Parallel Programming with OpenMP

- OpenMP (Open Multi-Processing) is a popular shared-memory programming model
- Supported by popular production C (also Fortran) compilers: Clang, GNU Gcc, IBM xlc, Intel icc
- These slides borrow heavily from Tim Mattson’s excellent OpenMP tutorial available at www.openmp.org, and from Jeffrey Jones (OSU CSE 5441)
What is OpenMP?

• A directive based parallel programming model
  – OpenMP program is essentially a sequential program augmented with compiler directives to specify parallelism
  – Eases conversion of existing sequential programs

• Main concepts:
  – Parallel regions: where parallel execution occurs via multiple concurrently executing threads
  – Each thread has its own program counter and executes one instruction at a time, similar to sequential program execution
  – Shared and private data: shared variables are the means of communicating data between threads
  – Synchronization: Fundamental means of coordinating execution of concurrent threads
  – Mechanism for automated work distribution across threads
OpenMP Core Syntax

• Most of the constructs in OpenMP are compiler directives:
  – #pragma omp construct [clause [clause]…]

• Example
  – #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 surrounded by “{ }”, with one point of entry at the top and one point of exit at the bottom.
Hello World in OpenMP

```c
#include <omp.h>
void main()
{
    #pragma omp parallel
    {
        int ID = 0;
        printf(" hello(%d ) \n", ID);
        printf(" world(%d ) \n", ID);
    }
}
```

- An OpenMP program starts with one “master” thread executing “main” as a sequential program
- “#pragma omp parallel” indicates beginning of a parallel region
  - Parallel threads are created and join the master thread
  - All threads execute the code within the parallel region
  - At the end of parallel region, only master thread executes
  - Implicit “barrier” synchronization; all threads must arrive before master proceeds onwards
Hello World in OpenMP

```c
#include <omp.h>
void main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf(" hello(%d ) ", ID);
        printf(" world(%d ) \n", ID);
    }
}
```

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

• Each thread has a unique integer “id”; master thread has “id” 0, and other threads have “id” 1, 2, …
• OpenMP runtime function `omp_get_thread_num()` returns a thread’s unique “id”.
• The function `omp_get_num_threads()` returns the total number of executing threads
• The function `omp_set_num_threads(x)` asks for “x” threads to execute in the next parallel region (must be set outside region)
Work Distribution in Loops

- Basic mechanism: threads can perform disjoint work division using their thread ids and knowledge of total # threads

```c
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    for (int i = t_id; i < 1000; i += omp_get_num_threads())
    {
        A[i] = foo(i);
    }
}
```

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int t_id = omp_get_thread_num();
    int b_size = 1000 / omp_get_num_threads();
    for (int i = t_id * b_size; i < (t_id+1) * b_size; i ++)
    {
        A[i] = foo(i);
    }
}
```

- Cyclic work distribution
- Block distribution of work
Specifying Number of Threads

- Desired number of threads can be specified in many ways
  - Setting environmental variable `OMP_NUM_THREADS`
  - Runtime OpenMP function `omp_set_num_threads(4)`
  - Clause in `#pragma` for parallel region

```c
double A[1000];

#pragma omp parallel num_threads(4)
{
    int t_id = omp_get_thread_num();
    for (int i = t_id; i < 1000; i += omp_get_num_threads())
    {
        A[i] = foo(i);
    }
}
```

Each thread will execute the code within the block.
OpenMP Data Environment

- Global variables (declared outside the scope of a parallel region) are **shared** among threads unless explicitly made private
- Automatic variables declared within parallel region scope are **private**
- Stack variables declared in functions called from within a parallel region are **private**

```c
#pragma omp parallel private(x)
```

- each thread receives its own **uninitialized** variable `x`
- the variable `x` falls out-of-scope after the parallel region
- a global variable with the same name is unaffected (3.0 and later)

