Brett Estrade
<estrabd@cs.uh.edu>

HPCTools Group
University of Houston
Department of Computer Science

http://www.cs.uh.edu/~hpctools
Multiple Processes on this System

Main Memory

- process 0
- process 1
- process 2
- process 3
Multiple Processes on this System - *logically no different than MPI's model*
A Single Process with Multiple Threads on this System

Main Memory

process 0

cpu 0

Thread 0

cpu 1

Thread 1

cpu 2

Thread 2

cpu 3

Thread 3

p0
OpenMP's Execution Model – *Fork & Join*

A Guide to OpenMP
An Example of Sharing Memory

- **Main Memory**
  - CPU 0
  - Thread 0
  - CPU 1
  - Thread 1
  - CPU 2
  - Thread 2
  - CPU 3
  - Thread 3

- **Process 0**
  - `int a`

- **Process 3**
  - `p0`
An Example of Sharing Memory

- `p0`
- `int a` written by `Thread 0` on `cpu 0`
- `Main Memory`
- `process 0`
An Example of Sharing Memory

- **Main Memory**
  - `p0`
  - `int a`
  - `write`
  - `read`

- **Processes**
  - `process 0`
  - `cpu 0`
  - `Thread 0`
  - `cpu 1`
  - `Thread 1`
  - `cpu 2`
  - `Thread 2`
  - `cpu 3`
  - `Thread 3`
An Example of Sharing Memory

```
process 0

Main Memory

process 0

int a

write

read

p0

read
```

Thread 0

Thread 1

Thread 2

Thread 3
An Example of Sharing Memory

- **Process 0**: Main Memory
  - **CPU 0**: Thread 0
    - Write to `int a`
  - **CPU 1**: Thread 1
    - Read from `int a`
  - **CPU 2**: Thread 2
    - Read from `int a`
  - **CPU 3**: Thread 3
    - Read from `int a`
Notes on the Example of Sharing Memory

- Okay, so that was *highly* idealized
- Read/Write order matters (R/W hazards apply)
- Could represent a *race condition*
- Race conditions introduce *non-determinism* (not good)
- Threaded programs can be extremely difficult to debug
- Proper precautions must be made to eliminate these
What is OpenMP?

- A directive based language standard
- A user level API and *runtime* environment
- A widely supported standard language specification
- A *community* of active users & researchers
```c
#include <stdio.h>
#include <omp.h>

int main (int argc, char *argv[]) {
    int tid, numt;
    numt = omp_get_num_threads();
    #pragma omp parallel private(tid) shared(numt)
    {
        tid = omp_get_thread_num();
        printf("hi, from %d\n", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!\n", numt);
        }
    }
    return 0;
}
```
The timeline of the OpenMP Standard Specification

- **1997**: OpenMP 1.0 for Fortran
- **2000**: OpenMP 2.0 for Fortran
- **2002**: OpenMP 1.0 for C/C++
- **2005**: OpenMP 2.5 for all
- **2008**: OpenMP 3.0 for all
- **Draft**: 3.1 for all
OpenMP vs MPI

There is no silver bullet

And that makes Teen Wolf happy
Some Benefits of using OpenMP

- It's **portable**, supported by most C/C++ & Fortran compilers
- The development cycle is a friendly one
  - Can be introduced *iteratively* into existing code
  - Correctness can be verified along the way
  - Likewise, performance benefits can be gauged
- Optimizing memory access in the serial program will benefit the threaded version (e.g., false sharing, etc)
- It can be fun to use (immediate gratification)
What Does OpenMP Provide?

- An abstraction above low level thread libraries
- Directives, hidden inside of structured comments
- A *runtime* library that manages execution dynamically
- Control via environmental variables & a *runtime* API
- Expectations of behavior & sensible defaults
- A promise of *interface* portability;
## What Compilers Support OpenMP?

