OpenMP Programming

Oil & Gas Workshop,
03/04/2015

Barbara Chapman, University of Houston
Deepak Eachempati, University of Houston

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

- An Introduction to OpenMP 3.1
- Quick Overview of OpenMP 4.0
- Outlook
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
What is OpenMP?

- De-facto standard **API** to write shared memory parallel applications in C, C++, and Fortran
  - Recent features go beyond shared memory
- Initial version released end of 1997
  - For Fortran only
  - Subsequent releases for C, C++
- Version 2.5 merged specs for all three languages
- Version 3.1 released July 2011; 4.0 July 2013
OpenMP 4.0
The OpenMP ARB

- OpenMP is maintained by the OpenMP Architecture Review Board (the ARB), which
  - Interprets OpenMP
  - Writes new specifications - keeps OpenMP relevant
  - Works to increase the impact of OpenMP

- Members are organizations - not individuals
  - Current members
    - Permanent: AMD, Convey Computer, Cray, Fujitsu, HP, IBM, Intel, Microsoft, NEC, Nvidia, Oracle, Red Hat, St Microelectronics, Texas Instruments
    - Auxiliary: ANL, BSC, cOMPunity, EPCC, NASA, LANL, ASC/LLNL, ORNL, RWTH Aachen, SNL, TACC, University of Houston

www.openmp.org
How Does OpenMP Work?

- OpenMP provides thread programming model at a “high level”
  - Threads collaborate to perform the computation
  - They communicate by sharing variables
  - They synchronize to order accesses and prevent data conflicts
  - Structured programming is encouraged to reduce likelihood of bugs
- Alternatives:
  - MPI
  - POSIX thread library is lower level
  - Automatic parallelization is higher level (user does nothing)
    - But usually successful on simple codes only

User makes strategic decisions; Compiler figures out details
Role of User

- User inserts directives telling compiler how statements are to be executed
  - what parts of the program are parallel
  - how to assign code in parallel regions to threads
  - what data is private (local) to threads

- User must remove any dependences in parallel parts
  - Or introduce appropriate synchronization

- OpenMP compiler does not check for them!
  - It is up to programmer to ensure correctness
  - Some tools exist to help check this
How is OpenMP Compiled?

- Most Fortran/C compilers today implement OpenMP
  - The user provides the required switch or switches
  - Sometimes this also needs a specific optimization level, so manual should be consulted
  - May also need to set threads’ stacksize explicitly

Examples
- Commercial: -openmp (Intel, Sun, NEC), -mp (SGI, PathScale, PGI), --openmp (Lahey, Fujitsu), -qsmp=omp (IBM) /openmp flag (Microsoft Visual Studio 2005), etc.
- Freeware: gcc, Omni, OdinMP, OMPi, OpenUH, (llvm)

Check information at http://www.openmp.org
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
OpenMP Fork-Join Execution Model

- Execution starts with single thread (the initial / master thread)
- Master thread spawns multiple worker threads as needed, together form a team
- *Parallel region* is a block of code executed by all threads in a team simultaneously

Number of threads in a team may be dynamically adjusted
OpenMP Memory Model

- OpenMP assumes a shared memory
- Threads communicate by sharing variables.
- Synchronization protects data conflicts.
  - Synchronization is expensive.
  - Change how data is accessed to minimize the need for synchronization.

- All threads have access to the same, **globally shared**, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit
Data-Sharing Attributes

- In OpenMP code, data needs to be “labeled”
- There are two basic types:
  - **Shared** – there is only one instance of the data
    - Threads can read and write the data simultaneously unless protected through a specific construct
    - All changes made are visible to all threads
      - But not necessarily immediately, unless enforced ......
  - **Private** - Each thread has a copy of the data
    - No other thread can access this data
    - Changes only visible to the thread owning the data

Data is shared by default
OpenMP Syntax

- Most OpenMP constructs are compiler directives
  - For C and C++, they are pragmas with the form:
    ```c
    #pragma omp construct [clause [clause]…]
    ```
  - For Fortran, the directives may have fixed or free form:
    ```fortran
    *$OMP construct [clause [clause]…]
    C$OMP construct [clause [clause]…]
    !$OMP construct [clause [clause]…]
    ```
- Include file and the OpenMP lib module
  ```
  #include <omp.h>
  use omp_lib
  ```
- Most OpenMP constructs apply to a “structured block”.
  - A block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It’s OK to have an exit() within the structured block.

OpenMP sentinel forms: #pragma omp  !$OMP
Example - The Reduction Clause

```c
sum = 0.0
!
$omp parallel default(none) &
$omp shared(n,x) private(i)
$omp
do reduction (+:sum)
do i = 1, n
    sum = sum + x(i)
end do
$omp end do
$omp end parallel
print *,sum
```

- The result is available after the parallel region
- The compiler generates optimized code that enables threads to collaborate to perform the reduction
- The reduction can be hidden in a function call

```c
reduction ( operator: list )
```
# OpenMP 3.1 Components

## Directives
- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

## Runtime library
- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

## Environment variables
- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

**Pragmas in C / C++**
- (specially written) comments in Fortran
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
A parallel region is a block of code executed by all threads in a team simultaneously

- Threads in team are numbered consecutively, starting from 0; the master thread has thread ID 0
- Thread adjustment (if enabled) is only done before entering a parallel region
- Parallel regions can be nested, but support for this is implementation dependent
- An "if" clause can be used to guard the parallel region; if the condition evaluates to "false", the code is executed serially

OpenMP Team := Master + Workers
Parallel Regions

- You create a team of threads in OpenMP with the "omp parallel" pragma.
- For example, to create a 4 thread parallel region:

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block.

- Each thread calls `pooh(ID,A)` for $ID = 0$ to $3$
Parallel Regions

- Each thread executes the same code redundantly.

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
omp_set_num_threads(4)
pooh(0,A)  pooh(1,A)  pooh(2,A)  pooh(3,A)
printf("all done\n");
```

A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a barrier)
Parallel Regions and The “if” Clause

Active vs. inactive parallel regions.

- An optional if clause causes the parallel region to be active only if the logical expression within the clause evaluates to true.
- An if clause that evaluates to false causes the parallel region to be inactive (i.e. executed by a team of size one).

```c
double A[N];
#pragma omp parallel if(N>1000)
{
    int ID = omp_get_thread_num();
    pooh(ID,A);
}
```
Scope of OpenMP Region

- A parallel region can span multiple source files.