```c
#pragma omp parallel firstprivate(x)
```

- `x` must be a global-scope variable
- each thread receives a **by-value copy** of `x`
- the local `x`’s fall out-of-scope after the parallel region
- the base global variable with the same name is unaffected
Example: Numerical Integration

Mathematically:

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

Which can be approximated by:

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

where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
int num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i = 0; i < num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
SPMD Programming

• **Single Program Multiple Data**
  – Each thread runs same program
  – Selection of data, or branching conditions, based on thread id

• in OpenMP implementation:
  • perform work division in parallel loops
  • query **thread_id** and **num_threads**
  • partition work among threads
Parallel Accumulation: Avoiding Race Conditions

\[ \text{sum} = \text{sum} + 4.0/(1.0+x*x); \]

- **load_register 1, \@sum**
- **set_register 2, 4.0**
- **set_register 3, 1.0**
- **load_register 4, \@x**
- **multiply 5, 4, 4**
- **add 4, 3, 5**
- **divide 3, 2, 4**
- **add 2, 1, 3**
- **store 2, \@sum**

- High-level C statement translates into a sequence of low-level instructions
  - Accumulation into shared variable sum is not atomic: contributions can be lost if multiple threads execute the statements concurrently
  - Must use suitable synchronization to avoid race conditions
#include <omp.h>
int num_steps = 100000;
double step;
#define NUM_THREADS 2

void main ()
{
  int i, nthreads;
double pi = 0.0, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);

  for (i = 0; i < num_steps; i += nthreads) {
    x = (i+0.5)*step;
    sum[id] += 4.0/(1.0+x*x);
  }
  for (i = 0; i < nthreads; i++)
  {
    pi += sum[i] * step;
  }
}
Avoiding False Sharing in Cache

```
sum[id] += 4.0/(1.0+x*x);
sum[id] = sum[id] + 4.0/(1.0+x*x);
```

- Array `sum[]` is a shared array, with each thread accessing exactly one element.
- Cache line holding multiple elements of `sum` will be locally cached by each processor in its private L1 cache.
- When a thread writes into an element in `sum`, the entire cache line becomes “dirty” and causes invalidation of that line in all other processor’s caches.
- Cache thrashing due to this “false sharing” causes performance degradation.
Block vs. Cyclic Work Distribution

- Block/cyclic work distribution will not impact performance here
- But if statement in loop were like: “A[i] += B[i]*C[i]”, block distribution would be preferable
Synchronization: Critical Sections

- Only one thread can enter critical section at a time; others are held at entry to critical section
- Prevents any race conditions in updating “res”

```c
float res;
#pragma omp parallel
{
    float B;
    int i, id, nthrds;

    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for( i = id;  i < MAX;  i += nthrds)
    {
        B = big_job(i);
        #pragma omp critical
        consume (B, res);
    }
}
```
Synchronization: Atomic

- Atomic: very efficient critical section for simple accumulation operations (x binop= expr; or x++, x--, etc.)
- Used hardware atomic instructions for implementation; much lower overhead than using critical section
int num_steps = 100000;
double step;
#define NUM_THREADS 2

void main()
{
    int i, nthreads;
double pi = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

    id = omp_get_thread_num();
    nthreads = omp_get_num_threads();
    if (id == 0) nthreads = nthreads;
    sum = 0.0;
    for (i = id; i < num_steps; i += nthreads)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    #pragma omp atomic
    {
        pi += sum * step;
    }
}
Loop worksharing constructs
A motivating example

Sequential code

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

OpenMP parallel region

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

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel
#pragma omp for
    for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```
Often a parallel region has a single work-shared loop

Combined construct for such cases: just add the work-sharing “for” clause to the parallel region pragma
Loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - `schedule(static [,chunk])`
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - `schedule(dynamic [,chunk])`
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
</tbody>
</table>

Least work at runtime: scheduling done at compile-time

Most work at runtime: complex scheduling logic used at run-time
# Loop Work-Sharing Constructs: The Schedule Clause

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Pre-determined and predictable by the programmer</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
<tr>
<td>AUTO</td>
<td>When the runtime can &quot;learn&quot; from previous executions of the same loop</td>
</tr>
</tbody>
</table>

- **STATIC**: Least work at runtime; scheduling done at compile-time.
- **DYNAMIC**: Most work at runtime; complex scheduling logic used at run-time.
OpenMP Reductions

- Reductions commonly occur in codes (as in pi example)
- OpenMP provides special support via “reduction” clause
  - OpenMP compiler automatically creates local variables for each thread, and divides work to form partial reductions, and code to combine the partial reductions
  - Predefined set of associative operators can be used with reduction clause, e.g., +, *, -, min, max

```c
double avg = 0.0;
double A[SIZE];
#pragma omp parallel for
for (int i = 0; i < SIZE; i++)
{
    avg += A[i];
}
avg = avg / SIZE;
```
OpenMP Reductions

double avg = 0.0;
double A[SIZE];
#pragma omp parallel for reduction(+ : avg)
for (int i = 0; i < SIZE; i++)
{
    avg += A[i];
}
avg = avg / SIZE;

- Reductions clause specifies an operator and a list of reduction variables (must be shared variables)
  - OpenMP compiler creates a local copy for each reduction variable, initialized to operator’s identity (e.g., 0 for +; 1 for *)
  - After work-shared loop completes, contents of local variables are combined with the “entry” value of the shared variable
  - Final result is placed in shared variable
int num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i = 0; i < num_steps; i++)
    {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}

int num_steps = 100000;
double step;

#define NUM_THREADS 2

void main ()
{
    int i;
    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel for private(x) reduction( + : sum)
    for (i = 0; i < num_steps; i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi += sum * step;
}
OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

<table>
<thead>
<tr>
<th>Operator</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>min</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>max</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.IEOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IOR.</td>
<td>0</td>
</tr>
<tr>
<td>.IAND.</td>
<td>All bits on</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</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></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization: Barrier

