Pragmas
- Pragma: a compiler directive in C or C++
- Stands for “pragmatic information”
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax: 
  ```
  #pragma omp <rest of pragma>
  ```

OpenMP Core Syntax
- Most of the constructs in OpenMP are compiler directives
  - ```
  #pragma omp construct [clause [clause] ... ]
  ```
- #pragma omp parallel num_threads(4)
- Function prototypes and types in the file: `#include <omp.h>`
- Most OpenMP constructs apply to a “structured block”
  - 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 a structured block

A Digression: Enabling OpenMP
- In Visual Studio
  - Open the project’s Property Pages dialog box.
  - Expand the Configuration Properties node.
  - Expand the C/C++ node.
  - Select the Language property page.
  - Modify the OpenMP Support property.
- In GCC
  - Compile using the `-fopenmp` option
    ```
    gcc -fopenmp -o hello hello.c
    ```

Example: Hello World
- Verify that your OpenMP environment works
- Write a multithreaded program where each thread prints “hello world”

OpenMP Thread Interaction
- How do threads interact?
  - OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables
  - Unintended sharing of data causes race conditions
    - Race condition: when the program’s outcome changes as the threads are scheduled differently
  - To control race conditions:
    - Use synchronization to protect data conflicts
    - Synchronization is expensive so:
      - Change how data is accessed to minimize the need for synchronization
Shared-memory Model vs. Message-passing Model (#1)
- **Shared-memory model**
  - Number active threads 1 at start and finish of program, changes dynamically during execution
- **Message-passing model**
  - All processes active throughout execution of program

Incremental Parallelization
- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)
- **Shared-memory model**
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted
- **Message-passing model**
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps

Creating Threads

OpenMP Programming Model
- **Fork-Join Parallelism**
  - Master thread spawns a team of threads as needed
  - Parallelism added incrementally until performance goals are met
  - i.e., the sequential program evolves into a parallel program

Function `omp_get_num_procs`
- Returns number of physical processors available for use by the parallel program

```
int omp_get_num_procs (void)
```
Function `omp_set_num_threads`
- Uses the parameter value to set the number of threads to be active in parallel sections of code
- May be called at multiple points in a program

```c
void omp_set_num_threads (int t)
```

Thread Creation: Parallel Regions
- You create threads in OpenMP with the parallel construct
- For example, to create a 4 thread parallel region:

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

How Would You Solve This Problem?

**Numerical Integration**
Mathematically, we know that:

\[
\int_{a}^{b} f(x) \, dx = \pi
\]

We can approximate the integral as a sum of rectangles:

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

Where each rectangle has width \(\Delta x\) and height \(F(x)\) at the middle of interval \(i\).
Synchronization
- Synchronization is used to impose order constraints and to protect access to shared data
- Mutual exclusion: Only one thread at a time can enter the critical region

```
float x;

#pragma omp parallel
{
  int i, id, thrcnt;
  id = omp_get_thread_num();
  create = omp_get_num_threads();
  thread = norm(id, creating);
  B = big_thing();
  #pragma omp critical
  x += B;
}
```

Synchronization (cont.)
- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of $X$ in this example)

```
#pragma omp parallel
{
  double tmp, B;
  B = DOT();
  tmp = big_thing(B);
  #pragma omp atomic
  X += tmp;
}
```

Numerical Integration Example Revisited
- In the numerical integration example, an array is used to create space for each thread to store its partial sum
- If the array elements happen to share a cache line, this leads to false sharing
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence “thrashing independent data” back and forth between threads
- How would we modify this solution?

Parallelizing Loops
- A parallel construct by itself creates a SPMD or “Single Program Multiple Data” program
  - Each thread redundantly executes the same code
  - How do you split up pathways through the code between threads within a team?
    - This is called worksharing

Loop Worksharing Constructs
- The loop worksharing construct splits up loop iterations among the threads in a team

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

SPMD vs. Worksharing
- A parallel construct by itself creates a SPMD or “Single Program Multiple Data” program
  - Each thread redundantly executes the same code
Parallel for Loops

- C programs often express data-parallel operations as for loops
  
  ```
  for (i = first; i < size; i += prime)
      marked[i] = 1;
  ```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel

- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

A Motivating Example

```
Sequential code
for(i=0;i<N;++i)
    a[i] = b[i] + c[i];
```

```
#pragma omp parallel for
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent
  - So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, M[MAX];
for (i=0; i<MAX; i++)
    for (j=2; j<=log2(M[i]); j++)
        M[i] = log2(M[i]);
```

Reductions

- A reduction is to combine values into a single accumulation variable

- Support for reduction operations is included in most parallel programming environments

- OpenMP reduction clause

Reductions (cont.)

```
double ave=0.0, A[MAX];
int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i<MAX; i++)
    ave += A[i];
ave = ave/MAX;
```
Reductions (cont.)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>1</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
</tbody>
</table>

Performance Improvement #1
- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
- Inversion does not significantly lower cache hit rate

Performance Improvement #2
- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The if clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,
  
  ```
  #pragma omp parallel for if(n > 5000)
  ```

Performance Improvement #3
- We can use `schedule` clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop’s execution. Remaining iterations allocated to threads that complete their assigned iterations.

