OpenMP Syntax

Compiling and running OpenMP programs

- C/C++:

  gcc –fopenmp –o prog prog.c -lgomp

  g++ –fopenmp –o prog prog.C -lgomp

- One can compile the sequential version – that is ignore the OpenMP pragmas with

  gcc –o prog prog.c -lgomp

  g++ –o prog prog.C -lgomp

- Only certain to work correctly, if there are no API calls

Running

- Standard environment variable determines the number of threads:
  - tcsh

    setenv OMP_NUM_THREADS 8
  - sh/bash

    export OMP_NUM_THREADS=8
- Run program and get wall-time:

  time prog

The OpenMP Programming Model

- Number of threads are determined
  - by system default – usually a thread per CPU or core
  - using the environment variable **OMP_NUM_THREADS**
  - from within the program by using function call
- The programmer is responsible for managing synchronization and data dependencies!

The Parallel Region

- Each thread has a thread number, which is an integer from 0 (the master thread) to the number of threads minus one.
  - Can be determined by a call to **omp_get_thread_num()**
- Threads can execute different paths of statements in the parallel region
  - Typically achieved by branching on the thread number:

```
#pragma omp parallel
{
        myid= omp_get_thread_num();
        if (myid == 0)
                do_something();
        else
                do_something_else(myid);
}
```


Parallel Regions: Execution Modes

- "Dynamic mode" (the default)
  - The number of threads used in a parallel region can vary, under control of the operating system, from one parallel region to the next.
  - Setting the number of threads just sets the maximum number of threads; you might get fewer!
- "Static mode"
  - The number of threads is fixed by the programmer; you must always get this many (or else fail to run).
  - Parallel regions may be nested, but a compiler may choose to "serialize" the inner parallel region, i.e., run it on a single thread.
- Execution mode is controlled by

  The environment variable **OMP_DYNAMIC**

  The OMP function **omp_set_dynamic()**

Parallel Regions

A team of threads is created at run time for a parallel region

A nested parallel region is allowed, but may contain a team of one thread

Nested parallelism is enabled with setenv OMP_NESTED TRUE

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
- Implicit 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
- Examples "static,2" or "dynamic,5"

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

OpenMP Directive Clauses

- **shared(var1,var2,…)**
  Variables to be shared among all threads (threads access same memory locations).
- **private(var1,var2,…)**
  Each thread has its own copy of the variables for the duration of the parallel code.
- **firstprivate(var1,var2,…)**
  Private variables that are initialized when parallel code is entered.
- **lastprivate(var1,var2,…)**
  Private variables that save their values at the last (serial) iteration.
- **if(expression)**
  Only parallelize if expressionis true.
- **default(shared|private|none)**
  Specifies default scoping for variables in parallel code.
- **schedule(type [,chunk])**
  Controls how loop iterations are distributed among threads.
- **reduction(operator|intrinsic:var1,var2…)**
  Ensures that a reduction operation (e.g., a global sum) is performed safely.

Clause - list

list

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

List of variables

firstprivate Clause

Variables initialized from shared variable

Variables are private (local to each thread), but are initialized to the value in the preceding serial code.

C++ objects are copy-constructed

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
      if ((I%2)==0) incr++;
      A(I)=incr;
}
```

lastprivate Clause

Variables are private (local to each thread)

The value corresponding to the last iteration of the loop (in serial mode) is saved following the parallel construct.

Variables can update shared variable using value from last iteration

C++ objects are

updated as if by

assignment

```
void sq2(int n,
         double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```


threadprivate Clause

- threadprivate applies to file-scope and static variables
- Makes them private to individual threads, but global within each thread.

Syntax:

- **#pragma omp threadprivate(var1,var2,…)**
- The *threadprivate* directive must appear after the declarations of the specified variables but before any references to them, and must itself be at file (or namespace) scope.
- Threadprivate variables can only appear in the *copyin*, *schedule* and *if* clauses.
- For threadprivate variables to persist over several parallel regions, must use static execution mode and the same number of threads in every region.
  - That is to guarantee persistence, the dynamic threads feature must be disabled

setenv OMP_DYNAMIC FALSE

- Use copyin to initialize from master thread

threadprivate Clause

```
struct Astruct A;
#pragma omp threadprivate(A)
…

#pragma omp parallel copyin(A)
  do_something_to(&A);
…

#pragma omp parallel
  do_something_else_to(&A);
```

Private copies of “A” persist between regions

Initializing threadprivate Variables – The copyin Clause

- Causes threadprivate variables to be given the master thread’s values at the onset of parallel code.

