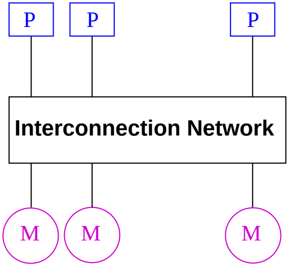
Shared Memory and OpenMP

- Background
    - Shared Memory Hardware
    - Shared Memory Languages
- OpenMP

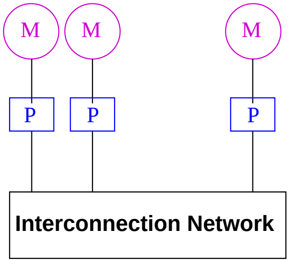
Parallel Hardware

Shared Memory Machines global memory can be acessed by all processors or cores. Information exchanged between threads using shared variables written by one thread and read by another. Need to coordinate access to shared variables.



Parallel Hardware

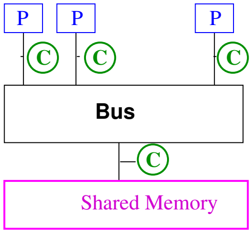
Distributed Memory Machines private memory for each processor, only accessible this processor, so no synchronization for memory accesses needed. Information exchanged by sending data from one processor to another via an interconnection network using explicit communication operations.



Hybrid approach increasingly common

Shared Memory Systems

Symmetric Multiprocessors (SMP): processors all connected to a large shared memory. Examples are processors connected by crossbar, or multicore chips. Key characteristic is *uniform memory access (UMA)*

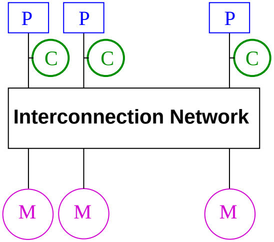


Caches are a problem - need to be kept *coherent* = when one CPU changes a value in memory, then all other CPUs will get the same value when they access it.

Shared Memory Systems

Distributed Shared Memory Memory is logically shared but physically distributed. Has *non-uniform memory access (NUMA)*

- Any processor can access any address in memory
- Cache lines (or pages) passed around machine. Difficulty is *cache coherency protocols*.
- CC-NUMA architecture (if network is cache-coherent)



(SGI Altix at NASA Ames - had 10,240 cpus of Itanium 2 nodes connected by Infiniband, was ranked 84 in June 2010 list, ranked 3 in 2008. Expensive!)

Parallel Programming Models

Programming model gives an abstract view of the machine describing

- Control
  - how is parallelism created?
  - what ordering is there between operations?
- Data
  - What data is private or shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations are used to coordinate parallelism
  - What operations are atomic (indivisible)?

Shared Memory Programming Model

Program consists of *threads* of control with

- shared variables
- private variables
- threads communicate implicitly by writing and reading shared variables
- threads coordinate by synchronizing on shared variables

Threads can be dynamically created and destroyed.

Other programming models: distributed memory, hybrid, data parallel programming model (single thread of control), shared address space,

What's a thread? A process?

Processes are independent execution units that contain their own state information and their own address space. They interact via interprocess communication mechanisms (generally managed by the operating system). One process may contain many threads. Processes are given system resources.

All threads within a process share the same address space, and can communicate directly using shared variables. Each thread has its own stack but only one data section, so global variables and heap-allocated data are shared (this can be dangerous).

What is state?

- instruction pointer
- Register file (one per thread)
- Stack pointer (one per thread)

Multithreaded Processors

- Both the above (SMP and Distributed Shared Memory Machines) are *shared address space* platforms.
- Also can have multithreading on a single processor. Switch between threads for long-latency memory operations
    - multiple thread *contexts* without full processors
    - Memory and some other state is shared
    - Can combine multithreading and multicore, e.g. Intel Hyperthreading, more generally SMT (simultaneous multithreading).
    - Cray MTA (MultiThreaded Architecture, hardware support for context switching every cycle), and Eldorado processors. Sun Niagra processors (multiple FPU and ALU per chip, 8 cores handle up to 8 threads per core)

Shared Memory Languages

- pthreads - POSIX (Portable Operating System Interface for Unix) threads; heavyweight, more clumsy
- PGAS languages - Partitioned Global Address Space UPC, Titanium, Co-Array Fortran; not yet popular enough, or efficient enough
- OpenMP - newer standard for shared memory parallel programming, lighter weight threads, not a programming language but an API for C and Fortran

OpenMP Overview

OpenMP is an API for multithreaded, shared memory parallelism.

