```

- OpenMP code very simple - try writing equivalent pthread code

### 13.10 Assigning Iterations to Threads

- The schedule clause of the for directive assigns iterations to threads

```
schedule(scheduling_clause[,parameter])
```
- `schedule(static[,chunk-size])`
  - splits iteration space into chunks of size `chunk-size` and allocates to threads in round-robin fashion
  - no specification implies number of chunks equals number of threads
- `schedule(dynamic[,chunk-size])`
  - iteration space split into `chunk-size` blocks that are scheduled dynamically
- `schedule(guided[,chunk-size])`
  - chunk size decreases exponentially with iterations to minimum of `chunk-size`
- `schedule(runtime)`
  - determine scheduling based on setting of OMP\_SCHEDULE environment variable

### 13.9 The for Worksharing Directive

- Used in conjunction with `parallel` directive to partition the for loop immediately afterwards

```
#pragma omp parallel default(private) shared(npoints) \
    reduction(+: sum) sum_threads(8)
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++){
        rand_x = (double) (rand_range(&seed,-1,1));
        rand_y = (double) (rand_range(&seed,-1,1));
        if ((rand_x * rand_x + rand_y * rand_y) <= 1.0) sum++;
    }
}
```

- loop index (`i`) is assumed to be private
- only two directives plus sequential code (code is easy to read/maintain)
- Implicit synchronization at end of loop
  - can add `nowait` clause to prevent synchronization

### 13.11 Sections

- Consider partitioning of fixed number of tasks across threads
  - much less common than for loop partitioning
  - explicit programming naturally limits number of threads (scalability)

```
#pragma omp sections
{
    #pragma omp section
    {
        taskA()
    }
    #pragma omp section
    {
        taskB()
    }
}
```

- Separate threads will run `taskA` and `taskB`
- Illegal to branch in or out of section blocks

### 13.12 Nesting Parallel Directives

- What happens for nested for loops

```
#pragma omp parallel for num_threads(2)
for (i = 0; i < Ni; i++){
    #pragma omp parallel for num_threads(2)
    for (j = 0; j < Nj; j++){
```

- By default inner loop is serialized and run by one thread
- To enable multiple threads in nested parallel loops requires environment variable OMP\_NESTED to be TRUE
- Note - the use of synchronization constructions in nested parallel sections requires care (see OpenMP specs)!

### 13.14 Synchronization#2

- **Ordered:** some operations within a for loop must be performed as if it were done so in sequential order

```
cumul_sum[0] = list[0];
#pragma omp parallel for shared (cumul_sum, list, n)
for (i=1; i<n; i++)
{
    /* other processing on list[i] if required*/
    #pragma omp ordered
    {
        cumul_sum[i] = cumul_sum[i-1] + list[i];
    }
}
```

- **Flush:** ensures a consistent view of memory (see following lectures on memory consistency models)
  - that variables have been flushed from registers into memory

```
#pragma omp flush[(list)]
```

### 13.13 Synchronization#1

- **Barrier:** threads wait until they have all reached this point

```
#pragma omp barrier
```

- **Single:** following block executed only by first thread to reach this point
  - others wait at end of structured block unless nowait clause used

```
#pragma omp single [clause-list]
/* structured block */
```

- **Master:** only master executes following block, other threads do NOT wait

```
#pragma omp master
/* structured block */
```

- **Critical Section:** only one thread is ever in the named critical section

```
#pragma omp critical [(name)]
/* structured block */
```

- **Atomic:** memory location updated in following instruction is done so in an atomic fashion
  - can achieve same effect using critical sections

### 13.15 Data Handling

- **private:** an uninitialized local copy of variable made for each thread
- **shared:** variables shared between threads
- **firstprivate:** make a local copy of an existing variable and assign it same value
  - often better than multiple reads to shared variable
- **lastprivate:** copies back to master value from thread executing the equivalent of the last loop iteration if executed serially
- **threadprivate:** creates private variables but they persist between multiple parallel regions maintaining their values
- **copyin:** like first private but for threadprivate variables

### 13.16 Library Functions#1

- Defined in header file

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

- Controlling threads and processors

```
void omp_set_num_threads(int num_threads)
int omp_get_num_threads()
int omp_get_max_threads()
int omp_get_thread_num()
int omp_get_num_procs()
int omp_in_parallel()
```

- Controlling thread creation

```
void omp_set_dynamic(int dynamic_threads)
int omp_get_dynamic()
void omp_set_nested(int nested)
int omp_get_nested()
```

### 13.17 Library Functions#2

- Mutual exclusion

```
void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)
void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)
int omp_test_lock(omp_lock_t *lock)
```

- Nested mutual exclusion

```
void omp_init_nest_lock(omp_nest_lock_t *lock)
void omp_destroy_nest_lock(omp_nest_lock_t *lock)
void omp_set_nest_lock(omp_nest_lock_t *lock)
void omp_unset_nest_lock(omp_nest_lock_t *lock)
int omp_test_nest_lock(omp_nest_lock_t *lock)
```

### 13.18 OpenMP Environment Variables

- OMP\_NUM\_THREADS: default number of threads entering parallel region
- OMP\_DYNAMIC: if TRUE it permits the number of threads to change during execution
- OMP\_NESTED: if TRUE it permits nested parallel regions
- OMP\_SCHEDULE: determines scheduling for loops that are defined to have runtime scheduling

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

### 13.19 OpenMP and Pthreads

- OpenMP removes the need for a programmer to initialize task attributes, set up arguments to threads, partition iteration spaces etc
- OpenMP code can closely resemble serial code
- OpenMP is particularly useful for static or regular problems
- OpenMP users are hostage to availability of an OpenMP compiler
  - performance heavily dependent on quality of compiler
- Pthreads data exchange is more apparent so false sharing and contention less likely
- Pthreads has a richer API that is much more flexible, e.g. condition waits, locks of different types etc
- Pthreads is library based

*Must balance above before deciding on parallel model*