```fortran
foo.f
C$OMP PARALLEL
  call whoami
C$OMP END PARALLEL

bar.f
subroutine whoami
  external omp_get_thread_num
  integer iam, omp_get_thread_num
  iam = omp_get_thread_num()
C$OMP CRITICAL
  print*, 'Hello from ', iam
C$OMP END CRITICAL
return
end
```

Orphaned directives can appear outside a parallel construct.
A Multi-threaded “Hello world” Program

- Each thread prints “hello world” in no specific order

```c
#include "omp.h"

void main()
{
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) 
", ID);
}
}
```

Sample Output:
- hello(1) hello(0) world(1)
- world(0)
- hello (3) hello(2) world(2)
- world(3)
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
Worksharing Constructs

Sequential code

\[
\text{for}(i=0; i<N; i++) \quad \{ \text{a}[i] = \text{a}[i] + \text{b}[i]; \}\]

OpenMP parallel region

\[
\#pragma \text{omp parallel}
\{
\text{int id, i, Nthrds, istart, iend;}
\text{id = omp_get_thread_num();}
\text{Nthrds = omp_get_num_threads();}
\text{istart = id * N / Nthrds; }
\text{iend = (id+1) * N / Nthrds; }
\text{for}(i=\text{istart}; i<\text{iend}; i++) \quad \{ \text{a}[i] = \text{a}[i] + \text{b}[i]; \}\}
\]

OpenMP parallel region and a worksharing for construct

\[
\#pragma \text{omp parallel} \\
\#pragma \text{omp for schedule(static)}
\text{for}(i=0; i<N; i++) \quad \{ \text{a}[i] = \text{a}[i] + \text{b}[i]; \}\]
OpenMP Worksharing Constructs

- Divides the execution of the enclosed code region among the members of the team
- The “for” worksharing construct splits up loop iterations among threads in a team
  - Each thread gets one or more “chunks”

```c
#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++) {
    work(i);
}
```

By default, all threads wait at a barrier at the end of the “OMP for”. Use the “nowait” clause to turn off the barrier.

```c
#pragma omp for nowait
```

“nowait” is useful between two consecutive, independent omp for loops.

*omp do* in Fortran
Example: OMP For

```c
#pragma omp parallel default(none) \shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /*-- End of parallel region --*/

(implied barrier)
```
Loops must be countable. To parallelize this loop, it is necessary to first count the number of iterations and then rewrite it as a for loop. More on this later…
Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- The collapse clause on for/do loop indicates how many loops should be collapsed
- The compiler forms a single loop and parallelizes it

```c
!$omp parallel do collapse(2) ...
do i = il, iu, is
  do j = jl, ju, js
    do k = kl, ku, ks
      ....
      end do
    end do
  end do
$omp end parallel do
```
# OpenMP Schedule Clause

The schedule clause affects how loop iterations are mapped onto threads.

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>Distribute iterations in blocks of size &quot;chunk&quot; over the threads in a round-robin fashion</td>
</tr>
<tr>
<td>dynamic</td>
<td>Fixed portions of work; size is controlled by the value of chunk. When a thread finishes, it starts on the next portion of work</td>
</tr>
<tr>
<td>guided</td>
<td>Same dynamic behavior as &quot;dynamic&quot;, but size of the portion of work decreases exponentially</td>
</tr>
<tr>
<td>auto</td>
<td>The compiler (or runtime system) decides what is best to use; choice could be implementation dependent</td>
</tr>
<tr>
<td>runtime</td>
<td>Iteration scheduling scheme is set at runtime via environment variable <strong>OMP_SCHEDULE</strong> or runtime library call</td>
</tr>
</tbody>
</table>
OpenMP Sections

- Work-sharing construct
- Gives a different structured block to each thread

```c
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
    x_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.
OpenMP Master

- Denotes a structured block executed by the master thread
- The other threads just skip it
  - no synchronization is implied

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
#pragma omp master
    {
        exchange_boundaries();
    }
#pragma omp barrier
    do_many_other_things();
}
```
OpenMP Single

- Denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

```c
#pragma omp parallel private (tmp)
{
    do_many_things();
#pragma omp single
    {
        exchange_boundaries();
    }
    do_many_other_things();
}
```
Combined Parallel/Workshare

- OpenMP shortcut: Put the “parallel” and the work-share on the same line

```c
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i< MAX; i++) { 
        res[i] = huge(); 
    }
}
```

These are equivalent

```c
double res[MAX]; int i;
#pragma omp parallel for
for (i=0; i< MAX; i++) {
    res[i] = huge();
}
```

- There’s also a “parallel sections” construct.
The OpenMP specification does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be *orphaned*.

They can appear outside the lexical extent of a parallel region.
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
OpenMP Synchronization

- Synchronization enables the user to
  - Control the **ordering of executions** in different threads
  - Ensure that at most one thread executes operation or region of code at any given time (**mutual exclusion**)

- High level synchronization:
  - barrier
  - critical section
  - atomic
  - ordered

- Low level synchronization:
  - flush
  - locks (both simple and nested)
When these loops are parallelized, we need to be sure to update all of \( a[] \) before using \( a[] \) *

All threads wait at the barrier point and only continue when all threads have reached the barrier point

*) If the mapping of iterations onto threads is guaranteed to be identical for both loops, we do not need to wait. This is the case with the static schedule under certain conditions
Barrier: Explicit and Implicit

- Each thread waits until all threads arrive.

```c
#pragma omp parallel shared (A, B, C) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    #pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
    #pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C, i); }
    A[id] = big_calc3(id);
}
```

- Implicit barrier at the end of a parallel region
- Implicit barrier at the end of a `for` work-sharing construct
- No implicit barrier due to `nowait`
The Nowait Clause

- Barriers are implied at end of parallel region, for/do, sections and single constructs
- Barrier can be suppressed by using the optional nowait clause
  - If present, threads do not synchronize/wait at the end of that particular construct

```c
#pragma omp for nowait
{
    :
}

!$omp do
    :
    :
!$omp end do nowait
```
Mutual Exclusion

- Code may only be executed by at most one thread at any given time
- Could lead to long wait times for other threads
  - Atomic updates for individual operations
  - Critical regions and locks for structured regions of code
Only one thread at a time can enter a critical region.