```c
#pragma omp parallel 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_calc4(id);
}
```
#pragma omp parallel
{
  do_many_things();

  #pragma omp master
  {
    reset_boundaries();
  }

do_many_other_things();
}

#pragma omp parallel
{
  do_many_things();

  #pragma omp single
  {
    reset_boundaries();
  }

do_many_other_things();
}
Synchronization: Locks

```c
omp_lock_t lck;
omp_init_lock(&lck);

#pragma omp parallel
{
    do_many_things();
    omp_set_lock(&lck);
    {code requiring mutual exclusion}
    omp_unset_lock(&lck);
    do_many_other_things();
}
omp_destroy_lock(&lck);
```

- Alternate way to critical sections of achieving mutual exclusion
- More flexible than critical sections (can use multiple locks)
- More error-prone – for example, deadlock if a thread does not unset a lock after acquiring it
OpenMP Sections

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

- Work-sharing for functional parallelism; complementary to “omp for” for loops
OpenMP memory model

- OpenMP supports a shared memory model
- All threads share an address space, but it can get complicated:

- Multiple copies of data may be present in memory, various levels of cache, or in registers
OpenMP and relaxed consistency

- OpenMP supports a **relaxed-consistency** shared memory model
  - Threads can maintain a **temporary view** of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent only at certain points in the program
  - The operation that enforces consistency is called the **flush operation**
Flush operation

• A flush is a sequence point at which a thread is guaranteed to see a consistent view of memory
  – All previous read/writes by this thread have completed and are visible to other threads
  – No subsequent read/writes by this thread have occurred

• A flush operation is analogous to a fence in other shared memory APIs
Flush and synchronization

• A flush operation is implied by OpenMP synchronizations, e.g.,
  – at entry/exit of parallel regions
  – at implicit and explicit barriers
  – at entry/exit of critical regions
  ….
  (but not at entry to worksharing regions)

This means if you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:

• the writing threads follow the writes with a construct that implies a flush.
• the reading threads precede the reads with a construct that implies a flush.

This is a rare event … or putting this another way, you should avoid writing code that depends on ordering reads/writes around flushes.
<table>
<thead>
<tr>
<th>OpenMP pragma, function, or clause</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma omp parallel</td>
<td>Parallel region, teams of threads, structured block, interleaved execution across threads</td>
</tr>
<tr>
<td>int omp_get_thread_num()</td>
<td>Create threads with a parallel region and split up the work using the number of threads and thread ID</td>
</tr>
<tr>
<td>int omp_get_num_threads()</td>
<td>Speedup and Amdahl's law. False Sharing and other performance issues</td>
</tr>
<tr>
<td>double omp_get_wtime()</td>
<td>Internal control variables. Setting the default number of threads with an environment variable</td>
</tr>
<tr>
<td>setenv OMP_NUM_THREADS N</td>
<td>Synchronization and race conditions. Revisit interleaved execution.</td>
</tr>
<tr>
<td>#pragma omp barrier</td>
<td>Worksharing, parallel loops, loop carried dependencies</td>
</tr>
<tr>
<td>#pragma omp critical</td>
<td>Reductions of values across a team of threads</td>
</tr>
<tr>
<td>#pragma omp for</td>
<td>Loop schedules, loop overheads and load balance</td>
</tr>
<tr>
<td>#pragma omp parallel for</td>
<td>Data environment</td>
</tr>
<tr>
<td>reduction(op:list)</td>
<td>Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive)</td>
</tr>
<tr>
<td>schedule(dynamic [,chunk])</td>
<td>Workshare with a single thread</td>
</tr>
<tr>
<td>schedule (static [,chunk])</td>
<td>Tasks including the data environment for tasks.</td>
</tr>
<tr>
<td>private(list), firstprivate(list), shared(list)</td>
<td></td>
</tr>
<tr>
<td>nowait</td>
<td></td>
</tr>
</tbody>
</table>
Books about OpenMP

- A book about OpenMP by a team of authors at the forefront of OpenMP’s evolution.

- A book about how to “think parallel” with examples in OpenMP, MPI and Java.