Programming Shared-memory Platforms with OpenMP

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Topics for Today

• Introduction to OpenMP

• OpenMP directives
  — concurrency directives
    – parallel regions
    – loops, sections, tasks
  — synchronization directives
    – reductions, barrier, critical, ordered
  — data handling clauses
    – shared, private, firstprivate, lastprivate
  — tasks

• Performance tuning hints

• Library primitives

• Environment variables
What is OpenMP?

**Open specifications for Multi Processing**

- An API for explicit multi-threaded, shared memory parallelism
- Three components
  - compiler directives
  - runtime library routines
  - environment variables
- Higher-level than library-based programming models
  - implicit mapping and load balancing of work
- Portable
  - API is specified for C/C++ and Fortran
  - implementations on almost all platforms
- Standard
OpenMP at a Glance

- Application
- Compiler
- Runtime Library
- OS Threads (e.g., Pthreads)
- User
- Environment Variables
OpenMP Is Not

• An automatic parallel programming model
  — parallelism is explicit
  — programmer has full control (and responsibility) over parallelization

• Meant for distributed-memory parallel systems (by itself)
  — designed for shared address spaced machines

• Necessarily implemented identically by all vendors

• Guaranteed to make the most efficient use of shared memory
  — no data locality control
OpenMP Targets Ease of Use

• OpenMP does not require that single-threaded code be changed for threading
  — enables incremental parallelization of a serial program

• OpenMP only adds compiler directives
  — pragmas (C/C++); significant comments in Fortran
    – if a compiler does not recognize a directive, it ignores it
  — simple & limited set of directives for shared memory programs
  — significant parallelism possible using just 3 or 4 directives
    – both coarse-grain and fine-grain parallelism

• If OpenMP is disabled when compiling a program, the program will execute sequentially
OpenMP: Fork-Join Parallelism

- OpenMP program begins execution as a single master thread
- Master thread executes sequentially until 1\textsuperscript{st} parallel region
- When a parallel region is encountered, master thread
  — creates a group of threads
  — becomes the master of this group of threads
  — is assigned the thread id 0 within the group

master thread shown in red
OpenMP Directive Format

• OpenMP directive forms
  — C and C++ use compiler directives
    – prefix: `#pragma` …
  — Fortran uses significant comments
    – prefixes: `!$omp`, `c$omp`, `*$omp`

• A directive consists of a directive name followed by clauses
  C: `#pragma omp parallel num_threads(4)`…
  Fortran: `!$omp parallel num_threads(4)`…
A Simple Example Using parallel

Program

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        printf("Hello from thread %d\n", i);
    }
}

Output
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3

order of output may vary!
OpenMP `parallel` Region Directive

```c
#pragma omp parallel [clause list]
```

Typical clauses in `[clause list]`

- **Conditional parallelization**
  - `if (scalar expression)`
    - determines whether the `parallel` construct creates threads

- **Degree of concurrency**
  - `num_threads(integer expression)`: max # threads to create

- **Data Scoping**
  - `private (variable list)`
    - specifies variables local to each thread
  - `firstprivate (variable list)`
    - similar to the private
    - private variables are initialized to variable value before the `parallel` directive
  - `shared (variable list)`
    - specifies that variables are shared across all the threads
  - `default (data scoping specifier)`
    - default data scoping specifier may be `shared` or `none`
Interpreting an OpenMP Parallel Directive

```c
#pragma omp parallel if (is_parallel==1) num_threads(8) \ 
    shared(b) private(a) firstprivate(c) default(none)
{
    /* structured block */
}
```

Meaning

- **if (is_parallel== 1) num_threads(8)**
  - If the value of the variable `is_parallel` is one, create 8 threads
- **shared(b)**
  - Each thread shares a single copy of variable `b`
- **private(a) firstprivate(c)**
  - Each thread gets private copies of variables `a` and `c`
  - Each private copy of `c` is initialized with the value of `c` in the “initial thread,” which is the one that encounters the parallel directive
- **default(none)**
  - Default state of a variable is specified as `none` (rather than `shared`)
  - Signals error if not all variables are specified as `shared` or `private`
Specifying Worksharing

Within the scope of a parallel directive, worksharing directives allow concurrency between iterations or tasks.