```c
float res;
#pragma omp parallel
{
    float B; int i;
    #pragma omp for
    for(i=0;i<niters;i++){
        B = big_job(i);
        #pragma omp critical
        consume (B, RES);
    }
}
```

Threads wait their turn – only one at a time calls `consume()`.

Use e.g. when all threads update a variable and the order in which they do so is unimportant. Preserves data integrity.
Atomic

- Atomic is a special case of mutual exclusion
- It applies only to the update of a memory location

```c
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
    tmp = big_ugly();
C$OMP ATOMIC
    X = X + temp
C$OMP END PARALLEL
```

The statement inside the atomic must be one of:
- `x binop= expr`
- `x = x binop expr`
- `x = expr binop x`
- `x++`
- `++x`
- `x--`
- `--x`

X is an lvalue of scalar type and binop is a non-overloaded built in operator.

OpenMP 3.1 describes the behavior in more detail via these clauses:
- read, write, update, capture

The pre-3.1 atomic construct is equivalent to
- `#pragma omp atomic update`
Ordered

- The **ordered** construct enforces the sequential order for a block.
- Code is executed in the order in which iterations would be performed sequentially

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered
for (i=0;i<N;i++){
    tmp = NEAT_STUFF(i);
#pragma ordered
    res += consum(tmp);
}
```
Updates to Shared Data

- Blocks of data are fetched into cache lines
- Values may temporarily differ from other copies of data within a parallel region
Flushing and The Flush Directive

- Flushing is what creates a consistent view of shared data: it causes a thread to write data back to main memory and retrieve new values of updated variables.
- It is automatically performed on a number of constructs.
  - Normally you don’t need to think about this.
- The **flush construct** allows the programmer to explicitly define a point where a thread makes its variable values consistent with main memory.
  - Caution: it does not enable a thread to retrieve values updated by another thread unless that thread also performs a flush.
  - It also does not synchronize threads.
  - Its use is tricky: be sure you understand it.
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
End of Part 1 ... 
And now, a few examples
Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

\[ \sum_{i=0}^{N} F(x_i) \Delta x \approx \pi \]

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
Pi Program: Sequential Version

```c
#define NUMSTEPS 100000000
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;

    step = 1.0/(double) NUMSTEPS;

    for (i=1;i<= NUMSTEPS; i++) {
        x = (i-0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```
Example: OpenMP Pi Program

```c
#include <omp.h>
static long num_steps = 100000000;
double step;
#define NUM_THREADS 8
void main ()
{
    int I, nthreads; double x, pi, sum[NUM_THREADS] = {0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    {
        double x; int id, i, nthrds;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        for (i=id; i< num_steps; i=i+nthrds) {
            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;
}
```

SPMD: Each thread runs the same code. The thread ID enables thread-specific behavior.

Promote scalar to array so each thread computes local sum

Only one thread copies value to global variable

Creates cyclic distribution of iterations to threads
Example: Parallel Pi

This version:

- Uses the parallel construct.
  - Paying close attention to shared versus private variables.
- In addition to a parallel construct, it uses these runtime library routines:
  - `intomp_get_num_threads();` Get / set number of threads in team
  - `voidomp_set_num_threads();`
  - `intomp_get_thread_num();` Get thread ID (rank)
#include <mpi.h>
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs);
    my_steps = num_steps/numprocs;
    for (i=my_id*my_steps; i<(my_id+1)*my_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
}
OpenMP and MPI

Calculating Pi: Comparing OpenMP (SPMD style) and MPI on dual-socket Intel Xeon E5-2665

Next Improvements:
- more flexible worksharing construct?
- Optimize use of cache
```c
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main () {
    int i; double x, pi, sum[NUM_THREADS] ={0.0};
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel
    { double x; int i, id;
        id = omp_get_thread_num();
        #pragma omp for
        for (i=0;i< num_steps; i++){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
    // Additional code here...
}
```
OpenMP PI Program with Reduction

```c
#include <omp.h>
static long num_steps = 100000; double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++)
    { 
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

OpenMP adds 1-2 lines of code
#include <stdlib.h>
#include <sys/time.h>

void * compute_pi(void *dat)
{
    int threadid = ((thr_data_t*)dat)->threadid;
    int num_threads = ((thr_data_t*)dat)->num_threads;
    int num_steps = ((thr_data_t*)dat)->num_steps;
    pthread_mutex_t *mtx = ((thr_data_t*)dat)->mtx;
    double *sump = ((thr_data_t*)dat)->sump;
    int i;
    double step;
    double x, local_sum;

    step = 1.0 / num_steps;
    local_sum = 0.0;
    /* round robin distribution of iterations */
    for (i = threadid; i < num_steps; i += num_threads) {
        x = (i - 0.5)*step;
        local_sum += 4.0 / (1.0 + x*x);
    }

    pthread_mutex_lock(mtx);
    *sump = *sump + local_sum;
    pthread_mutex_unlock(mtx);
    return NULL;
}

int main(int argc, char **argv)
{
    /* start pi calculation */
    threads = malloc(num_threads * sizeof *threads);
    step = 1.0 / num_steps;
    pthread_mutex_init(&mtx, NULL);

    /* spawn threads to work on computing pi */
    for (i = 0; i < num_threads; i++) {
        dat[i].threadid = i;
        dat[i].num_threads = num_threads;
        dat[i].num_steps = num_steps;
        dat[i].mtx = &mtx;
        dat[i].sump = &sum;
        pthread_create(&threads[i], NULL, compute_pi,
                       (void *)&dat[i]);
    }

    /* join threads */
    for (i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    pi = step * sum;
    free(dat);
    pthread_mutex_destroy(&mtx);
    free(threads);
    ...}
OpenMP and MPI

Calculating Pi: Comparing Pthreads, OpenMP, and MPI on dual-socket Intel Xeon E5-2665

- pthreads
- omp v1
- omp v2 (omp for)
- omp v3 (omp for reduction)
- mpi
Example: Seismic Data Processing (SDP)

```c
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
{
    Loadline(iLineIndex,...);
    for(j=0;j<iNumTraces;j++)
        for(k=0;k<iNumSamples;k++)
            processing();
    SaveLine(iLineIndex);
}
```

Load
Data

Process
Data

Save
Data

Timeline
First OpenMP Version of SDP Code

```c
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
{
    Loadline(iLineIndex,...);
    #pragma omp parallel for
    for(j=0;j<iNumTraces;j++)
        for(k=0;k<iNumSamples;k++)
            processing();
    SaveLine(iLineIndex);
}
```

Overhead for entering and leaving the parallel region

Better performance, but not too encouraging

Timeline
Example: Overlap I/O, Processing

```c
#pragma omp parallel
#pragma omp sections
{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
} /*-- End of parallel sections --*/
```
## Overlap I/O And Processing

<table>
<thead>
<tr>
<th>Time</th>
<th>Input Thread</th>
<th>Processing Thread(s)</th>
<th>Output Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Processing Thread(s)*
Another OpenMP Version of SDP Code

```
Loadline(nStartLine,...);  // preload the first line of data
#pragma omp parallel
{
    for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
    {
        #pragma omp single nowait
        { // loading the next line data, NO WAIT!
            Loadline(iLineIndex+1,...);
        }
        #pragma omp for schedule(dynamic)
        for(j=0;j<iNumTraces;j++)
            for(k=0;k<iNumSamples;k++)
                processing();
        #pragma omp single nowait
        {
            SaveLine(iLineIndex);
        }
    }
}
```
Part 2
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
OpenMP Memory Model

- OpenMP assumes a shared memory
- Threads communicate by sharing variables.
- Synchronization protects data conflicts.
  - Synchronization is expensive.
  - Change how data is accessed to minimize the need for synchronization.

- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit
OpenMP Data Environment

- Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared by default...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables defined inside the parallel region are PRIVATE.
- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

All data clauses apply to parallel regions, tasks and work-sharing constructs except “shared” which does not apply to work-sharing constructs.
double a[size][size], b=4;
#pragma omp parallel private (b)
{
    ....
}

Private variable b becomes undefined on exit from region.
program sort
  common /input/ A(10)
  integer index(10)
C$OMP PARALLEL
  call work (index)
C$OMP END PARALLEL
print*, index(1)

A, index and count are shared by all threads.
temp is local to each thread

subroutine work (index)
  common /input/ A(10)
  integer index(*)
  real temp(10)
  integer count
  save count
  ...........

A, index, count

A, index, count

temp

temp

temp
OpenMP Private Clause

- **private**(var) creates a local copy of var for each thread.
- The value is *uninitialized*
- Private copy is *not storage-associated* with original
- Parallel region does not modify original variable

```c
IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
  DO J=1,1000
    IS = IS + J
  END DO
C$OMP END PARALLEL DO
print *, IS
```

- **IS** was not initialized
- IS here is not storage-associated with the private variable with the same name
#pragma omp parallel private(x) shared(p0, p1)

Thread 0

x = ...;
p0 = &x;

/* references in the following line are **not** allowed */
... *p1 ...

Thread 1

x = ...;
p1 = &x;

... *p0 ...

- You can not reference another thread’s private variables ... even if you have a shared pointer between the two threads.
Firstprivate Clause

- **firstprivate** is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```fortran
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
  20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```

Each thread gets its own IS with an initial value of 0

The value of IS here is not influenced by the computation in the parallel region
Lastprivate Clause

- **Lastprivate** passes the value of a private variable from the **last** iteration to the variable of the master thread.

```c
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP& LASTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
  20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```

IS is defined as its value at the last iteration (i.e. for J=1000)

Each thread gets its own IS with an initial value of 0
A Data Environment Checkup

- Consider this example of PRIVATE and FIRSTPRIVATE

```c
variables A, B, and C = 1
C$OMP PARALLEL PRIVATE(B)
C$OMP& FIRSTPRIVATE(C)
```

- Are A, B, C local to each thread or shared inside the parallel region?
- What are their initial values inside and after the parallel region?

Inside this parallel region ...
- “A” is shared by all threads; equals 1
- “B” and “C” are local to each thread.
  - B’s initial value is undefined
  - C’s initial value equals 1

Outside this parallel region ...
- A has value from parallel region. The values of “B” and “C” are not influenced by code inside region.
OpenMP Reduction

- If it’s the sum of all J values that you need, there is a way to do that too.
- We have already seen how

```
IS = 0
C$OMP PARALLEL DO REDUCTION(+:IS)
    DO 1000 J=1,1000
        IS = IS + J
    1000 CONTINUE
print *, IS
```

Result variable is shared by default
An accumulation operation across threads

Inside a parallel or work-sharing construct:

- A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
- Compiler finds standard reduction expressions containing "op" and uses them to update the local copy.
- Local copies are reduced into a single value and combined with the original global value.

The variables in "list" must be shared in the enclosing parallel region.
**Reduction Operands/Initial Values**

- Associative operands used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Language</th>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++/Fortran</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>C/C++/Fortran</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>C/C++/Fortran</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>C/C++/Fortran</td>
<td>max</td>
<td>least representable</td>
</tr>
<tr>
<td>C/C++/Fortran</td>
<td>min</td>
<td>largest representable</td>
</tr>
<tr>
<td>C/C++</td>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>&amp;</td>
<td>bits all 1’s</td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>^</td>
<td>bits all 0’s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language</th>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>.and.</td>
<td>.true.</td>
</tr>
<tr>
<td>Fortran</td>
<td>.or.</td>
<td>.false.</td>
</tr>
<tr>
<td>Fortran</td>
<td>.equiv.</td>
<td>.true.</td>
</tr>
<tr>
<td>Fortran</td>
<td>.neqv.</td>
<td>.false.</td>
</tr>
<tr>
<td>Fortran</td>
<td>iand</td>
<td>bits all 1’s</td>
</tr>
<tr>
<td>Fortran</td>
<td>ior</td>
<td>bits all 0’s</td>
</tr>
<tr>
<td>Fortran</td>
<td>iexor</td>
<td>bits all 0’s</td>
</tr>
</tbody>
</table>
Default Clause Example

itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
    np = omp_get_num_threads()
    each = itotal/np
 .......... 
C$OMP END PARALLEL

Are these two codes equivalent?

yes
Example - Parallelizing Bulky Loops

```c
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...  
    b = ... a ..
    c[i] = ....
        ....
    for (j=0; j<m; j++)
    {
        <a lot more data and work in this loop>
    }
        ....
}
```
#pragma omp parallel for private(..) shared(..)

```c
for (int i=0; i<n; i++) /* Parallel loop */
{
    (void) FuncPar(i,m,c,...)
} /*-- End of parallel for --*/
```

- **Minimal scoping required**
- **Less error prone**

```c
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
    ....
    for (j=0; j<m; j++)
    {
        <a lot more work in this loop>
    }
    ....
}
```
OpenMP Threadprivate

- Makes global data private to a thread and persistent, **thus crossing parallel region boundary**
  - Fortran: COMMON blocks, variables with SAVE attribute
  - C: File scope and static variables
- Different from making them PRIVATE
  - With PRIVATE, global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.
- Some limitations on use of threadprivate
  - Consult specification before using this feature
A Threadprivate Example

Consider two different routines called within a parallel region.

subroutine poo
parameter (N=1000)
common,buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
do i=1, N
   B(i)= const* A(i)
end do
return
end

subroutine bar
parameter (N=1000)
common,buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
do i=1, N
   A(i) = sqrt(B(i))
end do
return
end

Because of the threadprivate construct, each thread executing these routines has its own copy of the common block /buf/.

Values of threadprivate are persistent across parallel regions.
The Copyin Clause

**copyin (list)**

- Applies to THREADPRIVATE data only
- At the start of the parallel region, data of the master thread is copied to the thread private copies

**Example:**

```plaintext
common /cblock/velocity
common /fields/xfield, yfield, zfield

! create thread private common blocks
!$omp threadprivate (/cblock/, /fields/)

!$omp parallel &
!$omp default (private) &
!$omp copyin ( /cblock/, zfield )
```

Data now available to threads
Copyprivate

- Used with a single region to broadcast values of private variables from one member of a team to the rest of the team.