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Languages</th>
<th>Supported Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>C/C++(10.1),Fortran(13.1)</td>
<td>Full 3.0 support</td>
</tr>
<tr>
<td>Sun/Oracle</td>
<td>C/C++,Fortran(12.1)</td>
<td>Full 3.0 support</td>
</tr>
<tr>
<td>Intel</td>
<td>C/C++,Fortran(11.0)</td>
<td>Full 3.0 support</td>
</tr>
<tr>
<td>Portland Group</td>
<td>C/C++,Fortran</td>
<td>Full 3.0 support</td>
</tr>
<tr>
<td>Absoft</td>
<td>Fortran(11.0)</td>
<td>Full 2.5 support</td>
</tr>
<tr>
<td>Lahey/Fujitsu</td>
<td>C/C++,Fortran(6.2)</td>
<td>Full 2.0 support</td>
</tr>
<tr>
<td>PathScale</td>
<td>C/C++,Fortran</td>
<td>Full 2.5 support (based on Open64)</td>
</tr>
<tr>
<td>HP</td>
<td>C/C++,Fortran</td>
<td>Full 2.5 support</td>
</tr>
<tr>
<td>Cray</td>
<td>C/C++,Fortran</td>
<td>Full 3.0 on Cray XT Series Linux</td>
</tr>
<tr>
<td>GNU</td>
<td>C/C++,Fortran</td>
<td>Working towards full 3.0</td>
</tr>
<tr>
<td>Microsoft</td>
<td>C/C++,Fortran</td>
<td>Full 2.0</td>
</tr>
</tbody>
</table>
A Guide to OpenMP

OpenMP Research Activities

- A lot of research goes into the OpenMP's standard
- International Workshop on OpenMP (IWOMP)
- Suites: validation, NAS, SPEC, EPCC, BOTS
- Open Source Research Compilers:
  - **OpenUH**
  - NANOS
  - Rose/{OMNI,GCC}
  - **MPC**, etc
  - Commercial R&D
- cOMPunity - [http://www.compunity.org](http://www.compunity.org)
- Applications research, i.e., HPC users, etc
### Compiling and Executing Examples

- **IBM XL Suite:**
  - `xlc_r, xlf90, etc`

  ```bash
  % xlc_r -qsmp=omp test.c -o test.x   # compile it
  % OMP_NUM_THREADS=4 ./test.x         # execute it
  ```

- **OpenUH:**
  - `uhcc, uhf90, etc`

  ```bash
  % uhcc -mp test.c -o test.x           # compile it
  % OMP_NUM_THREADS=4 ./test.x          # execute it
  ```
• Contained inside of *structured comments*

C/C++:

```c
#pragma omp <directive> <clauses>
```

Fortran:

```fortran
 !$OMP <directive> <clauses>
```

• OpenMP compliant compilers find and parse directives
• Non-compliant *should* safely ignore them as comments
• A *construct* is a directive that affects the enclosing code
• Imperative (standalone) directives exist
• *Clauses* control the behavior of directives
The OpenMP “Runtime” Library (RTL)

• The “runtime” manages the multi-threaded execution:
  - It's used by the resulting executable OpenMP program
  - It's what spawns threads (e.g., calls pthreads)
  - It's what manages shared & private memory
  - It's what distributes (shares) work among threads
  - It's what synchronizes threads & tasks
  - It's what reduces variables and keeps lastprivate
  - It's what is influenced by envars & the user level API

  • http://www2.cs.uh.edu/~estrabd/OpenUH/r593/html-libopenmp/
  • __omp_fork(...) call graph
Useful OpenMP Environmental Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_STACKSIZE
- OMP_NESTED
- OMP_THREAD_LIMIT
- OMP_MAX_ACTIVE_LEVELS
3 Types of Runtime Functions

Execution environment routines; e.g.,
- `omp_{set, get}_num_threads`
- `omp_{set, get}_dynamic`
- Each envvar has a corresponding get/set

Locking routines; e.g.,
- `omp_{init, destroy}_{, nest_}lock`
- `omp_test_{, nest_}lock`
- `omp_{set, unset}_{, nest_}lock`

Timing routines; e.g.,
- `omp_get_wtime`
- `omp_get_wtick`
How Is an OpenMP Program Compiled? Here's How OpenUH does it.

Liao, et. al.: http://www2.cs.uh.edu/~copper/openuh.pdf
What Does the Transformed Code Look like?