- A set of compiler directives inserted in the source program
  - pragmas in C/C++ (pragma = compiler directive external to prog. lang. for giving additional info., usually non-portable, treated like comments if not understood)
  - (specially written) comments in fortran
- Library functions
- Environment variables

Goal is standardization, ease of use, portability. Allows incremental approach. Significant parallelism possible with just 3 or 4 directives. Works on SMPs and DSMs.

Allows fine and coarse-grained parallelism; loop level as well as explicit work assignment to threads as in SPMD.

What is OpenMP?

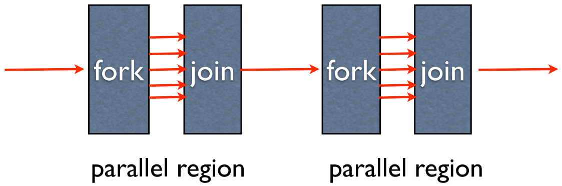
- http://www.openmp.org
- Maintained by the OpenMP Architecture Review Board (ARB) (non-profit group of organizations that interpret and update OpenMP, write new specs, etc. Includes Compaq/Digital, HP, Intel, IBM, KAI, SGI, Sun, DOE. (Endorsed by software and application vendors).
- Individuals also participate through cOMPunity, which participates in ARB, organizes workshops, etc.
- Started in 1997. OpenMP 3.0 just recently released.
- For Fortran (77,90,95), C and C++, on Unix, Windows NT and other platforms.

OpenMP = Open specifications for MultiProcessing

Basic Idea

Explicit programmer control of parallelization using fork-join model of parallel execution

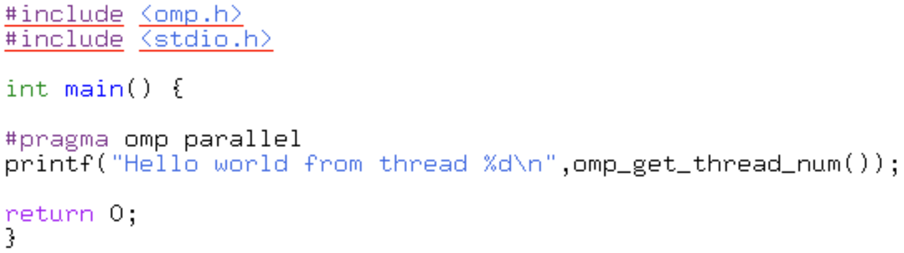
- all OpenMP programs begin as single process, the master thread, which executes until a parallel region construct encountered
- FORK: master thread creates team of parallel threads
- JOIN: When threads complete statements in parallel region construct they synchronize and terminate, leaving only the master thread. (similar to fork-join of Pthreads)



Basic Idea

- User inserts directives telling compiler how to execute statements
  - which parts are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads
  - `#pragma omp` in C and `!$omp` in Fortran
- Compiler generates explicit threaded code
- Rule of thumb: One thread per core (2 or 4 with hyperthreading)
- Dependencies in parallel parts require synchronization between threads

Simple Example



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

int main() {

#pragma omp parallel
printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Compile line:

```
gcc -fopenmp helloWorld.c
icc -openmp helloWorld.c
```

Simple Example

Sample Output:

```
MacBook-Pro% a.out
Hello world from thread 1
Hello world from thread 0
Hello world from thread 2
Hello world from thread 3

MacBook-Pro% a.out
Hello world from thread 0
Hello world from thread 3
Hello world from thread 2
Hello world from thread 1
```

(My laptop has 2 cores)
(Demos)