```c
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
    int Nsize, choice;

    #pragma omp parallel private (Nsize, choice)
    {
        ...  
        #pragma omp single copyprivate (Nsize, choice)
        input_parameters (Nsize, choice);

        do_work(Nsize, choice);
    }
}
```
C++ And Threadprivate

- OpenMP 3.0 clarified where/how threadprivate objects are constructed and destructed
- Allows C++ static class members to be threadprivate

```cpp
class T {
    public:
        static int i;
        #pragma omp threadprivate(i)
        ...
    }
};
```
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
OpenMP Runtime Functions

- OpenMP provides a set of runtime functions
  - They all start with “omp_”
- These functions can be used to:
  - Query for a specific feature, value or setting
    - E.g. what is my thread ID?
  - Change a setting
    - E.g. to change the number of threads in next parallel region
- A special category consists of the locking functions

C/C++ : Need to include file `<omp.h>`
Fortran : Add “use omp_lib” or include file “omp_lib.h”
OpenMP Library Routines

- Modify/Check the number of threads
  - `omp_set_num_threads()`, `omp_get_num_threads()`, `omp_get_thread_num()`, `omp_get_max_threads()`

- Are we in a parallel region?
  - `omp_in_parallel()`

- How many processors in the system?
  - `omp_num_procs()`
OpenMP Library Routines

To use a known, fixed number of threads in a program, (1) tell the system that you don’t want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```c
#include <omp.h>
void main()
{
    int num_threads;
    omp_set_dynamic( 0 );
    omp_set_num_threads( omp_num_procs() );
    #pragma omp parallel
    {
        int id=omp_get_thread_num();
        #pragma omp single
        num_threads = omp_get_num_threads();
        do_lots_of_stuff(id);
    }
}
```

- Disable dynamic adjustment of the number of threads.
- E.g. Request as many threads as you have processors.
- Protect this op since memory stores are not atomic
- Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.
Nested Parallelism

- Allows parallel regions to be contained in each other
  - Often accomplished by having parallel regions in different functions
- Required: OMP_NESTED=true or omp_set_nested(1)
  - Else the inner parallel region will be executed by a team of one thread (may happen anyway)
- Total number of threads created is the *product* of the number of threads in the teams at each level
  - Use omp_set_num_thread(n) or the num_threads() clause

Multiple levels of nesting team sizes can be defined via the
- OMP_NUM_THREADS environment variable
- setenv OMP_NUM_THREADS 4,2
Nested Parallelism

Note: Nesting level can be arbitrarily deep
Locking Routines

- Simple locks: may not be locked if already in a locked state
- Nestable locks: may be locked multiple times by the same thread before being unlocked
- The interface for functions dealing with nested locks is similar (but using nestable lock variables):

<table>
<thead>
<tr>
<th>Simple locks</th>
<th>Nestable locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_init_lock</td>
<td>omp_init_nest_lock</td>
</tr>
<tr>
<td>omp_destroy_lock</td>
<td>omp_destroy_nest_lock</td>
</tr>
<tr>
<td>omp_set_lock</td>
<td>omp_set_nest_lock</td>
</tr>
<tr>
<td>omp_unset_lock</td>
<td>omp_unset_nest_lock</td>
</tr>
<tr>
<td>omp_test_lock</td>
<td>omp_test_nest_lock</td>
</tr>
</tbody>
</table>
## OpenMP Runtime Functions (Threading)

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_max_threads</code></td>
<td>Max num of threads for parallel region</td>
</tr>
<tr>
<td><code>omp_get_num_threads</code></td>
<td>Number of threads in current team</td>
</tr>
<tr>
<td><code>omp_get_thread_num</code></td>
<td>Get thread ID in current team</td>
</tr>
<tr>
<td><code>omp_set_num_threads</code></td>
<td>Set num threads for next parallel region</td>
</tr>
<tr>
<td><code>omp_get_num_procs</code></td>
<td>Maximum number of processors on device</td>
</tr>
<tr>
<td><code>omp_set_dynamic</code></td>
<td>Activate dynamic thread adjustment</td>
</tr>
<tr>
<td></td>
<td><em>(but implementation is free to ignore this)</em></td>
</tr>
<tr>
<td><code>omp_get_dynamic</code></td>
<td>Check for dynamic thread adjustment</td>
</tr>
<tr>
<td><code>omp_get_thread_limit</code></td>
<td>Max num of threads available</td>
</tr>
<tr>
<td><code>*omp_get_proc_bind</code></td>
<td>Thread affinity policy for next parallel region</td>
</tr>
</tbody>
</table>

* OpenMP 4.0 addition
<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_in_parallel</td>
<td>Check whether in parallel region</td>
</tr>
<tr>
<td>omp_set_nested</td>
<td>Activate nested parallelism</td>
</tr>
<tr>
<td></td>
<td>(but implementation is free to ignore this)</td>
</tr>
<tr>
<td>omp_get_nested</td>
<td>Check for nested parallelism</td>
</tr>
<tr>
<td>omp_set_max_active_levels</td>
<td>Set limit on active nested regions</td>
</tr>
<tr>
<td>omp_get_max_active_levels</td>
<td>Check the limit for active nested regions</td>
</tr>
<tr>
<td>omp_get_ancestor_thread_num</td>
<td>Ancestor thread num at given nested level</td>
</tr>
<tr>
<td>omp_get_level</td>
<td>Check num of enclosing regions</td>
</tr>
<tr>
<td>omp_get_active_level</td>
<td>Check num of enclosing active regions</td>
</tr>
<tr>
<td>omp_get_team_size</td>
<td>Thread team size at given nested level</td>
</tr>
</tbody>
</table>
## OpenMP Runtime Functions (Loop Schedule)