- Intermediate code, “W2C”
  - uhcc -mp -gnu3 -CLIST:emit_nested_pu simple.c
  - http://www2.cs.uh.edu/~estrabd/OpenMP/simple/

```c
#include <stdio.h>
int main() {
    int my_id;
    #pragma omp parallel default(none) private(my_id)
    {
        my_id = omp_get_thread_num();
        printf("hello from %d\n", my_id);
    }
    return 0;
}
```

The original `main()` is outlined to `__omprg_main_1()`
The new `main()`

```c
extern _INT32 main() {
    register _INT32 _w2c___ompv_ok_to_fork;
    register _UINT64 _w2c_reg3;
    register _INT32 _w2c___comma;
    _INT32 my_id;
    _INT32 __ompv_gtid_s1;

    /* Begin_of_nested_PU(s) */

    _w2c___ompv_ok_to_fork = 1;
    if(_w2c___ompv_ok_to_fork) {
        _w2c___ompv_ok_to_fork = __ompc_can_fork();
    }
    if(_w2c___ompv_ok_to_fork) {
        __ompc_fork(0, &__omprg_main_1, _w2c_reg3);
    } else {
        __ompv_gtid_s1 = __ompc_get_local_thread_num();
        __ompc_serialized_parallel();
        _w2c___comma = omp_get_thread_num();
        my_id = _w2c___comma;
        printf("hello from %d\n", my_id);
        __ompc_end_serialized_parallel();
    }
    return 0;
} /* main */
```

calls RTL fork and passes function pointer to outlined `main()`

No body wants to code like this, so let the compiler and runtime do most all this tedious work!
The parallel Construct

- Where the “fork” occurs (__ompc_fork(...) )
- Encloses all other OpenMP constructs & directives
- This construct accepts the following clauses: if, num_threads, private, firstprivate, shared, default, copyin, reduction
- Can call functions that contain “orphan” constructs
  - Statically outside of parallel, but lexically inside during runtime
- Can be nested
A Simple OpenMP Example

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

int main (int argc, char *argv[]) {
    int tid, numt;
    numt = omp_get_num_threads();
    #pragma omp parallel private(tid) shared(numt)
    {
        tid = omp_get_thread_num();
        printf("hi, from %d
", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!
", numt);
        }
    }
    return 0;
}
```

Output using 4 *threads*:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

Note, thread order not guaranteed!
The Fortran Version

Output using **4 threads:**

- hi, from 3
- hi, from 0
- hi, from 2
- hi, from 1

**4 threads say hi!**

Note, thread order not guaranteed!
Now, Just the Parallelized Code

C/C++

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

int main (int argc, char *argv[]) {
    int tid, numt;
    numt = omp_get_num_threads();
    #pragma omp parallel private(tid) shared(numt)
    {
        tid = omp_get_thread_num();
        printf("hi, from %d
", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!
",numt);
        }
    }
    return 0;
}
```

F90

```f90
program hello90
use omp_lib
integer :: id, numt
numt = omp_get_num_threads()
!$omp parallel private(id) shared(numt)
    tid = omp_get_thread_num()
    write (*,*) 'hi, from', tid
!$omp barrier
    if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
    end if
!$omp end parallel
end program
```

Output using 4 threads:

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!
```

Note, thread order not guaranteed!
Trace of The Execution

A Guide to OpenMP

all threads call `printf`

only thread with `tid == 0` does this

fork

0

hi, from 0

B

0 == 0

5 threads say hi!

join

0

1

hi, from 1

B

1 != 0

2

hi, from 2

B

2 != 0

3

hi, from 3

B

3 != 0

B = wait for all threads @ barrier before progressing further.

other threads wait

J
• The “if” clause contains a conditional expression.
• If TRUE, forking occurs, else it doesn't

```c
int n = some_func();
#pragma omp parallel if(n>5)
{
    ... do stuff in parallel
}
```

• The “num_threads” clause is another way to control the number of threads active in a parallel construct

```c
int n = some_func();
#pragma omp parallel num_threads(n)
{
    ... do stuff in parallel
}
```
The Data Environment Among Threads

- default([shared] | none | private)
- shared(list,) - supported by parallel construct only
- private(list,)
- firstprivate(list,)
- lastprivate(list,) - supported by loop & sections constructs only
- reduction(<op>:list,)
- copyprivate(list,) - supported by single construct only
- threadprivate - its own directive