Data Sharing

Execution Context
- Every thread has its own execution context
- Execution context: address space containing all of the variables a thread may access
- Contents of execution context:
  - static variables
  - dynamically allocated data structures in the heap
  - variables on the run-time stack
  - additional run-time stack for functions invoked by the thread
Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

Declaring Private Variables

for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][k]+tmp);

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable j

Example Use of private Clause

#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j], a[i][k]+tmp);

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private
  
  ```
  private ( <variable list> )
  ```

firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable’s value in an iteration, subsequent iterations will get the modified value
Lastprivate Clause
- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- **lastprivate** clause: used to copy back to the master thread’s copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration

Critical Sections

```c
double area, pi, x;
int i, n;
... 
area = 0.0;
for (i = 0; i < n; i++) {
x += (i+0.5)/n;
area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition
- Consider this C program segment to compute using the rectangle rule:

```c
double area, pi, x;
int i, n;
... 
area = 0.0;
for (i = 0; i < n; i++) {
x = (i+0.5)/n;
area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition (cont.)
- If we simply parallelize the loop...

```c
double area, pi, x;
int i, n;
... 
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
x = (i+0.5)/n;
area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition (cont.)
- ... we set up a race condition in which one process may “race ahead” of another and not see its change to shared variable **area**

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

Race Condition Time Line
**critical Pragma**

- Critical section: a portion of code that only thread at a time may execute

- We denote a critical section by putting the pragma

  ```c
  #pragma omp critical
  ```

  in front of a block of C code

---

**Correct, But Inefficient, Code**

double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
  x = (i+0.5)/n;
  #pragma omp critical
  area += 4.0/(1.0 + x*x);
}
pi = area / n;

---

**π-finding Code with Reduction Clause**

double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for 
private(x) reduction(+:area)
for (i = 0; i < n; i++) {
  x = (i + 0.5)/n;
  area += 4.0/(1.0 + x*x);
}
pi = area / n;

---

**Barrier**

- Barrier: Each thread waits until all threads have arrived

  ```c
  #pragma omp parallel shared(A, B, C) private(id)
  {
    id = omp_get_thread_num();
    A[id] = big_calc1(id);
    #pragma omp barrier
    barrier A for worksharing construct
    #pragma omp for nowait
    for (i=0; i<n; i++) C[i] = big_calc3(A, C);
    A[id] = big_calc4(id);
  }
  ```

  ```c
  implicit barrier at the end of a parallel region
  no implicit barrier due to nowait
  ```

---

**Master Construct**

- The master construct denotes a structured block that is only executed by the master thread
- The other threads just skip it

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

---

**Single Worksharing Construct**

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread)
- Barrier is implied at the end of the single block

  ```c
  #pragma omp parallel
  {
    do_many_things();
    #pragma omp single
    {
      exchange_boundaries();
    }
    do_many_other_things();
  }
  ```
Ordered Construct
- The ordered region executes in the sequential order

```c
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction (+:res)
for (i=0;i<N;++i)
    
#pragma omp ordered
res += consum(tmp);
```

Functional Parallelism Example

```c
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%.2f\n", epsilon(x,y));
```

May execute alpha, beta, and delta in parallel

parallel sections Pragma
- Precedes a block of $k$ blocks of code that may be executed concurrently by $k$ threads
- Syntax:

```c
#pragma omp parallel sections
```

section Pragma
- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```c
#pragma omp section
```

Example of parallel sections

```c
#pragma omp parallel sections
{
#pragma omp section /* Optional */
v = alpha();
#pragma omp section
w = beta();
#pragma omp section
y = delta();
}
x = gamma(v, w);
printf ("%.2f\n", epsilon(x,y));
```

Another Approach

Execute alpha and beta in parallel. Execute gamma and delta in parallel.
sections Pragma
- Appears inside a parallel block of code
- Has same meaning as the `parallel sections` pragma
- If multiple `sections` pragmas inside one parallel block, may reduce fork/join costs

Use of sections Pragma
```c
#pragma omp parallel
{
  #pragma omp sections
  {
    v = alpha();
    #pragma omp section
    w = beta();
  }
  #pragma omp sections
  {
    x = gamma(v, w);
    #pragma omp section
    y = delta();
  }
  printf("%6.2f\n", epsilon(x,y));
}
```

Conclusions
- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause

Summary (1/3)
- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling

Summary (2/3)

Summary (3/3)

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suitable for multiprocessors</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitable for multicomputers</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Supports incremental parallelization</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Minimal extra code</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explicit control of memory hierarchy</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>