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_set_schedule</td>
<td>Set runtime loop schedule</td>
</tr>
<tr>
<td>omp_get_schedule</td>
<td>Check the runtime loop schedule</td>
</tr>
</tbody>
</table>
## OpenMP Runtime Functions (Timing)

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_get_wtime</code></td>
<td>Returns wall clock time</td>
</tr>
<tr>
<td><code>omp_get_wtick</code></td>
<td>Number of seconds between clock ticks</td>
</tr>
</tbody>
</table>
### OpenMP Runtime Functions (Tasking and Error-handling)

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_in_final</code></td>
<td>Check if in a final task region</td>
</tr>
<tr>
<td><code>omp_get_cancellation</code></td>
<td>Check if cancellation is activated</td>
</tr>
</tbody>
</table>

* OpenMP 4.0 addition
<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_init_lock</code></td>
<td>Initialize a simple lock</td>
</tr>
<tr>
<td><code>omp_init_nest_lock</code></td>
<td>Initialize a nestable lock</td>
</tr>
<tr>
<td><code>omp_destroy_lock</code></td>
<td>Destroy a simple lock</td>
</tr>
<tr>
<td><code>omp_destroy_nest_lock</code></td>
<td>Destroy a nestable lock</td>
</tr>
<tr>
<td><code>omp_set_lock</code></td>
<td>Wait for and then set a simple lock</td>
</tr>
<tr>
<td><code>omp_set_nest_lock</code></td>
<td>Wait for and then set a nestable lock</td>
</tr>
<tr>
<td><code>omp_unset_lock</code></td>
<td>Unsets a simple lock</td>
</tr>
<tr>
<td><code>omp_unset_nest_lock</code></td>
<td>Unsets a nestable lock</td>
</tr>
<tr>
<td><code>omp_test_lock</code></td>
<td>Test, and if available set, a simple lock</td>
</tr>
<tr>
<td><code>omp_test_nest_lock</code></td>
<td>Test, and if available set, a nestable lock</td>
</tr>
</tbody>
</table>
## OpenMP Runtime Functions (Devices)

<table>
<thead>
<tr>
<th>Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_is_initial_device</code></td>
<td>Check if executing on host device</td>
</tr>
<tr>
<td><code>omp_get_num_devices</code></td>
<td>Check number of target devices</td>
</tr>
<tr>
<td><code>omp_set_default_device</code></td>
<td>Set default target device number</td>
</tr>
<tr>
<td><code>omp_get_default_device</code></td>
<td>Check default target device number</td>
</tr>
<tr>
<td><code>omp_get_num_teams</code></td>
<td>Check if in a final task region</td>
</tr>
<tr>
<td><code>omp_get_team_num</code></td>
<td>Check if task cancellation is activated</td>
</tr>
</tbody>
</table>

* OpenMP 4.0 addition
Be careful when relying on defaults (because they are compiler dependent)

<table>
<thead>
<tr>
<th>OpenMP Environment Variable</th>
<th>IBM XL Compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>64 (BG/Q) number of available processors (other systems)</td>
</tr>
<tr>
<td>OMP_SCHEDULE &quot;schedule,[chunk]&quot;</td>
<td>auto</td>
</tr>
<tr>
<td>OMP_DYNAMIC {TRUE</td>
<td>FALSE}</td>
</tr>
<tr>
<td>OMP_NESTED {TRUE</td>
<td>FALSE}</td>
</tr>
<tr>
<td>OMP_STACKIZE &quot;size [B</td>
<td>K</td>
</tr>
<tr>
<td>OMP_WAIT_POLICY [ACTIVE</td>
<td>PASSIVE]</td>
</tr>
<tr>
<td>OMP_MAX_ACTIVE_LEVELS</td>
<td>5</td>
</tr>
</tbody>
</table>
## OpenMP Environment Variables/2

<table>
<thead>
<tr>
<th>OpenMP Environment Variable</th>
<th>Default</th>
<th>Oracle Solaris Studio</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_THREAD_LIMIT</td>
<td>max(OMP_NUM_THREADS, number of available processors) (other systems)</td>
<td>64 (BG/Q)</td>
</tr>
<tr>
<td>OMP_PROC_BIND {TRUE</td>
<td>FALSE}</td>
<td>TRUE (BG/Q, FALSE is ignored)</td>
</tr>
</tbody>
</table>
Implementing The Fork-Join Model

Use the `OMP_WAIT_POLICY` environment variable to control the behavior of idle threads. Values: ACTIVE, PASSIVE.
About The Stack

```c
void myfunc(float *Aglobal)
{
    int Alocal;
    .......
}

#pragma omp parallel shared(Aglobal)
{
    (void) myfunc(&Aglobal);
}
```

Variable Alocal is in private memory, managed by the thread owning it, and stored on the so-called stack

Set stacksize via OMP_STACKSIZE environment variable
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
- Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
Tasks In OpenMP

- Tasking was introduced in OpenMP 3.0
- Until then it was impossible to efficiently implement certain types of parallelism
  - Recursive algorithms
  - Linked lists, ...
- The initial functionality was very simple by design
  - The idea was (and still) is to augment tasking as we collectively gain more insight and experience
The Tasking Concept In OpenMP

- Thread
- Generate tasks
- Execute tasks
The Task Construct

**Define a task:**

<table>
<thead>
<tr>
<th>#pragma omp task</th>
</tr>
</thead>
<tbody>
<tr>
<td>!$omp task</td>
</tr>
</tbody>
</table>

- **A task** is a specific instance of executable code and its data environment.
- A task is generated when a thread encounters a task construct or a parallel construct. Comprised of a task region and data environment.
- A task region consists of all code encountered during the execution of a task.
- The data environment consists of all the variables associated with the execution of a given task. It is constructed from the data environment of the generating task at the time the task is generated.
Task Completion

Explicit wait on the completion of child tasks:

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

Does not include descendents of child tasks
Data Scoping in Tasks

- Static and global variables are shared
- Automatic storage (local) variables are private
- Variables are firstprivate unless shared in the enclosing context

```c
int a;
void foo()
{
    int b, c;

    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}
```
### Clauses On The Task Directive