Setting the Number of Threads

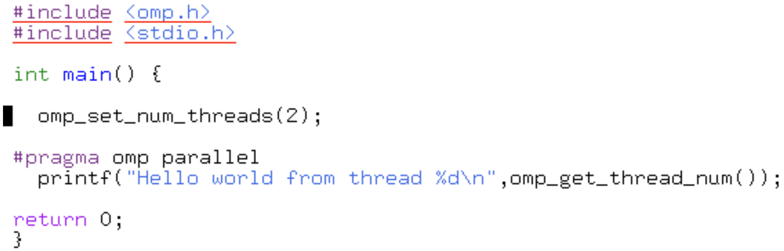
Environment Variables:

```
setenv OMP_NUM_THREADS 2
```
(cshell)

```
export OMP_NUM_THREADS=2
```
(bash shell)

Library call:

```
omp_set_num_threads(2)
```



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

int main() {

  omp_set_num_threads(2);

#pragma omp parallel
  printf("Hello world from thread %d\n",omp_get_thread_num());

return 0;
}
```

Parallel Construct

```
#include <omp.h>

int main(){
  int var1, var2, var3;

  ...serial Code

  #pragma omp parallel private(var1, var2) shared (var3)
 {
    ...parallel section
 }

 ...resume serial code

}
```

Parallel Directives

- When a thread reaches a PARALLEL directive, it becomes the master and has thread number 0.
- All threads execute the same code in the parallel region (Possibly redundant, or use work-sharing constructs to distribute the work)
- There is an implied barrier* at the end of a parallel section. Only the master thread continues past this point.
- If a thread terminates within a parallel region, all threads will terminates, and the result is undefined.
- Cannot branch into or out of a parallel region.

barrier - all threads wait for each other; no thread proceeds until all threads have reached that point

Parallel Directives

- If program compiled serially, openMP pragmas and comments ignored, stub library for omp library routines
- easy path to parallelization
- One source for both sequential and parallel helps maintenance.

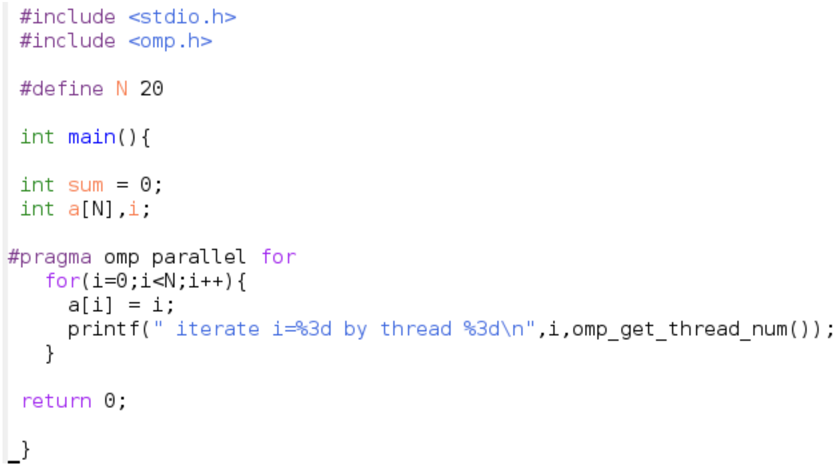
Work-Sharing Constructs

- work-sharing construct divides work among member threads. Must be dynamically within a parallel region.
- No new threads launched. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.

3 types of work-sharing construct (4 in Fortran - array constructs):

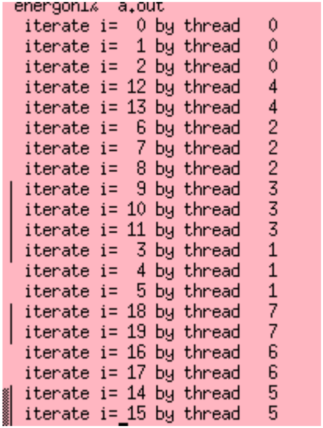
- for loop: share iterates of `for` loop (“data parallelism”) iterates must be independent
- sections: work broken into discrete section, each executed by a thread (“functional parallelism”)
- single: section of code executed by one thread only

FOR directive schedule example

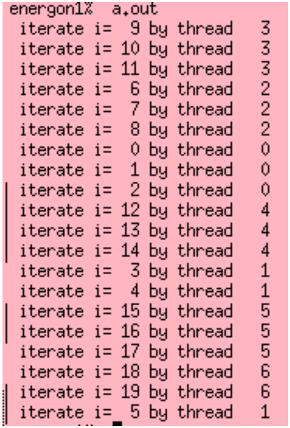