```c
#pragma omp threadprivate(list,)
!$omp threadprivate(list,)
```

- copyin(list,) - supported by parallel construct only
• **private(list,)**
  - Initialized value of variable(s) is undefined

• **firstprivate(list,)**
  - Initialized private variables with value at time of fork to the master's value

• **copyin(list,)**
  - Initialize private variables with the value of master's list

• **threadprivate(list,)**
  - Provides for the initialized of private variables that are treated as global variables inside of each thread
    - **static** variables in C/C++
    - **COMMON** blocks in Fortran
• Variables in `list` are technically shared

• `copyprivate(list,)`
  - Used by `single` to pass list to corresponding private vars in the other threads

• `lastprivate(list,)`
  - Vars in `list` will be assigned the last value assigned to it by a thread
  - Supported by `loop` & `sections` construct

• `reduction(<op>:list,)`
  - Aggregates vars in list using the defined operation
  - Supported by `parallel`, `loop`, & `sections` constructs
  - `<op>` must be an actual operator or an intrinsic function
private & shared in that Simple OpenMP Example

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

int main (int argc, char *argv[]) {
    int tid, numt;
    numt = omp_get_num_threads();
    #pragma omp parallel private(tid) shared(numt)
    {
        tid = omp_get_thread_num();
        printf("hi, from %d\n", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!\n",numt);
        }
    }
    return 0;
}
```

**Output using 4 threads:**

hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!

Note, thread order not guaranteed!
• OpenMP uses a “relaxed consistency” model
• In contrast to “sequential consistency”
• Cores may have out of date values in their cache
• Most constructs imply a “flush” of each thread's cache
• Treated as a memory “fence” by compilers when it comes to reordering operations
• OpenMP provides an explicit flush directive

```
#pragma flush (list,)
!$OMP FLUSH(list,)
```
• **Explicit** sync points are enabled with a `barrier`:
  
  ```
  #pragma omp barrier
  !$omp barrier
  ```

• **Implicit** sync points exist at the end of:
  
  - `parallel, for, do, sections, single, WORKSHARE`

• Implicit barriers can be turned off with, "nowait"

• There is no barrier associated with:
  
  - `critical, atomic, master`

• Explicit barriers must be used if this is required
An explicit **barrier** in that Simple OpenMP Example

### C/C++

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

int main (int argc, char *argv[]) {
    int tid, numt;
    numt = omp_get_num_threads();
    #pragma omp parallel private(tid) shared(numt)
    {
        tid = omp_get_thread_num();
        printf("hi, from %d\n", tid);
        #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!\n",numt);
        }
    }
    return 0;
}
```

### F90

```fortran
program hello90
    use omp_lib
    integer:: id, numt
    numt = omp_get_num_threads()
    !$omp parallel private(id) shared(numt)
    tid = omp_get_thread_num()
    write (*,*) 'hi, from', tid
    !$omp barrier
    if ( tid == 0 ) then
        write (*,*) numt,'threads say hi!'
    end if
    !$omp end parallel
end program
```

**Output using 4 threads:**

```
hi, from 3
hi, from 0
hi, from 2
hi, from 1
<barrier>
4 threads say hi!
```
A Guide to OpenMP

Trace of The Execution

B = wait for all threads @ barrier before progressing further.

F

#pragma omp barrier

0
hi, from 0

B

0 == 0
5 threads say hi!

0

1
hi, from 1

B

1 != 0

1

2
hi, from 2

B

2 != 0

2

3
hi, from 3

B

3 != 0

3

J

F

#pragma omp barrier
The reduction Clause

- Supported by parallel and worksharing constructs
  - parallel, for, do, sections
- Creates a private copy of a shared var for each thread
- At the end of the construct containing the reduction clause, all private values are reduced into one using the specified operator or intrinsic function

