Presented By:
Dr. Barbara Chapman

Created By:
Brett Estrade

HPCTools Group
University of Houston
Department of Computer Science

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

- process 0
- process 1
- process 2
- process 3
Multiple Processes on this System – this is MPI's model

- process 0
- cpu 0
- receive
- send

- process 1
- cpu 1
- receive
- send

- process 2
- cpu 2
- receive
- send

- process 3
- cpu 3
- receive
- send
A Single Process with Multiple Threads on this System

Main Memory

- CPU 0
  - Thread 0
- CPU 1
  - Thread 1
- CPU 2
  - Thread 2
- CPU 3
  - Thread 3

Process 0
Here, thread 0 is on CPU 0; at the fork, 3 new threads are created and are distributed to the remaining 3 CPUs.
An Example of Sharing Memory

- Process 0
  - CPU 0
    - Thread 0
  - CPU 1
    - Thread 1
  - CPU 2
    - Thread 2
  - CPU 3
    - Thread 3

Main Memory

int a
Thread 0 writes integer \( a \).
Thread 1 reads integer $a$. 

An Example of Sharing Memory
An Example of Sharing Memory

Thread 2 reads integer `a`.
Thread 3 reads integer a.
Notes on the Example of Sharing Memory

- Threads execute their code independently
- So usually Read/Write order matters
- If the order is not specified in some way, this could represent a (data) *race condition*
- Race conditions introduce *non-determinism* (not good)

- Threaded programs can be extremely difficult to debug
- Proper precautions must be made to eliminate data races
What is OpenMP?

- An industry standard for shared memory parallel programming
  - OpenMP Architecture Review Board
  - AMD, Intel, IBM, HP, Microsoft, Sun/Oracle, Fujitsu, NEC, Texas Instruments, PGI, CAPS; LLNL, ORNL, ANL, NASA, cOMPunity,..
  - http://www.openmp.org/
- A set of directives for describing parallelism in an application code
- A user-level API and runtime environment
- A widely supported standard set of parallel programming pragmas with bindings for Fortran, C, & C++
- A community of active users & researchers
# The Anatomy of an OpenMP Program

```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;
}
```

- **parallel (fork) directive**
- **structured parallel block**
- **runtime function**
- **clauses**
- **directive (thread barrier)**
The timeline of the OpenMP Standard Specification

- **1998**: OpenMP 1.0 for C/C++
- **2000**: OpenMP 2.0 for Fortran
- **2002**: OpenMP 2.0 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
- Much of serial code can be left untouched (most of the time)
- 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 (can help avoid false sharing, etc)
- It can be fun to use (immediate gratification)
What Does OpenMP Provide?

- More abstract programming interface than low level thread libraries
- User specifies strategy for parallel execution, but not all the details
- Directives, written as structured comments
- A *runtime* library that manages execution dynamically
- Control via environment variables & the *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>
OpenMP Research Activities

- A lot of research has gone into the OpenMP 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
- Applications work with new features
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 *structured comments*

  **C/C++:**
  
  ```
  #pragma omp <directive> <clauses>
  ```

  **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
• Order of clauses has no bearing on effect
Summary of OpenMP's Directives and Key Clauses

- Forking Threads
  - `parallel`

- Distributing Work
  - `for` (C/C++)
  - `DO` (Fortran)
  - `sections/section`
  - `WORKSHARE` (Fortran)

- Singling Out Threads
  - `single`
  - `Master`

- Mutual Exclusion
  - `critical`
  - `atomic`

- Synchronization
  - `barrier`
  - `flush`
  - `ordered`
  - `taskwait`

- Asynchronous Tasking
  - `task`

- Data Environment
  - `shared`
  - `private`
  - `threadprivate`
  - `reduction`
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

- Doxygen docs of OpenUH's OpenMP RTL, libopenmp
- The Doxygen call graph for __ompc_fork in libopenmp/omp_threads.c

http://www2.cs.uh.edu/~estrabd/OpenUH/r593/html-libopenmp/
Useful OpenMP Environment Variables

- OMP_NUM_THREADS
- OMP_SCHEDULE
- OMP_DYNAMIC
- OMP_STACKSIZE
- OMP_NESTED
- OMP_THREAD_LIMIT
- OMP_MAX_ACTIVE_LEVELS
- OMP_WAIT_POLICY
Execution environment routines; e.g.,
- `omp_{set,get}_num_threads`
- `omp_{set,get}_dynamic`
- Each envvar has a corresponding get/set

Locking routines (*generalized mutual exclusion*); e.g.,
- `omp_{init,set,test,unset,destroy}_lock`
- `omp_{...}_nest_lock`

Timing routines; e.g.,
- `omp_get_wtime`
- `omp_get_wtick`
How Is OpenMP Implemented?