```c
 #include <stdio.h>
 #include <omp.h>

 #define N 20

 int main(){

 int sum = 0;
 int a[N],i;

#pragma omp parallel for
   for(i=0;i<N;i++){
     a[i] = i;
     printf(" iterate i=%3d by thread %3d\n",i,omp_get_thread_num());
   }

 return 0;

}
```

FOR directive schedule example



```
energon1%  a.out
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i=  5 by thread   1
 iterate i= 18 by thread   7
 iterate i= 19 by thread   7
 iterate i= 16 by thread   6
 iterate i= 17 by thread   6
 iterate i= 14 by thread   5
 iterate i= 15 by thread   5
```



```
energon1%  a.out
 iterate i=  9 by thread   3
 iterate i= 10 by thread   3
 iterate i= 11 by thread   3
 iterate i=  6 by thread   2
 iterate i=  7 by thread   2
 iterate i=  8 by thread   2
 iterate i=  0 by thread   0
 iterate i=  1 by thread   0
 iterate i=  2 by thread   0
 iterate i= 12 by thread   4
 iterate i= 13 by thread   4
 iterate i= 14 by thread   4
 iterate i=  3 by thread   1
 iterate i=  4 by thread   1
 iterate i= 15 by thread   5
 iterate i= 16 by thread   5
 iterate i= 17 by thread   5
 iterate i= 18 by thread   6
 iterate i= 19 by thread   6
 iterate i=  5 by thread   1
```

for loop with 20 iterations and 8 threads:

icc: 4 threads get 3 iterations and 4 threads get 2
gcc: 6 threads get 3 iterations, 1 thread gets 2, 1 gets none

OMP Directives

All directives:

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

Directives are:

- Case sensitive (not for Fortran)
- Only one directive-name per statement
- Directives apply to at most one succeeding statement, which must be a structured block.
- Continue on succeeding lines with backslash (`"\"`)

FOR directive

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

SCHEDULE: describes how to divide the loop iterates

- **static** = divided into pieces of size *chunk*, and statically assigned to threads. Default is approx. equal sized chunks (at most 1 per thread)
- **dynamic** = divided into pieces of size *chunk* and dynamically scheduled as requested. Default chunk size 1.
- **guided** = size of chunk decreases over time. (Init. size proportional to the number of unassigned iterations divided by number of threads, decreasing to *chunk size*)
- **runtime**=schedule decision deferred to runtime, set by environment variable OMP_SCHEDULE.

FOR example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region

   #pragma omp for nowait
   for (i=0;i<n;i++)
     b[i] = += a[i];

   #pragma omp for nowait
   for (i=0;i<n;i++)
     x[i] = 1./y[i];

} // end parallel region  (implied barrier)
```

Spawning tasks is expensive: reuse if possible.
nowait clause: minimize synchronization.

SECTIONS directive

```
#pragma omp sections [clause ...]
                   private (list)
                   firstprivate(list)
                   lastprivate(list)
                   reduction (operator: list)
                   nowait
{
 #pragma omp section
   structured block
 #pragma omp section
   structured block
}
```

- implied barrier at the end of a SECTIONS directive, unless a NOWAIT clause used
- for different numbers of threads and SECTIONS some threads get none or more than one
- cannot count on which thread executes which section
- no branching in or out of sections

Sections example

```
#pragma omp parallel shared(n,a,b,x,y), private(i)
{ // start parallel region
   #pragma omp sections nowait
   {
      #pragma omp section
      for (i=0;i<n;i++)
        b[i] = += a[i];

      #pragma omp section
      for (i=0;i<n;i++)
        x[i] = 1./y[i];

   } // end sections
} // end parallel region
```

SINGLE directive

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate(list)
                   nowait
 structured block
```

- SINGLE directive says only one thread in the team executes the enclosed code
- useful for code that isn't thread-safe (e.g. I/O)
- rest of threads wait at the end of enclosed code block (unless NOWAIT clause specified)
- no branching in or out of SINGLE block

firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

firstprivate example

What is wrong with this code snippet?

