Overview: The OpenMP Programming Model

- motivation and overview
- the parallel directive: clauses, equivalent pthread code, examples
- the ‘for’ directive and scheduling of loop iterations
- Pi example in OpenMP
- nested parallel directives
- synchronization
- handling of variables between threads
- sections
- tasks
- library functions and environment variables
- comparison with Pthreads

Refs: Grama et al Ch 7, Hager & Wellein Ch 6-7, Lin & Snyder Ch 6; OpenMP Tutorial from LLNL, The OpenMP Foundation
Shared Memory Parallel Programming

- explicit thread programming is messy
  - low-level primitives
  - originally non-standard, although better since Pthreads
  - used by system programmers, but ···
    ··· application programmers have better things to do!
- many application codes can be usefully supported by higher level constructs
  - led to proprietary directive-based approaches of Cray, SGI, Sun etc
- OpenMP is an API for shared memory parallel programming targeting Fortran, C and C++
  - standardizes the form of the proprietary directives
  - avoids the need for explicitly setting up mutexes, condition variables, data scope, and initialization
OpenMP

- specifications maintained by OpenMP Architecture Review Board (ARB)
  - members include AMD, Intel, Fujitsu, IBM, NVIDIA · · · cOMPunity
- versions 1.0 (Fortran ’97, C ’98), 1.1 and 2.0 (Fortran ’00, C/C++ ‘02), 2.5 (unified Fortran and C, 2005), 3.0 (2008), 3.1 (2011), 4.0 (2013)
- comprises of compiler directives, library routines and environment variables
  - C directives (case sensitive)
    
    #pragma omp directive-name [clause-list]
  - library calls begin with omp_
    
    void omp_set_num_threads(int nthreads);
  - environment variables begin with OMP_
    
    export OMP_NUM_THREADS=4

- OpenMP requires compiler support
  - activated via -fopenmp (gcc) or -openmp (icc) compiler flags
The Parallel Directive

- OpenMP uses a fork/join model: programs execute serially until they encounter a parallel directive:
  - this creates a group of threads
  - the number of threads is dependent on the `OMP_NUM_THREADS` environment variable or set via a function call, e.g. `omp_set_num_threads(nthreads)`
  - the main thread becomes the master thread, with a thread id of 0

```c
#pragma omp parallel [clause-list]
{ /* structured block */
}
```

- each thread executes the structured block
Parallel Directive: Clauses

Clauses are used to specify:

- **conditional parallelization**: to determine if the parallel construct results in creation/use of threads

  ```
  if (scalar-expression)
  ```

- **degree of concurrency**: explicit specification of the number of threads created/used

  ```
  num_threads(integer-expression)
  ```

- **data handling**: to indicate if specific variables are local to the thread (allocated on the thread’s stack), global, or ‘special’

  ```
  private(variable-list)
  ```

  ```
  shared(variable-list)
  ```

  ```
  firstprivate(variable-list)
  ```

  ```
  default(shared | none)
  ```
Compiler Translation: OpenMP to Pthreads

- **OpenMP code**

  ```c
  main() {
    int a, b;
    // serial segment
    #pragma omp parallel num_threads(8) private(a) shared(b)
    { // parallel segment
    }
    // rest of serial segment
  }
  ```

- **Pthread equivalent (structured block is *outlined*)**

  ```c
  main() {
    int a, b;
    // serial segment
    for (i=0; i<8; i++) pthread_create(......, internal_thunk,......);
    for (i=0; i<8; i++) pthread_join(........);
    // rest of serial segment
  }
  ```

  ```c
  void *internal_thunk(void *packaged_argument) {
    int a;
    // parallel segment
  }
  ```
Parallel Directive Examples

```c
#pragma omp parallel if (is_parallel == 1) num_threads(8) \n private(a) shared(b) firstprivate(c)
```

- if the value of variable `is_parallel` is one, eight threads are used
- each thread has private copy of `a` and `c`, but shares a single copy of `b`
- the value of each private copy of `c` is initialized to value of `c` before the parallel region

```c
#pragma omp parallel reduction(+:sum) num_threads(8) \n default(private)
```

- eight threads get a copy of the variable `sum`
- when threads exit, the values of these local copies are accumulated into the `sum` variable on the master thread
  - other reduction operations include `*`, `−`, `&`, `|`, `^`, `&&`, `||`
- all variables are private unless otherwise specified
Example: Computing Pi