• A compiler translates OpenMP
  – It processes the directives and uses them to create explicitly multithreaded code
  – It translates the rest of the Fortran/C/C++ more or less as usual
  – Each compiler has its own strategy
• The generated code makes calls to the runtime library
• We have already seen what the runtime is typically responsible for
  – The runtime library (RTL) also implements OpenMP's user-level runtime routines
  – Each compiler has its own custom RTL
How Is an OpenMP Program Compiled? Here's How OpenUH does it.

Parse into Very High WHIRL

High level, serial program optimizations

Put OpenMP constructs into standard format

Loop level serial program optimizations

Transform OpenMP into threaded code

Optimize, now, potentially, threaded code

Output native instr. or interface with native compiler via source to source translation

Source code w/ OpenMP directives

Source code with runtime library calls

A Native Compiler

Object files

Linking

Executables

A Portable OpenMP Runtime library

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

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

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    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()`

```c
static void __omprg_main_1(__ompv_gtid_a, __ompv_slink_a) _INT32 __ompv_temp_gtid;
_REGISTER __ompv_gtid_a;
_UINT64 __ompv_slink_a;
{
    register _INT32 w2c_comma;
    _UINT64 temp_slink_sym0;
    _INT32 __ompv_temp_gtid;
    _INT32 __mplocal_my_id;

    /*Begin_of_nested_PU(s)*/
    _temp_slink_sym0 = __ompv_slink_a;
    __ompv_temp_gtid = __ompv_gtid_a;
    w2c_comma = omp_get_thread_num();
    __mplocal_my_id = w2c_comma;
    printf("hello from %d\n", __mplocal_my_id);
    return;
} /* __omprg_main_1 */

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
", my_id);
        __ompc_end_serialized_parallel();
    }
    return 0;
} /* main */
```

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

`__omprg_main_1`'s frame pointer

serial version

Nobody wants to code like this, so we let the compiler and runtime do most of this tedious work!
Programming with OpenMP 3.0
The parallel Construct

- Where the “fork” occurs (e.g., `__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 (lexically) outside parallel region, but dynamically 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\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!
The Fortran Version

```fortran
program hello90
use omp_lib
integer:: tid, numt
numt = omp_get_num_threads()
!$omp parallel private(tid) shared(numt)
  tid = omp_get_thread_num()
  write (*,*) 'hi, from', tid
!$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!
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\n", tid);
    #pragma omp barrier
        if ( tid == 0 ) {
            printf("%d threads say hi!\n",numt);
        }
    }
    return 0;
}
```

F90

```f90
program hello90
use omp_lib
integer:: tid, 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!
A Guide to OpenMP

Trace of The Execution

all threads call printf

0

B

hi, from 0

4 threads say hi!

0 == 0

join

fork

0

B

1

hi, from 1

1 != 0

1

B

2

hi, from 2

2 != 0

2

B

3

hi, from 3

3 != 0

3

B

only thread with tid == 0 does this

0

fork

4 threads say hi!

0

B

0

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

other threads wait

2

B

3

J

F

B

F

B

wait for all threads @ barrier before progressing further.
Controlling the Fork at Runtime

- 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** - a standalone directive, not a clause

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

- **copyin** (list, ) - supported by parallel construct only
Private Variable Initialization

- `private(list,)`
  - Initialized value of variable(s) is undefined

- `firstprivate(list,)`
  - private variables are initialized with value of corresponding variable from master thread at time of fork

- `copyin(list,)`
  - Copy value of master's threadprivate variables to corresponding variables of the other threads

- `threadprivate(list,)`
  - Global lifetime objects are replicated so each thread has its own copy
    - `static` variables in C/C++, `COMMON` blocks in Fortran
Getting Data Out

- **copyprivate(list,)**
  - Used by single thread to pass list values to corresponding private vars in the other, non-executing, threads

- **lastprivate(list,)**
  - vars in list will be given 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/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:: tid, numt
numt = omp_get_num_threads()
!$omp parallel private(tid) shared(numt)
    tid = omp_get_thread_num()
    write (*,*) 'hi, from', tid
!$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!
A Guide to OpenMP

Memory Consistency Model

- OpenMP uses a “relaxed consistency” model
- In contrast especially 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

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

```c
#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 when ordering is required and otherwise not guaranteed
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:: tid, 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!
```
Trace of The Execution

0
hi, from 0

1
hi, from 1

2
hi, from 2

3
hi, from 3

B

4 threads say hi!

0 == 0

1 != 0

2 != 0

3 != 0

#pragma omp barrier

B = wait for all threads @ barrier before progressing further.
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)
```
A reduction Example