```c
#pragma omp parallel reduction(+:i)
!$omp parallel reduction(+:i)
```
Trace of a variable reduction

\[
\text{reduction(+:i)} \quad i = i_0 + i_1 + i_2 + i_3 + i_4
\]
Valid Operations for the reduction Clause

- Reduction operations in C/C++:
  - Arithmetic: + - * /
  - Bitwise: & ^ |
  - Logical: && ||

- Reduction operations in Fortran
  - Equivalent arithmetic, bitwise, and logical operations
  - min, max

- User defined reductions (UDR) is an area of current research

- Note: initialized value matters!
Nested parallel Constructs

- Can be nested, but specification makes it optional
  - `OMP_NESTED={true,false}`
  - `OMP_MAX_ACTIVE_LEVELS={1,2,..}`
  - `omp_{get,set}_nested()`
  - `omp_get_level()`
  - `omp_get_ancestor_thread_num(level)`

- Each encountering thread becomes the master of the newly forked team
- Each subteam is numbered 0 through N-1
- Useful, but still incurs parallel overheads
The Uniques of Thread Numbers in Nesting

Nest Level 0

0 __ompc_fork(..)

Nest Level 1

__ompc_fork(..) __ompc_fork(..) __ompc_fork(..)

Nest Level 2

2 1 0 2 1 0 2 1 0
• Threads share work in shared memory.
• OpenMP provides “work sharing” constructs
• These constructions include:
  - for, DO
  - sections
  - WORKSHARE (Fortran only)
  - single, master
The loop constructs distribute iterations among threads according to some schedule (default is static)

Among first constructs used when introducing OpenMP

The clauses supported by the loop constructions are: private, firstprivate, lastprivate, reduction, schedule, order, collapse, nowait

The loop's schedule refers to the runtime policy used to distribute work among the threads.
OpenMP Parallelizes Loops by Distributing Iterations to Each Thread

int i;
#pragma omp for
for (i=0;i <= 99; i++) {
    // do stuff
}
for (i=0;i <= 33; i++) {
    // do stuff
}
for (i=34;i <= 67; i++) {
    // do stuff
}
for (i=68;i <= 99; i++) {
    // do stuff
}
#include <stdio.h>
#include <omp.h>
#define N 100

int main(void)
{
    float a[N], b[N], c[N];
    int i;
    omp_set_dynamic(0);    // ensures use of all available threads
    omp_set_num_threads(20);  // sets number of all available threads to 20
    /* Initialize arrays a and b. */
    for (i = 0; i < N; i++)
    {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
    /* Compute values of array c in parallel. */
    #pragma omp parallel shared(a, b, c) private(i)
    {
        #pragma omp for [nowait]
        for (i = 0; i < N; i++)
        {
            c[i] = a[i] + b[i];
        }
        printf ("%f\n", c[10]);
    }
}
#include <stdio.h>
#include <omp.h>
#define N 100

int main(void)
{
    float a[N], b[N], c[N];
    int i;
    omp_set_dynamic(0);           // ensures use of all available threads
    omp_set_num_threads(20);      // sets number of all available threads to 20
    /* Initialize arrays a and b. */
    for (i = 0; i < N; i++)
    {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }
    /* Compute values of array c in parallel. */

    #pragma omp parallel shared(a, b, c) private(i)
    {
        #pragma omp for [nowait]
        for (i = 0; i < N; i++)
            c[i] = a[i] + b[i];
        printf ("%f\n", c[10]);
    }
}
PROGRAM VECTOR_ADD
USE OMP_LIB
PARAMETER (N=100)
INTEGER N, I
REAL A(N), B(N), C(N)
CALL MP_SET_DYNAMIC (.FALSE.)  
!ensures use of all available threads
CALL OMP_SET_NUM_THREADS (20)  
!sets number of available threads to 20

! Initialize arrays A and B.
  DO I = 1, N
    A(I) = I * 1.0
    B(I) = I * 2.0
  ENDDO

! Compute values of array C in parallel.
!$OMP PARALLEL SHARED(A, B, C), PRIVATE(I)
 !$OMP DO
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO
!$OMP END DO [nowait]
 !... some more instructions
!$OMP END PARALLEL
PRINT *, C(10)
END
PROGRAM VECTOR_ADD
USE OMP_LIB
PARAMETER (N=100)
INTEGER N, I
REAL A(N), B(N), C(N)
CALL MP_SET_DYNAMIC (.FALSE.) !ensures use of all available threads
CALL OMP_SET_NUM_THREADS (20) !sets number of available threads to 20

! Initialize arrays A and B.
DO I = 1, N
   A(I) = I * 1.0
   B(I) = I * 2.0
ENDDO

! Compute values of array C in parallel.
!$OMP PARALLEL SHARED(A, B, C), PRIVATE(I)
!$OMP DO
DO I = 1, N
   C(I) = A(I) + B(I)
ENDDO
!$OMP END DO [nowait]
! ... some more instructions
!$OMP END PARALLEL
PRINT *, C(10)
END
Parallel Loop Scheduling

- Scheduling refers to how iterations are assigned to a particular thread;
- There are 5 types:
  - *static* – each thread is able to calculate its chunk
  - *dynamic* – first come, first serve managed by runtime
  - *guided* – decreasing chunk sizes, increasing work
  - *auto* – determined automatically by compiler or runtime
  - *runtime* – defined by `OMP_SCHEDULE` or `omp_set_schedule`
- Limitations
  - only one schedule type may be used at for a given loop
  - the chunk size applies to *all* threads
Parallel Loop Scheduling - Example

Fortran