```
#pragma omp parallel for
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

By default, `x` is shared variable (i is private).

Could have: Thread 0 set `x` for some `i`.
Thread 1 sets `x` for different `i`.
Thread 0 uses `x` but it is now incorrect.

firstprivate example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about `i`,`dx`,`y`?

firstprivate example

Instead use:

```
#pragma omp parallel for private(x)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

What about `i,dx,y`?

By default `dx,n,y` shared.
`dx,n` used but not changed. `y` changed, but independently for each `i`

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

firstprivate example

What is wrong with this code?

```
dx = 1/n.;
#pragma omp parallel for private(x,dx)
for (i=0;i<n;i++){
    x = i*dx
    y(i) = exp(x) * cos(x) * sqrt(6*x+5);
}
```

Specifying `dx` private creates a new private variable for each thread, but it is not initialized.

firstprivate clause creates private variables and initializes to the value from the master thread before the loop.

lastprivate copies last value computed by a thread (for *i*=*n*) to the maser thread copy to continue execution.

Clauses

These clauses not strictly necessary but may be convenient (and may have performance penalties too).

- lastprivate private data is undefined after parallel construct. this gives it the value of last iteration (as if sequential) or sections construct (in lexical order).
- firstprivate pre-initialize private vars with value of variable with same name before parallel construct.
- default (`none | shared`). In fortran can also have private. Then only need to list exceptions. (none is better habit).
- nowait suppress implicit barrier at end of work sharing construct. Cannot ignore at end of parallel region. (But no guarantee that if have 2 `for` loops where second depends on data from first that same threads execute same iterates)

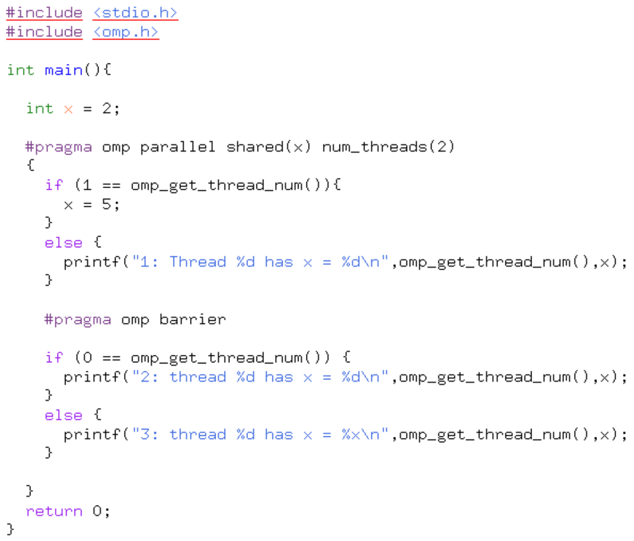
More Clauses

- if (logical expr) true = execute parallel region with team of threads; false = run serially (loop too small, too much overhead)
- reduction for assoc. and commutative operators compiler helps out; reduction variable is shared by default (no need to specify).

```
#pragma omp parallel for default(none) \
                          shared(n,a) \
                          reduction(+:sum)
  for (i=0;i<n;i++)
    sum += a[i]
  /* end of parallel reduction  */
```

  Also other arithmetic and logical ops., `min`,`max` instrinsics in Fortan only.
- copyprivate only with single direction. one thread reads and initializes private vars. which are copied to other threads before they leave barrier.
- threadprivate variables persist between different parallel sections (unlike private vars). (applies to global vars. must have dynamic false)

Race Condition* Example



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

int main(){

  int x = 2;

  #pragma omp parallel shared(x) num_threads(2)
  {
    if (1 == omp_get_thread_num()){
      x = 5;
    }
    else {
      printf("1: Thread %d has x = %d\n",omp_get_thread_num(),x);
    }

    #pragma omp barrier

    if (0 == omp_get_thread_num()) {
      printf("2: thread %d has x = %d\n",omp_get_thread_num(),x);
    }
    else {
      printf("3: thread %d has x = %x\n",omp_get_thread_num(),x);
    }

  }
  return 0;
}
```

*race condition= 2 or more threads access shared variable without synchronization and at least one is a write.

