OpenMP Reference Sheet for C/C++

Constructs

<parallelize a for loop by breaking apart iterations into chunks>
#pragma omp parallel for [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr), ordered, schedule(type[,chunkSize])]
<A,B,C such that total iterations known at start of loop>
for(A=C;A<B;A++) {
    <your code here>

    <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
    #pragma omp ordered {
        <your code here>
    }
}

<parallelized sections of code with each section operating in one thread>
#pragma omp parallel sections [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    #pragma omp section {
        <your code here>
    }
    #pragma omp section {
        <your code here>
    }
    ....
}

<grand parallelization region with optional work-sharing constructs defining more specific splitting of work and variables amongst threads. You may use work-sharing constructs without a grand parallelization region, but it will have no effect (sometimes useful if you are making OpenMP'able functions but want to leave the creation of threads to the user of those functions)>
#pragma omp parallel [shared(vars), private(vars), firstprivate(vars), lastprivate(vars), default(private|shared|none), reduction(op:vars), copyin(vars), if(expr)] {
    <the work-sharing constructs below can appear in any order, are optional, and can be used multiple times. Note that no new threads will be created by the constructs. They reuse the ones created by the above parallel construct.>

    <your code here (will be executed by all threads)>

    <parallelize a for loop by breaking apart iterations into chunks>
    #pragma omp for [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), ordered, schedule(type[,chunkSize]), nowait]
    <A,B,C such that total iterations known at start of loop>
    for(A=C;A<B;A++) {
        <your code here>

        <force ordered execution of part of the code. A=C will be guaranteed to execute before A=C+1>
        #pragma omp ordered {
            <your code here>
        }
    }

    <parallelized sections of code with each section operating in one thread>
    #pragma omp sections [private(vars), firstprivate(vars), lastprivate(vars), reduction(op:vars), nowait] {
        #pragma omp section {
            <your code here>
        }
        #pragma omp section {
            <your code here>
        }
        ....
    }

    <only one thread will execute the following. NOT always by the master thread>
    #pragma omp single {
        <your code here (only executed once)>
    }
}

Directives

shared(vars) *<share the same variables between all the threads>*
private(vars) *<each thread gets a private copy of variables. Note that other than the master thread, which uses the original, these variables are not initialized to anything.>*
firstprivate(vars) *<like private, but the variables do get copies of their master thread values>*
lastprivate(vars) *<copy back the last iteration (in a for loop) or the last section (in a sections) variables to the master thread copy (so it will persist even after the parallelization ends)>*
default(private|shared|none) *<set the default behavior of variables in the parallelization construct. shared is the default setting, so only the private and none setting have effects. none forces the user to specify the behavior of variables. Note that even with shared, the iterator variable in for loops still is private by necessity >*
reduction(op:vars) *<vars are treated as private and the specified operation(op, which can be +,*,-,&,|,&,&&,||) is performed using the private copies in each thread. The master thread copy (which will persist) is updated with the final value.>*

copyin(vars) *<used to perform the copying of threadprivate vars to the other threads. Similar to firstprivate for private vars.>*
if(expr) *<parallelization will only occur if expr evaluates to true.>*
schedule(type [,chunkSize]) *<thread scheduling model>*

type	*chunkSize*
static	*number of iterations per thread pre-assigned at beginning of loop (typical default is number of processors)*
dynamic	*number of iterations to allocate to a thread when available (typical default is 1)*
guided	*highly dependent on specific implementation of OpenMP*

nowait *<remove the implicit barrier which forces all threads to finish before continuation in the construct>*

Synchronization/Locking Constructs *<May be used almost anywhere, but will only have effects within parallelization constructs.>*

<only the master thread will execute the following. Sometimes useful for special handling of variables which will persist after the parallelization.>
#pragma omp master {
<your code here (only executed once and by the master thread).
}

<mutex lock the region. name allows the creation of unique mutex locks.>
#pragma omp critical [(name)] {
<your code here (only one thread allowed in at a time)>
}

<force all threads to complete their operations before continuing>
#pragma omp barrier

<like critical, but only works for simple operations and structures contained in one line of code>
#pragma omp atomic
<simple code operation, ex. a += 3; Typical supported operations are ++,--,+,,-,/,&,^,<<,>>,| on primitive data types>*

<force a register flush of the variables so all threads see the same memory>
#pragma omp flush[(vars)]

<applies the private clause to the vars of any future parallelize constructs encountered (a convenience routine)>
#pragma omp threadprivate(vars)

Function Based Locking *< nest versions allow recursive locking>*

void **omp_init_[nest_]lock**(omp_lock_t*) *<make a generic mutex lock>*
void **omp_destroy_[nest_]lock**(omp_lock_t*) *<destroy a generic mutex lock>*
void **omp_set_[nest_]lock**(omp_lock_t*) *<block until mutex lock obtained>*
void **omp_unset_[nest_]lock**(omp_lock_t*) *<unlock the mutex lock>*
int **omp_test_[nest_]lock**(omp_lock_t*) *<is lock currently locked by somebody>*

Settings and Control

int **omp_get_num_threads**() *<returns the number of threads used for the parallel region in which the function was called>*
int **omp_get_thread_num**() *<get the unique thread number used to handle this iteration/section of a parallel construct. You may break up algorithms into parts based on this number.>*
int **omp_in_parallel**() *<are you in a parallel construct>*
int **omp_get_max_threads**() *<get number of threads OpenMP can make>*
int **omp_get_num_procs**() *<get number of processors on this system>*
int **omp_get_dynamic**() *<is dynamic scheduling allowed>*
int **omp_get_nested**() *<is nested parallelism allowed>*
double **omp_get_wtime**() *<returns time (in seconds) of the system clock>*
double **omp_get_wtick**() *<number of seconds between ticks on the system clock>*
void **omp_set_num_threads**(int) *<set number of threads OpenMP can make>*
void **omp_set_dynamic**(int) *<allow dynamic scheduling (note this does not make dynamic scheduling the default)>*
void **omp_set_nested**(int) *<allow nested parallelism; Parallel constructs within other parallel constructs can make new threads (note this tends to be unimplemented in many OpenMP implementations)>*

<env vars- implementation dependent, but here are some common ones>
OMP_NUM_THREADS "number" *<maximum number of threads to use>*
OMP_SCHEDULE "type,chunkSize" *<default #pragma omp schedule settings>*

Legend

vars is a comma separated list of variables
[optional parameters and directives]
<descriptions, comments, suggestions>
.... above directive can be used multiple times
For mistakes, suggestions, and comments please email e_berta@plutospin.com