```fortran
!$OMP PARALLEL SHARED(A, B, C) PRIVATE(I)
!$OMP DO SCHEDULE (DYNAMIC, 4)
   DO I = 1, N
      C(I) = A(I) + B(I)
   ENDDO
!$OMP END DO [nowait]
!$OMP END PARALLEL
```

C/C++

```c
#pragma omp parallel shared(a, b, c) private(i)
{
#pragma omp for schedule (guided, 4) [nowait]
   for (i = 0; i < N; i++)
      c[i] = a[i] + b[i];
}
```
The **ordered** Clause and **ordered** construct

- An **ordered** loop contains code that must execute in serial order
- The ordered code must be inside of an **ordered** construct

```c
#pragma omp parallel shared(a, b, c) private(i)
{
#pragma omp for ordered
  for (i = 0; i <= 99; i++) {
    // do a lot of stuff concurrently
    #pragma omp ordered
    {
      a = i * (b + c);
      b = i * (a + c);
      c = i * (a + b);
    }
  }
}
```
The collapse Clause

- Specifies how many loop levels are to be associated with the loop construct
- The n levels are collapsed into a combined iteration space
- The schedule applies the entire iteration space as usual

```c
#pragma omp parallel shared(a, b, c) private(i)
{
    #pragma omp for schedule(dynamic,4) collapse(2)
    for (i = 0; i <= 99; i++) {
        for (j = i; j <= 99; j++) {
            // do stuff for each i,j
        }
    }
}
```
• Provides for parallel execution of code using F90 array syntax
• The clauses supported by the WORKSHARE construct are: private, firstprivate, copyprivate, nowait
• There is an implicit barrier at the end of this construct
• Valid Fortran code enclosed in a workshare construct:
  - Array & scalar variable assignments
  - FORALL statements & constructs
  - WHERE statements & constructs
  - User defined functions of type ELEMENTAL
  - OpenMP atomic, critical, & parallel
The *sections* construct defines code that is to be executed once by exactly one thread

- A barrier is implied

- Supported clauses include: `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`
```c
#include <stdio.h>
#include <omp.h>

int square(int n){
    return n*n;
}

int main(void){
    int x, y, z, xs, ys, zs;
    omp_set_dynamic(0);
    omp_set_num_threads(3);
    x = 2; y = 3; z = 5;

    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                xs = square(x);
                printf("id = %d, xs = %d\n", omp_get_thread_num(), xs);
            }
            #pragma omp section
            {
                ys = square(y);
                printf("id = %d, ys = %d\n", omp_get_thread_num(), ys);
            }
            #pragma omp section
            {
                zs = square(z);
                printf("id = %d, zs = %d\n", omp_get_thread_num(), zs);
            }
        }
    }
    return 0;
}
```

A section Construct Example

http://developers.sun.com/solaris/articles/studio_openmp.html
A section Construct Example

```c
#pragma omp sections
{
    #pragma omp section
    { xs = square(x);
      printf("id = %d, xs = %d\n", omp_get_thread_num(), xs);
    }
    #pragma omp section
    { ys = square(y);
      printf("id = %d, ys = %d\n", omp_get_thread_num(), ys);
    }
    #pragma omp section
    { zs = square(z);
      printf("id = %d, zs = %d\n", omp_get_thread_num(), zs);
    }
}
```

thread 0

thread 1

thread 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
t=0 |   |
t=1 |   |

Time
Combined parallel Constructs

- parallel may be combined with the following:
  - parallel, for, do, sections, WORKSHARE

- Semantics are identical to usage already discussed

```c
!$OMP PARALLEL DO SHARED(A, B, C) PRIVATE(I)
!$OMP& SCHEDULE(DYNAMIC,4)
  DO I = 1, N
    C(I) = A(I) + B(I)
  ENDDO
!$OMP END PARALLEL DO
```

