OpenMP II

B. Estrade
<estrabd@lsu.edu>
Objectives

- learn about OpenMP's *work sharing* constructs, including:
  - loops
  - parallel sections
  - mutual exclusion
Work Sharing

- refers to the dividing up of tasks among multiple threads
- OpenMP provides methods to:
  - parallelize loops
  - define parallel sections
  - define critical sections
  - protect shared variables from conflicting updates
Synchronization

- **barrier**
  - no threads may move beyond this point in code until all threads have reached this point

- **nowait**
  - used in conjunction with the OMP loop clauses
  - instructs threads not to wait for the others to complete before moving on
  - used with `DO/FOR`, `section`, and `single` directives to remove implicit barriers

- Directives that force barriers, include: `master`, `order`, `critical`, and `atomic`; the latter 3 force thread serialization

- Barriers make parallel programs less efficient, but are a necessary part of most non-trivial applications
OpenMP's Flush Directive

- The flush directive is a special kind of synchronization point that marks a point in which a consistent view of the global memory must be provided.

- Even on cache-coherent systems, this directive is available due to it being required in the OpenMP standard that this be able to be forced explicitly by the user.

- In other words, when called, the flush directive ensures that all threads are aware of the most recent value of the shared variables which were 'flushed'.

```fortran
! 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)
   A(I) = A(I) + 1
   B(I) = B(I) + 1
ENDDO
!$OMP END DO

! Flush shared variables, A, B, and C
!$OMP FLUSH (A,B,C)

!$OMP END PARALLEL
PRINT *, C(10)
END
```
Parallelizing Loops

for (int i=1; i <= 99; i++) {
// do stuff
}

for (int i=1; i <= 33; i++) {
  // do stuff
}

for (int i=34; i <= 66; i++) {
  // do stuff
}

for (int i=67; i <= 99; i++) {
  // do stuff
}

Thread 0
i = 1 .. 33

Thread 1
i = 34 .. 66

Thread 2
i = 67 .. 99
Parallelizing Loops

- OpenMP allows for loops to be parallelized pretty easily
- A few caveats
  - the loop must be a structured block and not be terminated by a break statement
    ```c
    // this is not something to parallelize with OpenMP
    for (int i=0; i < 100; i++) {
        if (i > 50)
            break; // breaking when i gt than 50
    }
    BAD
    ```
  - values of the loop control expression must be the same for all iterations of the loop
    ```c
    // something else not to parallelize with OpenMP
    for (int i=0; i < 100; i++) {
        if (i == 50)
            i = 0; // resetting i when eq to 50
    }
    BAD
    ```
  - critical sections, the (atomic) updating of shared variables, and used of the `ordered` directive should be avoided
  - OpenMP works best when the loops are static and very straightforward
Parallelizing Loops - Fortran

```fortran
PROGRAM VECTOR_ADD
USE OMP_LIB
PARAMETER (N=100)
INTEGER N, I
REAL A(N), B(N), C(N)
CALL OMP_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
```

http://developers.sun.com/solaris/articles/studio_openmp.html
Parallelizing Loops - C/C++

```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
Parallel Loop Scheduling

- Scheduling refers to how iterations are assigned to a particular thread;
- There are 4 types:
  - **static**: each thread is assigned a chunk in a round-robin fashion
  - **dynamic**: each thread is initialized with a chunk, and gets assigned a new chunk of iterations on a first come, first finished basis
  - **guided**: iterations are divided into pieces that successfully decrease exponentially, with *chunk* being the smallest size
  - **runtime**: schedule is deferred to runtime, and is set using **OMP_SCHEDULE**
- Limitations
  - only one schedule type may be used at for a given loop
  - the chunk size applies to *all* threads
## Comparison of Loop Schedules

### Static and Dynamic

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Threads</th>
<th>Chunk Size</th>
<th>Iterations Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>0</td>
<td>1</td>
<td>3 3 3 3 3</td>
</tr>
<tr>
<td>Dynamic</td>
<td>0</td>
<td>1</td>
<td>3 3 3 3 3</td>
</tr>
</tbody>
</table>

(* repeated in round robin fashion *)

(* each thread gets 5 more iterations as it completes *)

### Guided

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Threads</th>
<th>Chunk Size</th>
<th>Round</th>
<th>Iterations Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0:</td>
<td>(100)/5</td>
<td>= 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1:</td>
<td>(100-20)/5 = 80/5 = 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 2:</td>
<td>(80-16)/5 = 74/5 = 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 3:</td>
<td>(74-14)/5 = 60/5 = 12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 4:</td>
<td>(60-12)/5 = 48/5 = 9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(* effectively reverses list of threads seeking work *)

(* because threads with fewer iterations should finish sooner *)

(* chunk size limit reached! *)

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Threads</th>
<th>Chunk Size</th>
<th>Round</th>
<th>Iterations Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Thread 4:</td>
<td>(48-9)/5 = 39/5 = 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 3:</td>
<td>(39-7)/5 = 32/5 = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 2:</td>
<td>(32-6)/5 = 26/5 = 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1:</td>
<td>(26-5)/5 = 21/5 = 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0:</td>
<td>(21-5)/5 = 16/5 = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(* done *)
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 PARALLEL
```

C/C++

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

*optional

schedule type  chunk of iterations*

---

B. Estrade <estrabd@lsu.edu>, HPC @ LSU – High Performance Computing Workshop
Parallel Loop Ordering

- Ordering refers to ensuring that a particular section *inside* of a loop must happen in the same order as if the loop was run sequentially.