<table>
<thead>
<tr>
<th>-Clause Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if(scalar-expression)</code></td>
<td>if false, create an undeferred task: encountering thread must suspend the encountering task region until the current task region has completed. Any thread can resume after suspension if true, the generated task is a final task, meaning all descendent tasks are executed immediately.</td>
</tr>
<tr>
<td><code>untied</code></td>
<td>if the task is an undeferred task or an included task, the implementation may merge the task into the parent (a merged task)</td>
</tr>
<tr>
<td>`default(shared</td>
<td>none)`</td>
</tr>
<tr>
<td><code>private(list)</code></td>
<td></td>
</tr>
<tr>
<td><code>firstprivate(list)</code></td>
<td></td>
</tr>
<tr>
<td><code>shared(list)</code></td>
<td></td>
</tr>
<tr>
<td><code>final(scalar-expression)</code></td>
<td></td>
</tr>
<tr>
<td><code>mergeable</code></td>
<td></td>
</tr>
</tbody>
</table>
Example: A Linked List

```
.........
while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
.........
```

*Hard to do before OpenMP 3.0:*
First count number of iterations, then convert while loop to for loop
Example: A Linked List with Tasking

```c
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```
The taskyield directive specifies that the current task can be suspended in favor or execution of a different task.

Hint to the runtime

The waiting task may be suspended here so that the executing thread can perform other work.
Final Clause

For recursive problems that perform task decomposition
- stop task creation at a certain depth
- exposes enough parallelism while reducing overhead.

Warning: Merging the data environment may have side-effects

```c
void foo(bool arg)
{
    int i = 3;
    #pragma omp task final(arg) firstprivate(i)
        i++;
    printf("%d\n", i); // will print 3 or 4 depending on arg
}
```
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            {
                printf("race ");
            }
            #pragma omp task
            {
                printf("car ");
            }
            printf("is fun to watch ");
        }
        // End of parallel region
        printf("\n");
        return(0);
    }
}

What will this program print using 2 threads?
Task Example

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$
int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A ");
            #pragma omp task
            { printf("car ");}
            #pragma omp task
            { printf("race ");}
            #pragma omp taskwait
            printf("is fun to watch ");
        }
    } // End of parallel region
    printf("\n"); return(0);
}  

What will this program print using 2 threads?
Task Example

$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$

Tasks are executed first now
OpenMP Locking Example

- The protected region contains the update of a shared variable
- One thread acquires the lock and performs the update
- Meanwhile, the other thread performs some other work
- When the lock is released again, the other thread performs the update
Locking Example - The Code

Program Locks
....
Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
  Call Do_Something_Else()
End Do

Call Do_Work()

Call omp_unset_lock (LCK)

!$omp end parallel

Call omp_destroy_lock (LCK)

Stop
End
### Example Output Using 2 Threads

<table>
<thead>
<tr>
<th>TID: 1 at 09:07:27</th>
<th>entered parallel region</th>
</tr>
</thead>
<tbody>
<tr>
<td>TID: 1 at 09:07:27</td>
<td>done with WAIT loop and has the lock</td>
</tr>
<tr>
<td>TID: 1 at 09:07:27</td>
<td><strong>ready to do the parallel work</strong></td>
</tr>
<tr>
<td>TID: 1 at 09:07:27</td>
<td>this will take about 18 seconds</td>
</tr>
<tr>
<td>TID: 0 at 09:07:27</td>
<td>entered parallel region</td>
</tr>
<tr>
<td>TID: 0 at 09:07:27</td>
<td>WAIT for lock - will do something else for 5 seconds</td>
</tr>
<tr>
<td>TID: 0 at 09:07:32</td>
<td>WAIT for lock - will do something else for 5 seconds</td>
</tr>
<tr>
<td>TID: 0 at 09:07:37</td>
<td>WAIT for lock - will do something else for 5 seconds</td>
</tr>
<tr>
<td>TID: 0 at 09:07:42</td>
<td>WAIT for lock - will do something else for 5 seconds</td>
</tr>
<tr>
<td>TID: 1 at 09:07:45</td>
<td>done with my work</td>
</tr>
<tr>
<td>TID: 1 at 09:07:45</td>
<td>done with work loop - released the lock</td>
</tr>
<tr>
<td>TID: 1 at 09:07:45</td>
<td>ready to leave the parallel region</td>
</tr>
<tr>
<td>TID: 0 at 09:07:47</td>
<td>done with WAIT loop and has the lock</td>
</tr>
<tr>
<td>TID: 0 at 09:07:47</td>
<td><strong>ready to do the parallel work</strong></td>
</tr>
<tr>
<td>TID: 0 at 09:07:47</td>
<td>this will take about 18 seconds</td>
</tr>
<tr>
<td>TID: 0 at 09:08:05</td>
<td>done with my work</td>
</tr>
<tr>
<td>TID: 0 at 09:08:05</td>
<td>done with work loop - released the lock</td>
</tr>
<tr>
<td>TID: 0 at 09:08:05</td>
<td>ready to leave the parallel region</td>
</tr>
</tbody>
</table>

Done at 09:08:05 - value of SUM is 1100

---

**Note:** Program was instrumented to get this information
All 41 examples are available online!
As well as a forum on http://www.openmp.org

Download the examples and discuss in forum:
Agenda

- What is OpenMP?
- The elements of OpenMP
  - Parallel regions
  - Worksharing constructs
  - Synchronization
  - Managing the data environment
  - The runtime library and environment variables
  - Tasks
- OpenMP 4.0
  - A brief overview

We will also show some examples as we go along.
OpenMP 4.0

- Released July 2013

- Changes from 3.1 to 4.0 (Appendix E.1):
  - Accelerator: 2.9
  - SIMD extensions: 2.8
  - Places and thread affinity: 2.5.2, 4.5
  - Taskgroup and dependent tasks: 2.12.5, 2.11
  - Error handling: 2.13
  - User-defined reductions: 2.15
  - Sequentially consistent atomics: 2.12.6
  - Fortran 2003 support
Architectural Changes

- 256 Cores
  - 4-way SIMD FMACs @ 2.5–5 GHz
  - 5–10 TFlops on one chip
  - Some apps require 1 byte/flop
  - 5–10 TB/s of off-chip/on-chip BW

Source: Chuck Moore, *Data Processing in ExaScale-ClassComputer Systems*, Salishan, April 2011
Computation and Data Offloading for Accelerators

- **#pragma omp target** `device(id) map() if()`
  - **target**: create a data environment and offload computation to the device
  - **device (int_exp)**: specify a target device
  - **map(to|from|tofrom|alloc:var_list)**: data mapping between the current data environment and a device data environment

- **#pragma target data** `device (id) map() if()`
  - Create a device data environment: to be reused/inherited
Accelerator: Explicit Data Mapping

- Relatively small number of truly shared memory accelerators so far
- Requires the user to explicitly map data to and from the device memory
- Can use array sections

```c
long a = 0x858;
long b = 0;
int anArray[100]