```c
#pragma omp parallel for shared(a, b, c) private(i) schedule (guided,4) 
{ 
    for (i = 0; i < N; i++)
        c[i] = a[i] + b[i]; 
} 
```
Singling Out Threads with $\texttt{master}$ and $\texttt{single}$ Constructs

- Code inside of a $\texttt{master}$ construct will only be executed by the master thread.
- There is NO implicit barrier associated with $\texttt{master}$; other threads ignore it.

\begin{verbatim}
$\texttt{OMP MASTER}
  ... do stuff
$\texttt{OMP END MASTER}
\end{verbatim}

- Code inside of a $\texttt{single}$ construct will be executed by the first thread to encounter it.
- A $\texttt{single}$ construct contains an implicit barrier that will respect $\texttt{nowait}$.

\begin{verbatim}
$\texttt{OMP SINGLE}
  ... do stuff
$\texttt{OMP END SINGLE [nowait]}
\end{verbatim}
• Tasks were added in 3.0 to handle dynamic and unstructured applications
  – Recursion
  – Tree & graph traversals
• OpenMP's execution model based on threads was redefined
• A thread is considered to be an *implicit* task
• The *task* construct defines singular tasks explicitly
• Less overhead than nested *parallel* regions
Threads are now Implicit Tasks

CPU 0
- Implicit task 0

CPU 1
- Implicit task 1

CPU 2
- Implicit task 2

CPU 3
- Implicit task 3

Process 0

Main Memory

p0
The task Construct

- Clauses supported are: *if*, *default*, *private*, *firstprivate* *shared*, *tied/untied*
- By default, all variables are *firstprivate*
- Tasks can be nested syntactically, but are still asynchronous
- The *taskwait* directive causes a task to wait until all its children have completed
Each Thread Conceptually Has Both a **tied** & **untied** queue
struct node {
    struct node *left;
    struct node *right;
};

extern void process(struct node *);

void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);
}
RECURSIVE SUBROUTINE traverse ( P )
    TYPE Node
        TYPE(Node), POINTER :: left, right
    END TYPE Node
    TYPE(Node) :: P
    IF (associated(P%left)) THEN
        !$OMP TASK ! P is firstprivate by default
        call traverse(P%left)
        !$OMP END TASK
    ENDIF
    IF (associated(P%right)) THEN
        !$OMP TASK ! P is firstprivate by default
        call traverse(P%right)
        !$OMP END TASK
    ENDIF
    CALL process ( P )
END SUBROUTINE
• Some code must be executed by one thread at a time
• Effectively serializes the threads
• Also called critical sections
• OpenMP provides 3 ways to achieve mutual exclusion
  – The **critical** construct encloses a critical section
  – The **atomic** construct enclose updates to shared variables
  – A low level, general purpose locking mechanism
The critical Construct

- The critical construct encloses code that should be executed by all threads, just in some serial order.

```cpp
#pragma omp parallel
{
  #pragma omp critical
  {
    // some code
  }
}
```

- The effect is equivalent to a lock protecting the code.
#pragma omp parallel shared(a, b, c) private(i)
{
    #pragma omp critical
    {
        //
        // do stuff (one thread at a time)
        //
    }
}

Note: Encountering thread order not guaranteed!
The Named critical Construct

- Names may be applied to critical constructs.

```c
#pragma omp parallel
{  
#pragma omp critical(a)
{  
    // some code
  }
#pragma omp critical(b)
{  
    // some code
  }
#pragma omp critical(c)
{  
    // some code
  }
}
```

- The effect is equivalent to using a different lock for each section.
A Guide to OpenMP

A Named critical Construct Example

```c
#include <stdio.h>
#include <omp.h>
#define N 100

int main(void)
{
  float a[N], b[N], c[3];
  int i;
  // Initialize arrays a and b. */
  for (i = 0; i < N; i++)
    { a[i] = i * 1.0 + 1.0;
      b[i] = i * 2.0 + 2.0;
    }
  /* Compute values of array c in parallel. */
  #pragma omp parallel shared(a, b, c) private(i)
  {
    #pragma omp critical(a)
    { for (i = 0; i < N; i++)
      C[0] += a[i] + b[i];
      printf("%f\n",c[0]);
    }
    #pragma omp critical(b)
    { for (i = 0; i < N; i++)
      c[1] += a[i] + b[i];
      printf("%f\n",c[1]);
    }
    #pragma omp critical(c)
    { for (i = 0; i < N; i++)
      c[2] += a[i] + b[i];
      printf("%f\n",c[2]);
    }
  }
}
```
A Named critical Construct Example