- In general, this forces a serialization of threads, so loss of efficiency in *minimized* when a high percentage of the work done *per iteration* is *outside* of the *ordered* directive.

- Forces a serialization of threads, but this cost can be absorbed *if* a large percentage of time is spent in calculation, per iteration, before the ordered block.

```
#pragma omp parallel shared(a, b, c) private(i)
{
    #pragma omp for ordered
    for (i = 0; i <= 99; i++) {
        //
        // a lot of non-order dependent calculations
        // ...
        #pragma omp ordered
        {
            // some *really* quick set of instructions
        }
    }
}
```
Parallel Loop Ordering

(Example of 3 threads executing 3 iterations simultaneously)

Note: Provided that the ordered part of the loop is sufficiently small compared to the non-order dependent section of each iteration, the cost of the serialization per N threads can be minimized.
OpenMP Environmental Variables

- OMP_NUM_THREADS
  - required, informs execution of the number of threads to use

- OMP_SCHEDULE
  - The OMP_SCHEDULE environment variable applies to PARALLEL DO and work-sharing DO directives that have a schedule type of RUNTIME.

- OMP_DYNAMIC
  - The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions
Parallel Sections

- Parallel sections allow for blocks of code to be assigned to individual threads, which are executed *once*.

- This provides more fine grain control than a basic parallel loop, because in a loop, each iteration is essentially identical.

- There is an implicit barrier at the end of the sections definition, unless otherwise specified as `nowait`. 
Parallel Sections - Examples

Fortran

```fortran
PROGRAM SECTIONS
USE OMP_LIB
INTEGER SQUARE
INTEGER X, Y, Z, XS, YS, ZS
CALL OMP_SET_DYNAMIC (.FALSE.)
CALL OMP_SET_NUM_THREADS (3)
X = 2
Y = 3
Z = 5

!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
XS = SQUARE(X)
PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "XS =", XS
!$OMP SECTION
YS = SQUARE(Y)
PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "YS =", YS
!$OMP SECTION
ZS = SQUARE(Z)
PRINT *, "ID = ", OMP_GET_THREAD_NUM(), "ZS =", ZS
!$OMP END SECTIONS
!$OMP END PARALLEL
END

INTEGER FUNCTION SQUARE(N)
INTEGER N
SQUARE = N*N
END
```

C/C++

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

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

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

http://developers.sun.com/solaris/articles/studio_openmp.html
#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
Singling Out Threads

- OpenMP allows for the defining of blocks that are to be executed by the *master* thread, or simply the first thread to arrive at the block.
- This saves time and overhead if one needs to revert to a single thread for a particular code section.
- Unless specified, there is an implied barrier for all threads at the end of both *master* and *single* blocks.

Master

```
!$OMP MASTER
  block
!$OMP END MASTER
```

Single

```
!$OMP SINGLE
  block
!$OMP END SINGLE
```
Mutual Exclusion

- Mutual exclusion refers to restricting the execution of a block of code or the updating of a shared variable to only one thread at a time.

- In other words, it forces all threads to execute a particular block of code \textit{serially}.

- Mutual exclusion constructs, when used properly, provide for safe and efficient code execution.

- OpenMP provides two types of constructs to implement mutual exclusion in one's code:
  - \textit{critical sections}
  - \textit{atomic blocks}
Critical Sections

- A block of code defined inside of a critical section will be executed by each thread, one thread at a time.

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

- Critical sections make threads wait, so OpenMP allows for the creation of *named* critical sections that allow multiple threads to execute at a given time.

- The following code allows any 1 of 3 threads to take turns executing in a critical sections; if timed properly, idle time for each thread will be minimized due to this interleaving.

```c
#pragma omp critical(a)
{
    // some code
}
#pragma omp critical(b)
{
    // some code
}
#pragma omp critical(c)
{
    // some code
}
```
Critical Section - Example

```c
/* Compute values of array c in parallel. */
#pragma omp parallel shared(a, b, c) private(i)
{
    #pragma omp critical
    {
        //
        // do stuff (one thread at a time)
        //
    }
}
```

Note, thread order not guaranteed!
#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]);
        }
    }
  }
Each critical section is run by a single thread at a time, but multiple threads may be active at the same time.
Named Critical Sections Example

```c
#include <omp.h>

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

Note, thread order not guaranteed!
Atomic Variable Updating

- Forces for the atomic updating of a shared variable by a single thread at once
- This serializes updates to a particular variable, and protects it from multiple writes occurring at once; can be considered “mini” critical sections.

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int count = 0;
#pragma omp parallel shared(count)
    {
#pragma omp atomic
        count++;
    }
    printf_s("Number of threads: %d\n", count);
}
```

Note, thread order not guaranteed!

- The example above run without the atomic protection for count will allow for all threads to update the shared variable, causing unpredictable results.
- There is no such thing as a named atomic section to improve efficiency through interleaving, so if this is desired, use critical sections
References

- http://www.nersc.gov/nusers/help/tutorials/openmp
Additional Resources

- https://docs.loni.org/wiki/Introduction_to_OpenMP
- https://docs.loni.org/wiki/Advanced_OpenMP
- https://docs.loni.org/wiki/Using_OpenMP