#pragma omp target data map(to:a) \n    map(tofrom:b,anArray[0:64])
{
    /* a, b and anArray are mapped * to the device */

    /* work on the device */
    #pragma omp target …
    {
        ...
    }

    /* b and anArray are mapped * back to the host */
```
Target and Map Examples

```c
void vec_mult(int N)
{
    int i;
    float p[N], v1[N], v2[N];
    init(v1, v2, N);
    #pragma omp target map(to: v1, v2) map(from: p)
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```

```c
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
    #pragma omp parallel for
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(from: p[0:N])
    {
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        init_again(v1, v2, N);
        #pragma omp target map(to: v1[:N], v2[:N])
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
Accelerator: Hierarchical Parallelism

- Organize massive number of threads
  - teams of threads, e.g. map to CUDA grid/block
- Distribute loops over teams

```c
#pragma omp target
#pragma omp teams num_teams(2) num_threads(8)
{
    //-- creates a “league” of teams
    //-- only local barriers permitted
#pragma omp distribute
    for (int i=0; i<N; i++) { ... }
}
```
Teams and Distribute Loop Example

```c
float dotprod_teams(float B[], float C[], int N, int num_blocks,
                   int block_threads)
{
    float sum = 0;
    int i, i0;

    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_blocks) thread_limit(block_threads)
    reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += num_blocks)
        #pragma omp parallel for reduction(+:sum)
        for (i=i0; i< min(i0+num_blocks,N); i++)
            sum += B[i] * C[i];

    return sum;
}
```

Double-nested loops are mapped to the two levels of thread hierarchy (league and team)
SIMD Loops

- **omp for simd**: loop chunked among team of threads and then each chunk is simdized

```c
void vec_mult(float *p, float *v1, float *v2, int N) {
    int i;
    init(v1, v2, N);
    #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
    #pragma omp distribute parallel for simd
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```
## Places and Thread Affinity

- **OMP_PLACES** to describe a list of places (4.5) and the *hardware threads* of each place
  - OMP_PLACES = 
  - OMP_PLACES = threads | cores | sockets
    - threads: place $\rightarrow$ hardware thread
    - cores: place $\rightarrow$ core (may have multiple threads)
    - sockets: place $\rightarrow$ socket (may have multiple cores)

```bash
setenv OMP_PLACES threads
setenv OMP_PLACES "threads(4)"
setenv OMP_PLACES "\{0,1,2,3\},\{4,5,6,7\},\{8,9,10,11\},\{12,13,14,15\}"
setenv OMP_PLACES "\{0:4\},\{4:4\},\{8:4\},\{12:4\}"
setenv OMP_PLACES "\{0:4\}:4:4"
```
Places and Thread Affinity

- **proc_bind(master|close|spread) clause of parallel** to specify policy of assigning **OpenMP threads** to places
  - **master**: All threads of the team go to the place of the master thread
  - **close**: assign threads to place close to the place of parent thread
    ```c
    void work2()
    {
        #pragma omp parallel num_threads(12) proc_bind(close)
        {
            /* do work here */
        }
    }
    OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"
    ```
    - If master thread is in place {2,3}, then:
      - threads 0-2 execute on the place {2,3}
      - threads 3-5 execute on the place {4,5}
      - threads 6-8 execute on the place {6,7}
      - threads 9-11 execute on the place {0,1}
  - **Place set for each implicit task is still "{0,1},{2,3},{4,5},{6,7}"**
Places and Thread Affinity

- **proc_bind(master|close|spread)** clause of **parallel** to specify policy of assigning **OpenMP threads** to places
  - **spread**: subpartition parent place sets, and then assign threads to place close to the new places
  - each implicit task’s place set may change (nested parallel region)

```c
void work3()
{
    #pragma omp parallel num_threads(4) proc_bind(spread)
    {
        /* do work here */
    }
}
```

- If master thread is in place \{2,3\}:
  - threads 0-2 execute on the place \{2,3\}
  - threads 3-5 execute on the place \{4,5\}
  - threads 6-8 execute on the place \{6,7\}
  - threads 9-11 execute on the place \{0,1\}

```c
OMP_PLACES = "{0,1},{2,3},{4,5},{6,7}"
```
OpenMP 4.0 Tasking Extensions

- Specifying dependencies of tasks sharing the same parent:
  
  ```c
  #pragma omp task depend (out:t_1, t_2,..., t_n) \n  depend (in:t_1,t_2, ..., t_n) depend (inout:t_1 ,t_2, ..., t_n)
  ```

- `taskgroup`:
  - Join all tasks created within the group (like X10 finish)
  - `taskwait vs taskgroup`
    - Taskwait only joins the child tasks
    - Taskgroup joins the child and descendendent tasks
Cancellation (Error Handling)

- **cancel directive**
  - `#pragma omp cancel [clause[, ]clause] ...`
  - `!$omp cancel [clause[, ]clause] ...`
  - Clauses: parallel, sections, for/do, taskgroup
User-Defined Reductions

```plaintext
#pragma omp declare reduction( reduction-identifier : typename-list : combiner ) [initializer-clause] new-line
```

where:

- `reduction-identifier` is either a base language identifier or one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&` and `||`
- `typename-list` is list of type names
- `combiner` is an expression
- `initializer-clause` is `initializer( initializer-expr )` where `initializer-expr` is `omp_priv = initializer` or `function-name( argument-list )`
Atomic Extensions

```c
#pragma omp atomic [read|write|update|capture][seq_cast]
```

- capture clause: to allow new atomic update and capture the original or result value, e.g. fetch-and-add operation
- seq_cst clause: to support sequentially consistent atomic operations, i.e. force a flush
OpenMP 4.0

- Released July 2013
  - A document of examples is expected to release soon

- Changes from 3.1 to 4.0 (Appendix E.1):
  - Accelerator: 2.9
  - SIMD extensions: 2.8
  - Places and thread affinity: 2.5.2, 4.5
  - Taskgroup and dependent tasks: 2.12.5, 2.11
  - Error handling: 2.13
  - User-defined reductions: 2.15
  - Sequentially consistent atomics: 2.12.6
  - Fortran 2003 support
Summary and Outlook

- We have seen features of OpenMP and some examples of their use
  - Powerful, flexible, portable API
  - Worksharing, synchronization; runtime routines for dynamic threadcount, nesting, ..

- What is coming?
  - OpenMP 4.0 implementations available, and more to come soon
  - OpenMP 4.1 specification to be released later this year; release candidate (TR3) available for your review and feedback at Openmp.org