OpenMP provides four directives:

- **DO/for**: concurrent loop iterations
- **sections**: concurrent tasks
- **workshare**: partition execution of statements in block
- **single**: one arbitrary thread executes the code
Worksharing **DO/for** Directive

**for** directive partitions parallel iterations across threads

**DO** is the analogous directive for Fortran

- **Usage:**
  ```
  #pragma omp for [clause list]
  /* for loop */
  ```

- **Possible clauses in** [clause list]
  - private, firstprivate, lastprivate
  - reduction
  - schedule, nowait, and ordered

- **Implicit barrier at end of** for loop
A Simple Example Using **parallel** and **for**

**Program**

```c
void main() {
    #pragma omp parallel num_threads(3)
    {
        int i;
        printf("Hello world\n");
        #pragma omp for
        for (i = 1; i <= 4; i++) {
            printf("Iteration %d\n",i);
        }
        printf("Goodbye world\n");
    }
}
```

**Output**

Hello world
Hello world
Hello world
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Goodbye world
Goodbye world
Goodbye world
Reduction Clause for Parallel Directive

Specifies how to combine local copies of a variable in different threads into a single copy at the master when threads exit

• Usage: `reduction (operator: variable list)`
  — variables in list are implicitly private to threads

• Reduction operators: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, and `||`

• Usage sketch

```c
#pragma omp parallel reduction(+: sum) num_threads(8)
{
  /* compute local sum in each thread here */
}
/* sum here contains sum of all local instances of sum */
```
Approximate Pi

— generate random points with $x, y \in [-0.5, 0.5]$ 
— test if point inside the circle, i.e., $x^2 + y^2 < (0.5)^2$
— ratio of circle to square = $\pi r^2 / 4r^2 = \pi / 4$
— $\pi \approx 4 \times (\text{number of points inside the circle}) / (\text{number of points total})$
OpenMP Reduction Clause Example

OpenMP threaded program to estimate PI

```c
#pragma omp parallel default(private) shared (npoints) \  reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        coord_x = (double)(rand_r(&seed))/(double)(RAND_MAX) - 0.5;
        coord_y = (double)(rand_r(&seed))/(double)(RAND_MAX) - 0.5;
        if ((coord_x * coord_x + coord_y * coord_y) < 0.25)
            sum++;
    }
}
```

- a local copy of sum for each thread
- all local copies of sum added together and stored in master
Using Worksharing for Directive

```c
#pragma omp parallel default(private) shared (npoints) \ 
    reduction(+: sum) num_threads(8)
{
    sum = 0;

    #pragma omp for
    for (i = 0; i < npoints; i++) {
        rand_no_x = (double)(rand_r(&seed))/(double)(RAND_MAX);
        rand_no_y = (double)(rand_r(&seed))/(double)(RAND_MAX);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
             (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            sum ++;
    }
}
```

worksharing for
divides work

Implicit barrier at end of loop
Mapping Iterations to Threads

**schedule clause of the for directive**

- **Recipe for mapping iterations to threads**
- **Usage:** `schedule(scheduling_class[,chunk])`.
- **Four scheduling classes**
  - **static:** work partitioned at compile time
    - iterations statically divided into pieces of size `chunk`
    - statically assigned to threads
  - **dynamic:** work evenly partitioned at run time
    - iterations are divided into pieces of size `chunk`
    - chunks dynamically scheduled among the threads
    - when a thread finishes one chunk, it is dynamically assigned another
    - default chunk size is 1
  - **guided:** guided self-scheduling
    - chunk size is exponentially reduced with each dispatched piece of work
    - the default minimum chunk size is 1
  - **runtime:**
    - scheduling decision from environment variable OMP_SCHEDULE
    - illegal to specify a chunk size for this clause.
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) \ 
    shared (a, b, c, dim) num_threads(4)
#pragma omp for schedule(static)
for (i = 0; i < dim; i++) {
    for (j = 0; j < dim; j++) {
        c(i,j) = 0;
        for (k = 0; k < dim; k++) {
            c(i,j) += a(i, k) * b(k, j);
        }
    }
}
Avoiding Unwanted Synchronization

- Default: worksharing `for` loops end with an implicit barrier
- Often, less synchronization is appropriate
  - series of independent `for`-directives within a parallel construct
- `nowait` clause
  - modifies a `for` directive
  - avoids implicit barrier at end of `for`
Avoiding Synchronization with `nowait`

```c
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < nmax; i++)
    a[i] = ...;

    #pragma omp for
    for (i = 0; i < mmax; i++)
    b[i] = ... anything but a ...;
}
```

any thread can begin second loop immediately without waiting for other threads to finish first loop
Worksharing `sections` Directive

The `sections` directive enables specification of task parallelism.