- compute $\pi$ by generating random numbers in square with side length of 2 centered at (0,0) and counting numbers that fall within a circle of radius 1
  - area of square = 4, area of circle: $\pi r^2 = \pi$
  - ratio of points in circle to outside approaches $\pi/4$

```c
#pragma omp parallel private(i) shared(npoints) reduction(+: sum) num_threads(8)
{
  int seed = omp_num_threads(); //private
  num_threads = omp_get_num_threads();
  sum = 0;
  for (i = 0; i < npoints/num_threads; i++) {
    rand_x = (double) rand_range(&seed, -1, 1);
    rand_y = (double) rand_range(&seed, -1, 1);
    if ((rand_x * rand_x + rand_y * rand_y) <= 1.0)
      sum++;
  }
}
```

- the OpenMP code is very simple – c.f. the equivalent pthread code
The For Work-sharing Directive

- used in conjunction with the parallel directive to partition the for loop immediately afterwards

```c
#pragma omp parallel shared(npoints) \
    reduction(+: sum) num_threads(8)
{
    int seed = omp_get_thread_num();
    sum = 0;
    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_x = (double) rand_range(&seed, -1, 1);
        rand_y = (double) rand_range(&seed, -1, 1);
        if ((rand_x * rand_x + rand_y * rand_y) <= 1.0) sum++;
    }
}
```

- the loop index (i) is assumed to be private
- only two directives plus sequential code (code is easy to read/maintain)

- there is an implicit synchronization at the end of the loop
  - can add a nowait clause to prevent this
- it is common to merge the directives: #pragma omp parallel for ...
Assigning Iterations to Threads

- the `schedule` clause of the `for` directive assigns iterations to threads
- `schedule(static[,chunk-size])`
  - splits the iteration space into chunks of size `chunk-size` and allocates to threads in round-robin fashion
  - if the chunk size is unspecified, the number of chunks equals number of threads
- `schedule(dynamic[,chunk-size])`
  - the iteration space is split into `chunk-size` blocks that are scheduled dynamically
- `schedule(guided[,chunk-size])`
  - the chunk size decreases exponentially with iterations to a minimum of `chunk-size`
- `schedule(runtime)`
  - determine scheduling based on setting of the `OMP_SCHEDULE` environment variable

Case study: use of OpenMP in an advection application
Nesting Parallel Directives

- what happens for nested for loops?

```c
#pragma omp parallel for num_threads(2)
for (i = 0; i < Ni; i++) {
    #pragma omp parallel for num_threads(2)
    for (j = 0; j < Nj; j++) {
        ...
    }
}
```

- by default, the inner loop is serialized and run by one thread
- to enable multiple threads in nested loops requires the environment variable `OMP_NESTED` to be `TRUE`
- note - the use of synchronization constructs in nested parallel sections requires care (see the OpenMP specs)!
Synchronization #1

- **barrier**: threads wait until they have all reached this point

  ```
  #pragma omp barrier
  ```

- **single**: the following structured block is executed only by first thread to reach this point

  ```
  #pragma omp single [clause-list]
  ```
  
  - others wait at the end of structured block unless a `nowait` clause used

- **master**: only the master thread executes following block, other threads do NOT wait

  ```
  #pragma omp master
  ```

- **critical section**: only one thread is ever in the named critical section at any time

  ```
  #pragma omp critical [name]
  ```

- **atomic**: memory location updated in the following statement is done so in an atomic fashion

  - can achieve same effect using critical sections (possibly faster?)
Synchronization #2

- **ordered**: some operations within a `for` loop must be performed as if it were done so in sequential order
  
  ```c
  cumul_sum[0] = list[0];
  #pragma omp parallel for shared(cumul_sum, list, n)
  for (i=1; i<n; i++)
  {
      /* other processing on list[i] if required */
      #pragma omp ordered
      {
          cumul_sum[i] = cumul_sum[i-1] + list[i];
      }
  }
  ```

- **flush**: enforces a consistent view of memory

  - that variables have been flushed from registers into memory

  ```c
  #pragma omp flush [(variable-list)]
  ```
Data Handling

- **private**: an uninitialized local copy of the variable is made for each thread
- **shared**: variables are shared between threads
- **firstprivate**: make a local copy of an existing variable and assign it the same value
  - often better than multiple reads to a shared variable
- **lastprivate**: copies back to the master thread the variable's value from the thread which executed the last loop iteration (if executed serially)
- **threadprivate**: creates private variables but they persist between multiple parallel regions maintaining their values
- **copyin**: like firstprivate but for threadprivate variables
void matVecMult(int M, int N, double * restrict A,
                 double * restrict b, double * restrict c) {
    int i, j;
    for (i=0; i < M; i++)
        for (j=0; j < N; j++)
            c[i] = A[i+j*M] * b[j] + c[i];
}

● what are the possible ways you could use OpenMP to parallelize this code? (you may change the order of the loops)

● what data handling clauses would be needed?