```c
#pragma omp critical(a)
{
    // some code
}
#pragma omp critical(b)
{
    // some code
}
#pragma omp critical(c)
{
    // some code
}
```

Note:
Encountering thread order not guaranteed!
The `atomic` Construct for Safely Updating Shared Variables

- Protected writes to shared variables
- Lighter weight than using a `critical` contract

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

int main(void) {
    int count = 0;
    #pragma omp parallel shared(count)
    {
        #pragma omp atomic
        count++;
    }
    printf("Number of threads: %d\n",count);
}
```

Note:
Encountering thread order not guaranteed!
Locks in OpenMP

- `omp_lock_t`, `omp_lock_kind`
- Threads set/unset locks
- Nested locks can be set multiple times by the same thread before releasing them
- More flexible than critical construct
Using Locks in OpenMP

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

int main()
{
    int x;
    omp_lock_t lck;
    omp_init_lock (&lck);
    omp_set_lock (&lck);
    x = 0;

    #pragma omp parallel shared (x)
    {
        #pragma omp master
        {
            x = x + 1;
            omp_unset_lock (&lck);
        }
    /* Some more stuff. */
    }

    omp_destroy_lock (&lck);
}
```
#include <omp.h>

typedef struct {
    int a, b; omp_nest_lock_t lck; } pair;

int work1();  
int work2();  
int work3();

void incr_a(pair *p, int a) {
    /* Called only from incr_pair, no need to lock. */
    p->a += a;
}

void incr_b(pair *p, int b) {
    /* Called both from incr_pair and elsewhere, */
    /* so need a nestable lock. */
    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b) {
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void a45(pair *p) {
    #pragma omp parallel sections
    {
        #pragma omp section
            incr_pair(p, work1(), work2());
        #pragma omp section
            incr_b(p, work3());
    }
}
• In fixed form Fortran OpenMP directives can hide behind the following “sentinals”

    !$ [OMP], c$ [OMP], *$ [OMP]

• Free form requires “!$”

• Sentinals can enable conditional compilation

    !$ omp_set_num_threads(n)

• Fortran directives should start in column 0

• Long directive continuations take a form similar to:

    !$OMP PARALLEL DEFAULT(NONE)
    !$OMP & SHARED(INP,OUTP,BOXL,TEMP,RHO,NSTEP,TSTEP,X,Y,Z,VX,VY,VZ,BOXL)
    !$OMP & SHARED(XO,YO,ZO,TSTEP,V2T,VXT,VYT,VZT,IPRINT,ISTEP,ETOT,ERUN)
    !$OMP & SHARED(FX,FY,FZ,PENER)
    !$OMP & PRIVATE(I)
• No line continuations, entire directive on single line
• No conditional compilation sentinals, use “#ifdef”, etc
• Coding style

```c
int main () {
    ...
    #pragma parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            { xs = square(x);
                printf ("id = %d, xs = %d\n", omp_get_thread_num(), xs);
            }
            #pragma omp section
            { ys = square(y);
                printf ("id = %d, ys = %d\n", omp_get_thread_num(), ys);
            }
        }
        return 0; /* end main */
    }
```

General Programming Tips

- Minimize parallel constructs
- Use combined constructs, if it doesn't violate the above
- Minimize shared variables, maximize private
- Minimize barriers, but don't sacrifice safety
- When inserting OpenMP into existing code
  - Use a disciplined, iterative cycle – test against serial version
  - Use barriers liberally
  - Optimize OpenMP & asynchronize last
- When starting from scratch
  - Start with an optimized serial version
• Won't cover directly, but they exist for:
  • Pipelining computations
  • Effectively using I/O (especially in a pipelined context)
  • Creating user defined reductions (UDR) (e.g., for divide & conquer algorithms, map-reduce type applications)
  • Interleaving N units of critical work with M threads to minimize idle time
  • Effective use of nested parallelism and tasks for unbalanced and dynamical work loads
  • ...many more
Other Issues Not Covered Directly

- Profiling & optimizations
- Debugging & troubleshooting techniques
- Real world OpenMP
- OpenMP in hybrid contexts
The Future of OpenMP

- It's not going anywhere; vendor buy-in is as strong as ever
- Big 3:
  - Refinement to tasking model (scheduling, etc)
    - Error handling
    - Accelerators
- Scaling
  - Thousands of threads
  - Data locality
  - More efficient synchronization constructs & implementations
- Remaining relevant
Additional Resources

- http://www.cs.uh.edu/~hpctools
- http://www.compunity.org
- http://www.openmp.org
  - Specification 3.0
- “Using OpenMP”, Chapman, et. al.

Covers through 2.5