C syntax:

copyin(var1,var2,…)

- Note: copyin is also a valid clause for *parallel for* loops and the *parallel sections* construct

copyprivate

- provides a method to use a private variable to broadcast a value from one thread executing a parallel region to the other threads executing it
- if used with the *single* construct the broadcast is completed before any of the threads leave the barrier at the end

The `if` clause

- Don't want to parallelize a loop if the overhead outweighs the speedup

if(expression)

```
#pragma omp parallel for if(n.ge.2000)
        for(i=0;i<n;i++)  a(i) = b(i)*c + d(i);
```

The default clause

```
#pragma omp parallel default(private) shared(a)
{       myid=omp_get_thread_num();
        x = work(myid);
        if (x < 1.0)  a(myid) = x;
}
```

- Default clause automatically makes x and myid private

Clause – operator: list

operator: list

- **reduction(*operator: list*)**

Operators includes:

- **+**
- *****
- **&**
- **|**
- **^**
- **&&**
- **||**

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);
}
```


Clause – schedule(type, size)

schedule(type, size)

- **schedule(static)**
  - **Allocates** ceiling(N/t) **contiguous iterations to each thread,** where N is the number of iterations and t is the number of threads
- **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. W**hen a thread is ready to receive new work, it is assigned the next pending chunk of size c
- **schedule(guided, *C*)**
  - **Allocates decreasingly large iterations to each thread until size reaches *C*. *A*** variant of dynamic scheduling in which the size of the chunk decreases exponentially from chunk to *C*. Default value for chunk is ceiling(N/p)
- **schedule(guided)**
  - **Same as (guided, *C*), with *C* = 1**
- **schedule(runtime)**
  - Indicates that the schedule type and chunk are specified by **environment variable OMP_SCHEDULE**
  - Example of run-time specified scheduling

    setenv OMP_SCHEDULE “dynamic,2”

Scheduling defaults

- If the schedule clause is missing, an implementation dependent schedule is selected. Gomp default is the static schedule
- Static scheduling has low overhead and provides better data locality
- Dynamic and guided scheduling may provide better load balancing

Static Scheduling: Doing It By Hand

Must know:

- Number of threads (Nthrds)
- Each thread ID number (id)

Compute start and end iterations:

```
#pragma omp parallel
{
        int i, istart, iend;
        istart = id * N / Nthrds;
        iend = (id+1) * N / Nthrds;
        for(i=istart;i<iend;i++){
           c[i] = a[i] + b[i];}
}
```

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 */
}
```


Advanced Synchronization: Lock Functions (C/C++)

- **void omp_init_lock(omp_lock_t *lock);**
  - Initializes the lock associated with the parameter lock
- **void omp_destroy_lock(omp_lock_t *lock);**
  - Ensures the lock variable **lock** is uninitialized
- **void omp_set_lock(omp_lock_t *lock);**
  - Blocks the thread executing the function until lock is available, then sets the lock and proceeds.
- **void omp_unset_lock(omp_lock_t *lock);**
  - Releases ownership of **lock**
- **integer omp_test_lock(omp_lock_t *lock);**
  - Tries to set the lock, but does not block the thread from executing.
  - Returns non-zero (“true”) if the lock was successfully set.
- Must **#include <omp.h>**

```
#include <omp.h>
void main()
{
        omp_lock_t lock;
        int myid;
        omp_init_lock(&lock);
        #pragma omp parallel shared(lock) private(myid)
        {
                myid = omp_get_thread_num();
                omp_set_lock(&lock);
                printf("Hello from thread %d\n", myid);
                omp_unset_lock(&lock);
                while (! omp_test_lock(&lock)) {
                        skip(myid);
                }
                do_work(myid);
                omp_unset_lock(&lock);
        }
        omp_destroy_lock(&lock);
}
```