● what if one or other of the matrix sizes was small?

● what parallel (and serial) performance issues might arise?
consider the partitioning of a fixed number of tasks across threads

- much less common than for loop partitioning
- an example of functional decomposition
- explicit programming naturally limits number of threads (scalability)

```c
#pragma omp sections
{
    #pragma omp section
    {
        taskA();
    }
    #pragma omp section
    {
        taskB();
    }
}
```

- separate threads will run taskA() and taskB()
- it is illegal to branch in or out of section blocks
OpenMP Tasks

- a task has
  - code to execute
  - a data environment (it owns its data)
  - an assigned thread that executes the code and uses the data
- creating a task involves two activities: packaging and execution
  - each encountering thread packages a new instance of a task (code and data)
  - some thread in the team executes the task at some later time
Task Syntax

```c
#pragma omp task [clause …]
```

where each clause can be one of:

- **if** (*scalar-expression*)
  If the expression is false, the creating thread executes the task (in its own environment)

- **untied**
  Relaxes the default binding of tasks to threads

- any of the previous data-handling directives `default, private, firstprivate or shared`

Task synchronization:

- at thread barriers (explicit or implicit) and task barriers, all tasks generated in that region must complete

```c
#pragma omp taskwait
```
Task Example

```c
int fib(int n)
{
    int i, j;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(i) firstprivate(n)
        i = fib(n-1);
        #pragma omp task shared(j) firstprivate(n)
        j = fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}
int main()
{
    int n = 10;
    omp_set_dynamic(0);
    omp_set_num_threads(4);
    #pragma omp parallel shared(n)
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
```
Task Issues

- **task switching:**
  - task scheduling points occur on task creation/exit/yields and the task synchronization points
  - when a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (task switching)
  - it can then return to original task and resume

- **tied tasks:**
  - by default suspended tasks must resume execution on the same thread as it was previously executing on
  - the `untied` clause relaxes this constraint

- **task generation pitfalls:**
  - very easy to generate many tasks very quickly!
  - the generating task will be suspended and its thread may start working on a long and boring task.
  - Other threads can consume all their tasks and have nothing to do . . .
Library Functions #1

- defined in header file

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

- controlling threads and processors
  ```c
  void omp_set_num_threads(int num_threads);
  int omp_get_num_threads();
  int omp_get_max_threads();
  int omp_get_thread_num();
  int omp_get_num_procs();
  int omp_in_parallel();
  ```

- controlling thread creation:
  ```c
  void omp_set_dynamic(int dynamic_threads);
  int omp_get_dynamic();
  void omp_set_nested(int nested);
  int omp_get_nested();
  ```
**Library Functions #2**

- **mutual exclusion support:**
  ```c
  void omp_init_lock(omp_lock_t *lock);
  void omp_destroy_lock(omp_lock_t *lock);
  void omp_set_lock(omp_lock_t *lock);
  void omp_unset_lock(omp_lock_t *lock);
  int omp_test_lock(omp_lock_t *lock);
  ```

- **nested mutual exclusion** (lock may be set $n$ times and only released after it is unset $n$ times):
  ```c
  void omp_init_nest_lock(omp_nest_lock_t *lock);
  void omp_destroy_nest_lock(omp_nest_lock_t *lock);
  omp_set_nest_lock(omp_nest_lock_t *lock);
  void omp_unset_nest_lock(omp_nest_lock_t *lock);
  int omp_test_nest_lock(omp_nest_lock_t *lock);
  ```
OpenMP Environment Variables

- **OMP_NUM_THREADS**: default number of threads entering parallel region
- **OMP_DYNAMIC**: if TRUE it permits the number of threads to change during execution
- **OMP_NESTED**: if TRUE it permits nested parallel regions
- **OMP_SCHEDULE**: determines scheduling for loops that are specified to have runtime scheduling
  
  ```
  export OMP_SCHEDULE="static,4"
  export OMP_SCHEDULE="dynamic"
  export OMP_SCHEDULE="guided"
  ```
OpenMP and Pthreads

- OpenMP removes the need for a programmer to initialize task attributes, set up arguments to threads, partition iteration spaces etc.
- OpenMP code can closely resemble serial code.
- OpenMP is particularly useful for static or regular problems.
- OpenMP users are hostage to availability of an OpenMP compiler.
  - Performance heavily dependent on quality of compiler.
- OpenMP more easily allows incremental parallelization.
- Pthreads data exchange is more apparent so false sharing and contention is less likely.
- Pthreads has a richer API that is much more flexible, e.g. condition waits, locks of different types etc.
- Pthreads is library based.

Must balance above before deciding on parallel model.