



Fueling the source with OpenMP

Jonathan Chen

Resources

- OpenMP.org (<http://www.openmp.org>)
 - Documentation (<http://www.openmp.org/drupal/mp-documents/spec25.pdf>)
- Lawrence Livermore National Laboratory (<http://www.llnl.gov>)
 - Tutorial (<http://www.llnl.gov/computing/tutorials/openMP/>)
- Slides available on www.jonathanchen.com

Directives

- General format:
`#pragma omp directive-name [clause, ...] newline`
- `#pragma omp`
 - Required for all OpenMP C/C++ directives
- *directive-name*
 - A valid OpenMP directive. Must appear after pragma and before clauses
- *[clause, ...]*
 - Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
- *newline*
 - Required. Followed by structured block

Directives

■ General Rules:

- Case sensitive
- Follow conventions of C/C++ standards for compile directives
- Only one directive name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block
- Long directive lines can be continued by succeeding lines by escaping the newline character with a backslash ('\ ') at the end of a directive line

Syntax – **atomic**

- **#pragma omp atomic 'statement'**
- **'statement'** can be:
 - **x bin_op= expr**
 - **bin_op**: {+ * - / & ^ | << >>}
 - **expr**: an expression of scalar type that does not reference **x**
 - **x++**
 - **++x**
 - **x--**
 - **--x**
- Indicates that the specified memory location must be updated atomically and not be exposed to multiple, simultaneous writing threads.

Syntax – `parallel`

- `#pragma omp parallel 'clause'`
- `'clause'` can be:
 - `if(exp)`
 - `private(list)`
 - `firstprivate(list)`
 - `num_threads(int_exp)`
 - `shared(list)`
 - `default(shared|none)`
 - `copyin(list)`
 - `reduction(operator: list)`
- Indicates that the code section is to be parallelized

Syntax – **for**

- **#pragma omp for 'clause'**
- **'clause'** can be:
 - `private(list)`
 - `firstprivate(list)`
 - `lastprivate(list)`
 - `reduction(operator: list)`
 - `ordered`
 - `schedule(type)`
 - `Nowait`
- Compiler distributes loop iterations within team of threads

Syntax – **ordered**

- **#pragma omp ordered**

- Indicates that the code section must be executed in sequential order

Syntax – `parallel for`

- `#pragma omp parallel for 'clause'`
- `'clause'` can be:
 - `if(exp)`
 - `private(list)`
 - `firstprivate(list)`
 - `lastprivate(list)`
 - `num_threads(int_exp)`
 - `shared(list)`
 - `default(shared|none)`
 - `copyin(list)`
 - `reduction(operator: list)`
 - `ordered`
 - `schedule(type)`
- Combines the `omp parallel` and `omp for` directives

Syntax – **sections**

- `#pragma omp sections 'clause'`
- `'clause'` can be:
 - `private(list)`
 - `firstprivate(list)`
 - `lastprivate(list)`
 - `reduction(operator: list)`
 - `nowait`
- In structured block following the directive, an `omp section` directive will indicate that the following sub-block can be distributed for parallel execution.

Syntax – `parallel sections`

- `#pragma omp parallel sections 'clause'`
- `'clause'` can be:
 - `if(exp)`
 - `private(list)`
 - `firstprivate(list)`
 - `lastprivate(list)`
 - `shared(list)`
 - `default(shared|none)`
 - `copyin(list)`
 - `reduction(operator: list)`
 - `Nowait`
- Combines the `omp parallel` and `omp sections` directives

Syntax – **single**

- `#pragma omp single 'clause'`
- `'clause'` can be:
 - `private(list)`
 - `copyprivate(list)`
 - `firstprivate(list)`
 - `Nowait`
- Indicates that the code section must only be run by a single available thread.

Syntax – `master`

- `#pragma omp master`

- Indicates that the code section must only be run by master thread

Syntax – `critical`

■ `#pragma omp critical`

- Indicates that the code section can only be executed by a single thread at any given time

Syntax – **barrier**

■ **#pragma omp barrier**

- Identifies a synchronization point at which threads in a parallel region will not continue until all other threads in that section reach the same spot
- Explicit for a few directives
 - **omp parallel**
 - **omp for**

Syntax – `flush`

■ `#pragma omp flush (list)`

- Identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory. If no list is given, then all shared objects are synchronized.
- `flush` is implicit for the following directives:
 - `omp barrier`
 - Entrance and exit of `omp critical`
 - Exit of `omp parallel`
 - Exit of `omp for`
 - Exit of `omp sections`
 - Exit of `omp single`