- **Usage**

  ```
  #pragma omp sections [clause list]
  {
      [#pragma omp section
       /* structured block */
      ]
      [#pragma omp section
       /* structured block */
      ]
      ...
  }
  ```

  Brackets here represent that the section is optional, not the syntax for using them.
Using the `sections` Directive

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

- **Parallel section encloses all parallel work**
- **Sections: task parallelism**
- Three concurrent tasks; tasks need not be procedure calls
Nesting parallel Directives

• Nested parallelism enabled using the OMP_NESTED environment variable
  – OMP_NESTED = TRUE → nested parallelism is enabled

• Each parallel directive creates a new team of threads

master thread shown in red
Synchronization Constructs in OpenMP

```c
#pragma omp barrier
wait until all threads arrive here
```

```c
#pragma omp single [clause list]
structured block
```

```c
#pragma omp master
structured block
```

Use MASTER instead of SINGLE wherever possible

- MASTER = IF-statement with no implicit BARRIER
  - equivalent to
    ```c
    IF(omp_get_thread_num() == 0) {...}
    ```

- SINGLE: implemented like other worksharing constructs
  - keeping track of which thread reached SINGLE first adds overhead
Synchronization Constructs in OpenMP

`#pragma omp critical [(name)]`  critical section: like a named lock
structured block

`#pragma omp ordered`  for loops with carried dependences
structured block
#pragma omp parallel
{
    #pragma omp for nowait shared(best_cost)
    for (i = 0; i < nmax; i++) {
        my_cost = ...;
        ...
    }
    #pragma omp critical
    {
        if (best_cost < my_cost)
            best_cost = my_cost;
    }
    ...
}

critical ensures mutual exclusion when accessing shared state
Example Using \texttt{ordered}

```c
#pragma omp parallel
{
#pragma omp for nowait shared(a)
    for (k = 0; k < nmax; k++) {
        ...
#pragma omp ordered
    {
        a[k] = a[k-1] + ...;
    }
        ...
    }
}
```

\texttt{ordered} ensures carried dependence does not cause a data race
Orphaned Directives

- Directives may not be lexically nested in a parallel region
  - may occur in a separate program unit

- Dynamically bind to enclosing parallel region at run time

- Benefits
  - enables parallelism to be added with a minimum of restructuring
  - improves performance: enables single parallel region to bind with worksharing constructs in multiple called routines

- Execution rules
  - an orphaned worksharing construct is executed serially when not called from within a parallel region
OpenMP 3.0 Tasks

• Motivation: support parallelization of irregular problems
  — unbounded loops
  — recursive algorithms
  — producer consumer

• What is a task?
  — work unit
    – execution can begin immediately, or be deferred
  — components of a task
    – code to execute, data environment, internal control variables

• Task execution
  — data environment is constructed at creation
  — tasks are executed by threads of a team
  — a task can be tied to a thread (i.e. migration/stealing not allowed)
    – by default: a task is tied to the first thread that executes it
OpenMP 3.0 Tasks

```c
#pragma omp task [clause list]
```

Possible clauses in `[clause list]`

- **Conditional parallelization**
  - `if` *(scalar expression)*
    - determines whether the construct creates a task

- **Binding to threads**
  - `untied`

- **Data scoping**
  - `private` *(variable list)*
    - specifies variables local to the child task
  - `firstprivate` *(variable list)*
    - similar to the private
    - private variables are initialized to value in parent task before the directive
  - `shared` *(variable list)*
    - specifies that variables are shared with the parent task
  - `default` *(data handling specifier)*
    - default data handling specifier may be `shared` or `none`
Composing Tasks and Regions

```c
#pragma omp parallel
{
    #pragma omp task
    x();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        y();
    }
}
```

- one x task created for each thread in the parallel region
- all x tasks complete at barrier
- one y task created
- region end: y task completes
Data Scoping for Tasks is Tricky

If no default clause specified

- Static and global variables are **shared**
- Automatic (local) variables are **private**
- Variables for orphaned tasks are **firstprivate** by default
- Variables for non-orphaned tasks inherit the shared attribute
  - Task variables are **firstprivate unless shared** in the enclosing context
Fibonacci using (Power9ed) OpenMP 3.0 Tasks

```c
int fib ( int n )
{
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}

int main (int argc, char **argv)
{
    int n, result;
    n = atoi(argv[1]);
    #pragma omp parallel
    {
        #pragma omp single
        {
            result = fib(n);
        }
    }
    printf("fib(%d) = %d\n", n, result);
}
```

- `need shared` for `x` and `y`; default would be `firstprivate`
- `suspend parent task until children finish`
- `create team of threads to execute tasks`
- `only one thread performs the outermost call`
List Traversal

