Quick look at OpenMP

Goals

Understand that there are other ways of writing parallel programs

See examples of OpenMP

You are not expected to be able to use OpenMP from this presentation.

What Is OpenMP?

Portable, shared-memory threading API
- Fortran, C, and C++
- Multi-vendor support for both Linux and Windows

Standardizes task & loop-level parallelism

Supports coarse-grained parallelism

Combines serial and parallel code in single source

Standardizes ~ 20 years of compiler-directed threading experience

Programming Model

Fork-Join Parallelism:
- Master thread spawns a team of threads as needed
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
A Few Syntax Details to Get Started

Most of the constructs in OpenMP are compiler directives or pragmas

- For C and C++, the pragmas take the form:
  
  ```
  #pragma omp construct [clause [clause]...]
  ```

- For Fortran, the directives take one of the forms:

  ```
  C$OMP construct [clause [clause]...]
  !$OMP construct [clause [clause]...]
  !$OMP construct [clause [clause]...]
  ```

Header file or Fortran 90 module

```
#include "omp.h"
use omp_lib
```

Parallel Region & Structured Blocks (C/C++)

- Structured block: a block with one point of entry at the top and one point of exit at the bottom
- The only "branches" allowed are STOP statements in Fortran and exit() in C/C++

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if (conv (res[id]) goto done;
}
```

The Private Clause

Reproduces the variable for each task

- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
    float x, y; int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
```
Quick look at OpenMP

**Task Decomposition**

```plaintext
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf("%6.2f\n", bigboss(s, c));
```

alice, bob, and cy can be computed in parallel

**omp sections**

```c
#pragma omp sections
Must be inside a parallel region
Precedes a code block containing of \( N \) blocks of code that may be executed concurrently by \( N \) threads
Encompasses each omp section
```

```c
#pragma omp section
Precedes each block of code within the encompassing block described above
May be omitted for first parallel section after the parallel sections pragma
Enclosed program segments are distributed for parallel execution among available threads
```

**Functional Level Parallelism w sections**

```c
#pragma omp parallel sections
{
#pragma omp section /* Optional */
  a = alice();
#pragma omp section
  b = bob();
#pragma omp section
  c = cy();
}
```

```c
s = boss(a, b);
printf("%6.2f\n", bigboss(s, c));
```

**Advantage of Parallel Sections**

Independent sections of code can execute concurrently – reduce execution time
A race condition is nondeterministic behavior caused by the times at which two or more threads access a shared variable. For example, suppose both Thread A and Thread B are executing the statement

```
area += 4.0 / (1.0 + x*x);
```

### Protect Shared Data

Must protect access to shared, modifiable data.

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

### Two Timings

#### Value of area

<table>
<thead>
<tr>
<th>Value</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td></td>
<td>+3.765</td>
</tr>
<tr>
<td>15.432</td>
<td>+3.563</td>
<td></td>
</tr>
<tr>
<td>18.995</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Order of thread execution causes non-determinant behavior in a data race.

### OpenMP* Critical Construct

```
#pragma omp critical [[(lock_name)]]
```

Defines a critical region on a structured block.

Threads wait their turn—at a time, only one calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional.

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

Good Practice – Name all critical sections.
**OpenMP* Reduction Clause**

```
reduction (op : list)
```

The variables in "list" must be shared in the enclosing parallel region.

Inside parallel or work-sharing construct:
- A PRIVATE copy of each list variable is created and initialized depending on the "op"
- These copies are updated locally by threads
- At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

**Reduction Example**

```
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```

Local copy of `sum` for each thread
All local copies of `sum` added together and stored in "global" variable

**C/C++ Reduction Operations**

A range of associative operands can be used with reduction
Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>