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 )
", ID);
        printf(" world(%d ) 
", 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

• 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)

```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)
```
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);
    }
}
```

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);
    }
}
```

- implicit barrier

  { 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

```plaintext
sum = 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
```c
#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);

    #pragma omp parallel
    {
        int i, id, nt;
double x;

        id = omp_get_thread_num();
        nt = omp_get_num_threads();
        if (id == 0) nthreads = nt;

        sum[id] = 0.0;
        for ( i = id; i < num_steps; i += nt)
        {
            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

- 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.

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

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

```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 atomic
    res += B;
  }
}
```
Parallel pi: No False Sharing

```c
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);

    #pragma omp parallel
    {
        int i, id, nthrds;
double x, sum;
        id = omp_get_thread_num();
nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;
        sum = 0.0;
        for ( i = id; i < num_steps; i += nthrds)
        {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
            #pragma omp atomic
            {
                pi += sum * step;
            }
        }
    }
}
```

no array, no false sharing -> sum is now local
^ each thread adds its partial sum one thread at a time
OpenMP Loop Work-Sharing

- Loop structure in parallel region is same as sequential code
- No explicit thread-id based work division by user; instead system automatically divides loop iterations among threads
- User can control work division: block, cyclic, block-cyclic, etc., via “schedule” clause in pragma

```c
float res;
#pragma omp parallel
{
    // id = omp_get_thread_num();
    // nthrds = omp_get_num_threads();
    // for( i = id;  i < MAX;  i + nthrds)
    #pragma omp for
    for( i = 0;  i < MAX;  i++)
    {
        B = big_job(i);
        #pragma omp critical
        consume (B, res);
    }
}
```
• 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

```c
#pragma omp parallel
{
#pragma omp for
for( i = 0;  i < MAX;  i++)
{
    B = big_job(i);
    #pragma omp critical
    consume (B, res);
}
}
```
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
going for (int i = 0; i < SIZE; i++)
{
    avg += A[i];
}
avg = avg / SIZE;
```
OpenMP Reductions

• 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

```c
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;
```
```c
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;
}
```

```c
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;
}
```
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);
}
```
Synchronization: Master and Single

```c
#pragma omp parallel
{
    do_many_things();

    #pragma omp master
    {
        reset_boundaries();
    }

do_many_other_things();
}
```

Multiple threads of control → Only master thread executes this region

```
#pragma omp parallel
{
    do_many_things();

    #pragma omp single
    {
        reset_boundaries();
    }

do_many_other_things();
}
```

Multiple threads of control → A single thread is chosen to execute this region

Implicit barrier

Multiple threads of control
Synchronization: Locks

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

multiple threads of control ➔ each section assigned to a different thread

by default: extra threads are idled
Controlling Work Distribution: Schedule Clause

- The schedule clause determines how loop iterators are mapped onto threads
  - `#pragma omp parallel for schedule( static [, chunk] )`
  - fixed-sized chunks assigned (alternating) to `num_threads`
  - typical default is: `chunk = iterations / num_threads`
  - set `chunk = 1` for cyclic distribution
  - `#pragma omp parallel for schedule( dynamic [, chunk] )`
  - run-time scheduling (with associated overhead)
  - each thread grabs “chunk” iterations off queue until all iterations have been scheduled
  - good load-balancing for uneven workloads
  - `#pragma omp parallel for schedule( guided[, chunk] )`
  - threads dynamically grab blocks of iterations
  - chunk size starts relatively large, to get all threads busy with good amortization of overhead
  - subsequently, chunk size is reduced to produce good workload balance
  - `#pragma omp parallel for schedule( runtime )`
  - schedule and chunk size taken from environment variable or from runtime library routines