Syntax – `threadprivate`

- `#pragma omp threadprivate (var)`
 - `omp threadprivate` makes the variable private to a thread

OpenMP Functions

- `void omp_set_num_threads (int)`
 - Called inside serial section. Can exceed available processors
- `int omp_get_num_threads (void)`
 - Returns number of active threads
- `int omp_get_max_threads (void)`
 - Returns max system allowed threads
- `int omp_get_thread_num (void)`
 - Returns thread's ID number (ranges from 0 to $t-1$)
- `int omp_get_num_procs (void)`
 - Returns number of processors available to the program

OpenMP Functions

- `int omp_in_parallel (void)`
 - Returns **1** if called inside a parallel block
- `void omp_set_dynamic (int)`
 - Enable (**1**) or disable (**0**) dynamic threads
- `int omp_get_dynamic (void)`
 - Returns **1** if dynamic threads enabled
- `void omp_set_nested (int)`
 - Enable (**1**) or disable (**0**) nested parallelism
- `int omp_get_nested (void)`
 - Returns **1** if nested parallelism enabled (default **0**)

OpenMP Functions

- `void omp_init_lock(omp_lock_t*)`
 - Initializes a lock associated with the lock variable
- `void omp_destroy_lock(omp_lock_t*)`
 - Disassociates the given lock variable from any locks
- `void omp_set_lock(omp_lock_t*)`
 - Wait until specified lock is available
- `void omp_unset_lock(omp_lock_t*)`
 - Releases the lock from executing routine
- `int omp_test_lock(omp_lock_t*)`
 - Attempts to set a lock, but does not wait if the lock is unavailable
 - Returns non-zero value on success

OpenMP Functions

- **double omp_get_wtime(void)**

- Returns the number of elapsed seconds since some point in the past

- **double omp_get_wtick(void)**

- Returns the number of elapsed seconds between successive clock ticks

Environment Variables

■ OMP_SCHEDULE

- Applies only to parallel for directives with their schedule clause set to **RUNTIME**
- Determines how iterations of the loop are scheduled

■ OMP_NUM_THREADS

- Maximum number of threads to use for execution

■ OMP_DYNAMIC

- Enable (**1**) or disable (**0**) dynamic adjustment of threads available for execution

■ OMP_NESTED

- Enable (**1**) or disable (**0**) nested parallelism

Clause - `list`

■ `list`

- `private(list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `shared(list)`
- `copyin(list)`

■ List of variables

Clause – operator: list

- *operator: list*

- `reduction(operator: list)`

- **Operators includes:**

- +

- *

- &

- |

- ^

- &&

- ||

Clause – `schedule (type , size)`

- `schedule (type , size)`
 - `schedule (static)`
 - Allocates n / t contiguous iterations to each thread
 - `schedule (static , C)`
 - Allocates C contiguous iterations to each thread
 - `schedule (dynamic)`
 - Allocates 1 iteration at a time, dynamically
 - `schedule (dynamic , C)`
 - Allocates C iterations at a time, dynamically
 - `schedule (guided , C)`
 - Allocates decreasingly large iterations to each thread until size reaches C
 - `schedule (guided)`
 - Same as `(guided , C)`, with $C = 1$
 - `schedule (runtime)`
 - Based on environment variable `OMP_SCHEDULE`

Examples – Reduction

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int    i, n;
    float a[100], b[100], sum;

    n = 100; /* Some initializations */
    for (i=0; i < n; i++)
        a[i] = b[i] = i * 1.0;
    sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
        for (i=0; i < n; i++)
            sum = sum + (a[i] * b[i]);
    printf("    Sum = %f\n",sum);
}
```

Examples – OpenMP Functions

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

int main (int argc, char *argv[]){
int nthreads, tid, procs, maxt, inpar,
    dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid) {
    /* Obtain thread number */
    tid = omp_get_thread_num();

    /* Only master thread does this */
    if (tid == 0) {
        printf("Thread %d getting info...\n", tid);
```

```
/* Get environment information */
procs = omp_get_num_procs();
nthreads = omp_get_num_threads();
maxt = omp_get_max_threads();
inpar = omp_in_parallel();
dynamic = omp_get_dynamic();
nested = omp_get_nested();

/* Print environment information */
printf("Number of processors = %d\n", procs);
printf("Number of threads = %d\n", nthreads);
printf("Max threads = %d\n", maxt);
printf("In parallel? = %d\n", inpar);
printf("Dynamic threads? = %d\n", dynamic);
printf("Nested parallelism? = %d\n", nested);
    }
} /* Done */
}
```