C/C++

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

int main (int argc, char *argv[]) {
    int t, i;
    i = 0;
    #pragma omp parallel private(t) reduction(+,i)
    {
        t = omp_get_thread_num();
        i = t + 1;
        printf("hi, from %d\n", t);
    #pragma omp barrier
        if ( t == 0 ) {
            int numt = omp_get_num_threads();
            printf("%d threads say hi!\n",numt);
        }
    }
    printf("i is reduced to %d\n",i);
    return 0;
}
```

F90

```fortran
program hello90
    use omp_lib
    integer:: t, i, numt
    i = 0
    !$omp parallel private(t) reduction(+:i)
    t = omp_get_thread_num()
    i = t + 1;
    write (*,*) 'hi, from', t
    !$omp barrier
    if ( t == 0 ) then
        numt = omp_get_num_threads()
        write (*,*) numt,'threads say hi!'
    end if
    !$omp end parallel
    write (*,*) 'i is reduced to ', i
end program
```

Output using 4 threads:

```
  hi, from 3
  hi, from 0
  hi, from 2
  hi, from 1
  4 threads say hi!
i is reduced to 10
```
A shared variable is initialized at the beginning of the construct.

Each thread writes to its private variable, and the reduction occurs at the end of the construct.

The final value of the variable is available after the end of the construct.
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

- Specification makes this feature 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 Uniqueness of Thread Numbers in Nesting

Thread numbers are not unique; paths to each thread are.

Because of the tree structure, each thread can be uniquely identified by its full path from the root of its sub-tree;

This path tuple can be calculated in $O(\text{level})$ using `omp_get_level` and `omp_get_ancestor_thread_num` in combination.
• Threads “share” work, access data in shared memory
• OpenMP provides “work sharing” constructs
• These specify the work to be distributed among the threads and state how the work is assigned to them
• Data race conditions are illegal
• They include:
  - loops (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 constructs 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. The code snippet below demonstrates this:

```c
int i;
#pragma omp for
for (i=0; i <= 99; i++) {
    // do stuff
}
```

This code is divided into three parts, each running on a different thread:

- **Thread 0**: for (i=0; i <= 33; i++) {
  // do stuff
}
  - i = 0 thru 33

- **Thread 1**: for (i=34; i <= 67; i++) {
  // do stuff
}
  - i = 34 thru 67

- **Thread 2**: for (i=68; i <= 99; i++) {
  // do stuff
}
  - i = 68 thru 99
```c
#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]);
    }
}
```

http://developers.sun.com/solaris/articles/studio_openmp.html
```c
#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]);
}
```

"nowait" is optional
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
!$OMP END DO [nowait]
!$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
The scheduling is a description that states which loop iterations are assigned to a particular thread.

There are 5 types:

- **static** – each thread calculates its own chunk
- **dynamic** – “first come, first served”, managed by runtime
- **guided** – dynamic, with decreasing chunk sizes
- **auto** – determined automatically by compiler or runtime
- **runtime** – defined by OMP_SCHEDULE or omp_set_schedule

Limitations

- only one schedule type may be used 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 \textit{ordered} Clause and \textit{ordered} construct

- An \textit{ordered} loop contains code that must execute in serial order
- No clauses are supported
- The ordered code must be inside an \textit{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

- n levels are collapsed into a combined iteration space
- The schedule and ordered constructs are applied as expected
- E.g.,

```c
#pragma omp parallel shared(a, b, c) private(i)
{
    #pragma omp for schedule(dynamic,4) collapse(2)
    for (i = 0; i <= 2; i++) {
        for (j = 0; j <= 2; j++) {
            // do stuff for each i,j
        }
    }
}
```

- Iterations are distributed as \((i \times j)\) n-tuples, e.g., \(<i, j>:\)
  - \(<0,0>; <0,1>; <0,2>; <1,0>; <1,1>; <1,2>; <2,0>; <2,1>; <2,2>;\)
• 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 shared(xs,ys,zs)
{
#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

```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

t=0 t=1

Time
Combined parallel Constructs

- **parallel** may be combined with the following:
  - `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 master and single Constructs

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

```c
!$OMP MASTER
  ... do stuff
!$OMP END MASTER
```

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

```c
#pragma omp single [nowait] [copyprivate(list,)]
  {
    ... do stuff
  }
```

```fortran
!$OMP SINGLE
  ... do stuff
!$OMP END SINGLE [nowait] [copyprivate(list,)]
```

C/C++

Fortran

note position of clauses in Fortran
The task Construct

- 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
Each Thread Conceptually Has Both a tied & untied queue

- cpu 0
  - Implicit task 0
  - tied (private)
  - untied (public)

- cpu 1
  - Implicit task 1
  - tied (private)
  - untied (public)

- cpu 2
  - Implicit task 2
  - tied (private)
  - untied (public)

- cpu 3
  - Implicit task 3
  - tied (private)
  - untied (public)
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
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 execution of the code
• Such code regions often 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.