Element first, e;
#pragma omp parallel
#pragma omp single
{
    for (e = first; e; e = e->next)
#pragma omp task firstprivate(e)
    process(e);
}
• Tied tasks
  — only the thread that the task is tied to may execute it
  — task can only be suspended at a suspend point
    – task creation
    – task finish
    – taskwait
    – barrier
  — if a task is not suspended at a barrier, it can only switch to a descendant of any task tied to the thread

• Untied tasks
  — no scheduling restrictions
    – can suspend at any point
    – can switch to any task
  — implementation may schedule for locality and/or load balance
## Summary of Clause Applicability

<table>
<thead>
<tr>
<th>Clause</th>
<th>PARALLEL</th>
<th>DO/for</th>
<th>SECTIONS</th>
<th>SINGLE</th>
<th>PARALLEL DO/for</th>
<th>PARALLEL SECTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIRSTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LASTPRIVATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REDUCTION</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COPYIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCHEDULE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ORDERED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOWAIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Parallelize at the highest level, e.g. outermost DO/for loops

```fortran
!$OMP PARALLEL
....
do j = 1, 20000
 !$OMP DO
  do k = 1, 10000
   ...
  enddo !k
 !$OMP END DO
enddo !j
 ...
 !$OMP END PARALLEL
```

```fortran
!$OMP PARALLEL
....
 !$OMP DO
 do k = 1, 10000
   do j = 1, 20000
     ...
   enddo !j
 enddo !k
 !$OMP END DO
....
 !$OMP END PARALLEL
```

Slower

Faster
Performance Tuning Hints

Merge independent parallel loops when possible

Slower

Faster
Performance Tuning Hints

Minimize use of synchronization

- **BARRIER**
- **CRITICAL sections**
  - if necessary, use named CRITICAL for fine-grained locking
- **ORDERED regions**
- Use **NOWAIT** clause to avoid unnecessary barriers
  - adding NOWAIT to a region’s final DO eliminates a redundant barrier
- Use explicit **FLUSH** with care
  - flushes can evict cached values
  - subsequent data accesses may require reloads from memory

```c
data = ...
#pragma omp flush (data)
data_available = true;
```
OpenMP Library Functions

• Processor count

  int omp_get_num_procs(); /* # processors currently available */
  int omp_in_parallel(); /* determine whether running in parallel */

• Thread count and identity

  /* max # threads for next parallel region. only call in serial region */
  void omp_set_num_threads(int num_threads);

  int omp_get_num_threads(); /*# threads currently active */
OpenMP Library Functions

• Controlling and monitoring thread creation

  void omp_set_dynamic (int dynamic_threads);
  int omp_get_dynamic ();
  void omp_set_nested (int nested);
  int omp_get_nested ();

• Mutual exclusion

  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);

  — Lock routines have a nested lock counterpart for recursive mutexes
OpenMP Environment Variables

- **OMP_NUM_THREADS**
  - specifies the default number of threads for a parallel region

- **OMP_DYNAMIC**
  - specifies if the number of threads can be dynamically changed

- **OMP_NESTED**
  - enables nested parallelism (may be nominal: one thread)

- **OMP_SCHEDULE**
  - specifies scheduling of for-loops if the clause specifies runtime

- **OMP_STACKSIZE** (for non-master threads)

- **OMP_WAIT_POLICY** (active or passive)

- **OMP_MAX_ACTIVE_LEVELS**
  - integer value for maximum # nested parallel regions

- **OMP_THREAD_LIMIT** (# threads for entire program)
OpenMP Directives vs. Library-based Models

• Directive advantages
  — directives facilitate a variety of thread-related tasks
  — frees programmer from
    – initializing attribute objects
    – setting up thread arguments
    – partitioning iteration spaces, …

• Directive disadvantages
  — data exchange is less apparent
    – leads to mysterious overheads
      data movement, false sharing, and contention
  — API is less expressive than Pthreads
    – lacks condition waits, locks of different types, and flexibility for building composite synchronization operations
OpenMP is Continuing to Evolve

- OpenMP 4.5 is the most recent standard (November 2015)
- Features new to OpenMP 4
  - SIMD support
    - e.g., $a[0:n-1] = 0$
  - locality and affinity
    - control mapping of threads to processor cores
    - `proc_bind(master, spread, close)`
  - additional synchronization mechanisms
    - e.g., `taskgroup`, `taskwait`
  - offload computation to accelerators, e.g. GPUs
- Work in progress
  - OpenMP Tools API (OMPT)
References


• Adapted from slides “Programming Shared Address Space Platforms” by Ananth Grama


• Sun Microsystems. OpenMP OpenMP API User's Guide. Chapter 7 “Performance Considerations” http://docs.sun.com/source/819-3694/7_tuning.html