Synchronization

- Implicit barrier synchronization at end of parallel region (no explicit support for synch. subset of threads). Can invoke explicitly with `#pragma omp barrier`. All threads must see same sequence of work-sharing and barrier regions .
- critical sections: only one thread at a time in critical region with the same name. `#pragma omp critical [(name)]`
- atomic operation: protects updates to individual memory loc. Only simple expressions allowed. `#pragma omp atomic`
- locks: low-level run-time library routines (like mutex vars., semaphores)
- flush operation - forces the executing thread to make its values of shared data consistent with shared memory
- master (like single but not implied barrier at end), *ordered*, ...

At all these (implicit or explicit) synchronization points OpenMP ensures that threads have consistent values of shared data.

Critical Example

```
#pragma omp parallel sections
{
  #pragma omp section
  {
      task = produce_task();
      #pragma omp critical (task_queue)
      {
         insert_into_queue(task);
      }
  }
  #pragma omp section
  {
     #pragma omp critical (task_queue)
     {
       task = delete_from_queue(task);
     }
     consume_task(task);
  }
}
```

Atomic Examples

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
          ic = ic +1;
   }
```

`ic` incremented atomically

```
#pragma omp parallel shared(n,ic) private(i)
   for (i=0;i<n;i++){
     #pragma omp atomic
        ic = ic + bigfunc();
   }
```

`bigfunc` not atomic, only `ic` update

allowable atomic operations:

```
 x binop=  expr      x++ or ++x        x-- or --x
```

Atomic Example

```
int sum = 0;
#pragma omp parallel for shared(n,a,sum)
{
  for (i=0; i<n; i++){
  #pragma omp atomic
     sum = sum + a[i];
  }
}
```

Better to use a *reduction* clause:

```
int sum = 0;
#pragma omp parallel for shared(n,a)   \
                     reduction(+:sum)
{
  for (i=0; i<n; i++){
     sum += a[i];
  }
}
```

Locks

Locks control access to shared resources. Up to implementation to use spin locks (busy waiting) or not.

- Lock variables must be accessed only through locking routines:

  ```
  omp_init_lock   omp_destroy_lock
  omp_set_lock    omp_unset_lock   omp_test_lock
  ```

- In C, lock is a type `omp_lock_t` or `omp_nest_lock_t` (In Fortran lock variable is integer)
- initial state of lock is unlocked.
- `omp_set_lock(omp_lock_t *lock)` forces calling thread to wait until the specified lock is available. (Non-blocking version is `omp_test_lock`

Examing and setting a lock must be *uninterruptible* operation.

Lock Example

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

omp_lock_t my_lock;

int main() {
   omp_init_lock(&my_lock);

   #pragma omp parallel
   {
      int tid = omp_get_thread_num( );
      int i, j;

      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf("Thread %d - starting locked region\n", tid);

         printf("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }

   omp_destroy_lock(&my_lock);
}
```

Deadlock

Runtime situation that occurs when a thread is waiting for a resource that will never be available. Common situation is when two (or more) actions are each waiting for the other to finish (for example, 2 threads acquire 2 locks in different order)

```
work1() {  /* do some work */
  #pragma omp barrier
}
work2(){ /* do some work */
}
main(){
   #pragma omp parallel sections
   {
     #pragma omp section
         work1();

     #pragma omp section
         work2();
   }
} /* end main */
```

Also livelock: state changes but no progress is made.

References

- http://computing.llnl.gov/tutorials/openMP/
  very complete description of OpenMP for Fortran and C
- Rauber and Runger text
  text has small OpenMP section in chapter 6.
- *Using OpenMP*
  Portable Shared Memory Parallel Programming
  by Chapman, Jost and Van Der Pas
- http://www.openmp.org