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

- The effect is equivalent to a lock protecting the code.
A critical Construct Example

```c
#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 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
- Typically lighter weight than a critical construct

```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`
- `omp_nest_lock_t`, `omp_nest_lock_kind`
- Threads set/unset locks
- Nested locks can be set multiple times by the same thread before releasing them
- More flexible than `critical` and `atomic` constructs – the behavior of both is defined by behavior of locks
  - E.g., “named” `critical` sections behave as if each `critical` section used a different lock variable
- Locking routines imply a memory `flush`
Single Locks vs Nested Locks

- Single locks may be set once and only once
- Single locks must be unset by the owning thread, but only once
- Nested locks may be set multiple times, but only by the thread initially setting it
- Nested lock must be unset for each time it is set
- Nested locks are not *unlocked* until the lock count has been decremented to 0
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

/*
   this example mimics the behavior of the atomic
   construct used to safely update locked variables
*/

int main()
{
    int x;
    omp_lock_t lck;
    /* master thread initialized lock var before fork*/
    omp_init_lock(&lck);
    x = 0;
    #pragma omp parallel shared (x,lck)
    {
        omp_set_lock(&lck);  /* threads wait to acquire lock */
        x = x+1;            /* thread holding lock updates x */
        omp_unset_lock(&lck); /* thread releases lock */
    }
    /* master thread destroy lock after join */
    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());
    }
}
• Directives are case insensitive
• In fixed form Fortran OpenMP directives can hide behind the following “sentinals”

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

• Free form requires “!$”
• Sentinals can enable conditional compilation

```fortran
!$ omp_set_num_threads(n)
```

• Fortran directives should start in column 0
• Long directive continuations take a form similar to:

```fortran
!$OMP PARALLEL DEFAULT(NONE)
!$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)
```
C/C++ Programming Tips

- No line continuations, entire directive on single line
- Directives are case sensitive
- No conditional compilation sentinals, use "#ifdef", etc
- Code 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
The Future of OpenMP

- Vendor buy-in and R&D support is as strong as ever
- Must remain relevant
- Active areas of research:
  - Refinement to tasking model (scheduling, etc)
  - User defined reductions (UDRs)
  - Accelerators & heterogeneous environments
  - Error handling
  - Hybrid models
- Scaling issues being addressed:
  - Thousands of threads
  - Data locality
  - More flexible & efficient synchronization
A Practical Example – Calculating \( \pi \)

static long num_steps = 100000;
double step;

void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}

Mathematically, we know:

\[
\int_{0}^{1} \frac{4.0}{1+x^2} \, dx = \pi
\]

And this can be approximated as a sum of the area of rectangles:

\[
\sum_{i=1}^{N} F(x_i) \Delta x \approx \pi
\]

Where each rectangle has a width of \( \Delta x \) and a height of \( F(x_i) \) at the middle of interval \( i \).
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i, id, nthreads; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x)
    {
        id = omp_get_thread_num();
        #pragma omp single
        nthreads = omp_get_num_threads();
        for (i=id, sum[id]=0.0; i< num_steps; i=i+nthreads){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0; i<nthreads; i++) pi += sum[i] * step;
}
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
  int i, id, nthreads;  double x, pi, sum;
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x, sum)
  {
    id = omp_get_thread_num();
#pragma omp single
    nthreads = omp_get_num_threads();
    for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
    }
#pragma omp critical
    pi += sum * step;
  }
}
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for private(x) reduction(+:sum)
        for (i=0;i<= num_steps; i++){
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
    pi = step * sum;
}
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for private(x) reduction(+:sum)
    for (i=0;i<=num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}

In practice, the number of threads is set using the environmental variable, OMP_NUM_THREADS.

For good OpenMP implementations, reduction is more scalable than a critical construct.
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*
• Brett Estrade
• Dr. Barbara Chapman
• Amrita Banerjee

The latest version of this presentation may be downloaded at:

http://www.cs.uh.edu/~hpctools/OpenMP

All feedback is welcome and appreciated. Please send all feedback to Brett:

estrabdb@cs.uh.edu