OpenMP

The following provides a summary of the LLNL OpenMP tutorial by Blaise Barney *https://computing.llnl.gov/tutorials/openMP/.*
OpenMP is an application program interface to enable multi-threaded, shared memory parallelism.
Basic execution model:

- All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
- FORK: the master thread then creates a team of parallel threads.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

The three primary API components are compiler directives, runtime library routines and environment variables.

- **Compiler Directives:**
  Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise!
  The main uses of OpenMP compiler directives are:
  - Spawning a parallel region
  - Dividing blocks of code among threads
  - Distributing loop iterations between threads
  - Serializing sections of code
  - Synchronization of work among threads
- **Run-time Library Routines:**
  Note that for C/C++, you usually need to include the <omp.h> header file! In contrast to Fortran routines C/C++ routines are case sensitive!
  Their main purpose is to enable:
  - Setting and querying the number of threads
  - Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
  - Setting and querying the dynamic threads feature

  - Querying if in a parallel region, and at what level
  - Setting and querying nested parallelism
  - Setting, initializing and terminating locks and nested locks
  - Querying wall clock time and resolution

- **Environment Variables:**
  Setting OpenMP environment variables depends upon which shell you use, e.g. sh/bash: `export OMP_NUM_THREADS=8`.
  Environment variables allow to control things as:
  - Setting the number of threads
  - Specifying how loop iterations are divided
  - Binding threads to processors
  - Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
  - Enabling/disabling dynamic threads
  - Setting thread stack size
  - Setting thread wait policy

Compiler Directives

Basic rules and tips to keep in mind:

- Only one directive-name may be specified per directive
- Long directive lines can be continued on succeeding lines by escaping the newline character with a backslash (\) at the end of a directive line.
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and **has thread number 0 within that team**.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- **There is an implied barrier at the end of a parallel region. Only the master thread continues execution past this point.** This behavior can be changed with the `nowait` clause.
- **If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.**

- A parallel region must be a structured block **that does not span multiple routines or code files**.
- **It is illegal to branch (goto) into or out of a parallel region!**

Listing 1: PARALLEL Region Construct

```
#pragma omp parallel [clause ...]  newline
                     if (scalar_expression)
                     private (list)
                     shared (list)
                     default (shared | none)
                     firstprivate (list)
                     reduction (operator: list)
                     copyin (list)
                     num_threads (integer-expression)


   structured_block
--------------------------EXAMPLE-------------------------

#include <omp.h>

 main(int argc, char *argv[]) {

 int nthreads, tid;

 /* Fork a team of threads with each thread having a
    ↪ private tid variable */
 #pragma omp parallel private(tid){

   /* Obtain and print thread id */
   tid = omp_get_thread_num();
   printf("Hello World from thread = %d\n", tid);

   /* Only master thread does this */
   if (tid == 0){
     nthreads = omp_get_num_threads();
     printf("Number of threads = %d\n", nthreads);
     }

   }  /* All threads join master thread and terminate */

 }
```

#PRAGMA OMP FOR

The for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team.
This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor!

Listing 2: FOR Directive

```
#pragma omp for [clause ...]  newline
                schedule (type [,chunk])
                ordered
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                nowait

   for_loop
--------------------------EXAMPLE----------------------
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main(int argc, char *argv[]){

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i){

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }   /* end of parallel region */

}
```

NO WAIT: If specified, then threads do not synchronize at the end of the parallel loop.

SCHEDULES:
A schedules describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

- **STATIC:** Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.
- **DYNAMIC:** Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
- **GUIDED:** Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:
  `number_of_iterations / number_of_threads`
  Subsequent blocks are proportional to:
  `number_of_iterations_remaining / number_of_threads`

  The chunk parameter defines the minimum block size. The default chunk size is 1.
- **RUNTIME:** The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- **AUTO:** The scheduling decision is delegated to the compiler and/or runtime system.

The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads!

#PRAGMA OMP PARALLEL FOR

This directive behaves mostly identically to an individual PARALLEL directive being immediately followed by an according directive.

Listing 3: PARALLEL FOR Directive

```
1  #include <omp.h>
2  #define N       1000
3  #define CHUNKSIZE   100
4
5  main(int argc, char *argv[]) {
6
7  int i, chunk;
8  float a[N], b[N], c[N];
9
10 /* Some initializations */
11 for (i=0; i < N; i++)
12   a[i] = b[i] = i * 1.0;
13 chunk = CHUNKSIZE;
14
15 #pragma omp parallel for \
16   shared(a,b,c,chunk) private(i) \
17   schedule(static,chunk)
18   for (i=0; i < n; i++)
19     c[i] = a[i] + b[i];
20 }
```

#PRAGMA OMP SINGLE

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
May be useful when dealing with sections of code that are not thread safe (such as I/O).

Listing 4: SINGLE Directive

```
1  #pragma omp single [clause ...]  newline
2                     private (list)
3                     firstprivate (list)
4                     nowait
5
6       structured_block
```

Synchronization Constructs

- **MASTER Directive**: specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section. There is no implied barrier associated with this directive. `#pragma omp master newline`
- **CRITICAL Directive**: specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, **it will block until the first thread exits that critical region!** The optional name enables multiple different critical regions to exist.
  `#pragma omp critical [ name ] newline`
- **BARRIER Directive**: synchronizes all threads in the team. When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier. `#pragma omp barrier newline`
- **ATOMIC Directive**: specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-critical section. The directive applies only to **a single, immediately following statement**. `#pragma omp atomic newline`

Data Scope Attribute Clauses

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL,for,...) to control the scoping of enclosed variables. They define which variables will be visible to all threads in the parallel regions and which variables will be privately allocated to all threads.
Important: Please consult the latest OpenMP specs for important details and discussion on this topic.

- **PRIVATE**: declares variables in its list to be private to each thread. A new object of the same type is declared once for each thread in the team. Variables declared private should be assumed to be **uninitialized** for each thread. `private (list)`
- **FIRSTPRIVATE**: combines the behavior of the private clause with **automatic initialization** of the variables in its list. `firstprivate (list)`
- **LASTPRIVATE**: combines the behavior of the private clause with **a copy from the last loop iteration or section to the original variable object**. `lastprivate (list)`
- **SHARED**: declares variables in its list to be shared among all threads in the team. A shared variable exists in only one memory location and all threads can

read or write to that address. It is the programmer's responsibility to ensure that multiple threads properly access shared variables (such as via critical sections).
`shared (list)`

- **DEFAULT**: allows the user to specify a default scope for all variables in the lexical extent of any parallel region.
- **REDUCTION**: performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
  `reduction (operator: list)`
  NOTE: Variables in the list must be **named scalar variables**. They can not be array or structure type variables. They must also be **declared shared** in the enclosing context. Reduction operations may not be associative for real numbers.

Listing 5: REDUCTION CLAUSE

```
#include <omp.h>

main(int argc, char *argv[]){
 int   i, n, chunk;
 float a[100], b[100], result;

 /* Some initializations */
 n = 100;
 chunk = 10;
 result = 0.0;
 for (i=0; i < n; i++) {
   a[i] = i * 1.0;
   b[i] = i * 2.0;
   }

 #pragma omp parallel for      \
   default(shared) private(i)  \
   schedule(static,chunk)      \
   reduction(+:result)

   for (i=0; i < n; i++){
     result = result + (a[i] * b[i]);
   }

 printf("Final result= %f\n",result);

 }
```