Introduction to OpenMP

- What is OpenMP?
  - Open specification for Multi-Processing
  - “Standard” API for defining multi-threaded shared-memory programs
    - [www.openmp.org](http://www.openmp.org) – Talks, examples, forums, etc.

- High-level API
  - Preprocessor (compiler) directives (~ 80%)
  - Library Calls (~ 19%)
  - Environment Variables (~ 1%)
A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C or Fortran)

- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than T concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs

- OpenMP will not:
  - Parallelize (or detect!) dependencies
  - Guarantee speedup
  - Provide freedom from data races

Outline

- Introduction
  - Motivating example
  - Parallel Programming is Hard

- OpenMP Programming Model
  - Easier than PThreads

- Microbenchmark Performance Comparison
  - vs. PThreads

- Discussion
  - specOMP
Current Parallel Programming

1. Start with a parallel algorithm
2. Implement, keeping in mind:
   • Data races
   • Synchronization
   • Threading Syntax
3. Test & Debug
4. Debug
5. Debug

Motivation – Threading Library

```c
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}

int main() {
    pthread_attr_t attr;
    pthread_t threads[16];
    int tn;
    pthread_attr_init(&attr);
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], &attr, SayHello, NULL);
    }
    for(tn=0; tn<16; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```
Motivation

• Thread libraries are hard to use
  – P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  – Programmer must code with multiple threads in mind

• Synchronization between threads introduces a new dimension of program correctness

Motivation

• Wouldn't it be nice to write serial programs and somehow parallelize them “automatically”?
  • OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence
  • OpenMP is a small API that hides cumbersome threading calls with simpler directives
Better Parallel Programming

1. Start with some algorithm
   - Embarrassing parallelism is helpful, but not necessary

2. Implement serially, ignoring:
   - Data Races
   - Synchronization
   - Threading Syntax

3. Test and Debug

4. Automatically (magically?) parallelize
   - Expect linear speedup

Motivation – OpenMP

```c
int main() {

  // Do this part in parallel

  printf( "Hello, World!\n" );

  return 0;
}
```
Motivation – OpenMP

```c
int main() {
    omp_set_num_threads(16);
    // Do this part in parallel
#pragma omp parallel
    {
        printf( "Hello, World!\n" );
    }
    return 0;
}
```

OpenMP Parallel Programming

1. Start with a parallelizable algorithm
   • Embarrassing parallelism is good, loop-level parallelism is necessary
2. Implement serially, mostly ignoring:
   • Data Races
   • Synchronization
   • Threading Syntax
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
   • Hope for linear speedup
5. Test and Debug
Programming Model - Threading

- Serial regions by default, annotate to create *parallel* regions
  - Generic parallel regions
  - Parallelized loops
  - Sectioned parallel regions

- Thread-like Fork/Join model
  - Arbitrary number of *logical* thread creation/ destruction events

int main() {
    // serial region
    printf("Hello...\n");
    // parallel region
    #pragma omp parallel
    {
        printf("World\n");
    }
    // serial again
    printf("!\n");
}
Programming Model – Nested Threading

- Fork/Join can be nested
  - Nesting complication handled “automagically” at compile-time
  - Independent of the number of threads actually running

Programming Model – Thread Identification

Master Thread
- Thread with ID=0
- Only thread that exists in sequential regions
- Depending on implementation, may have special purpose inside parallel regions
- Some special directives affect only the master thread (like `master`)
Example

```c
int main() {
    int tid, nthreads;
    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel private(nthreads, tid)
    {
        printf("Hello, World!\n");
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("I'm the master, Number of threads = %d\n", nthreads);
        }
    }

    return 0;
}
```

Programming Model – Data/Control Parallelism

- **Data parallelism**
  - Threads perform similar functions, guided by thread identifier

- **Control parallelism**
  - Threads perform differing functions
    - One thread for I/O, one for computation, etc…
Programming model: Summary

Fork and Join: Master thread spawns a team of threads as needed

- Parallel regions where child threads are:
  -- spawned upon entering
  -- released upon exiting

- Thread 0, or master thread, performs tasks in both serial and parallel regions

Memory Model

- Shared memory communication
  - Threads cooperates by accessing shared variables
- The sharing is defined syntactically
  - Any variable that is seen by two or more threads is shared
  - Any variable that is seen by one thread only is private
- Race conditions possible
  - Use synchronization to protect from conflicts
  - Change how data is stored to minimize the synchronization
OpenMP easily parallelizes loops
- No data dependencies between iterations!

Preprocessor calculates loop bounds for each thread directly from serial source

```c
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
    printf("Foo");
}
```
The problem

- Executes the same code as many times as there are threads
- How many threads do we have? `omp_set_num_threads(n)`
  What is the use of repeating the same work n times in parallel? Can use `omp_thread_num()` to distribute the work between threads.
- D is shared between the threads, i and sum are private

```c
double D[1000];
#pragma omp parallel
{
    int i; double sum = 0;
    for (i=0; i<1000; i++) sum += D[i];
    printf("Thread %d computes %f\n", omp_thread_num(), sum);
}
```

Programming Model – Concurrent Loops

<table>
<thead>
<tr>
<th>Sequential code</th>
<th>(Semi) manual parallelization</th>
<th>Automatic parallelization</th>
</tr>
</thead>
</table>
| `for (int i=0; i<N; i++) { a[i]=b[i]+c[i]; }` | `#pragma omp parallel
{`
  | `int id = omp_get_thread_num();
  int Nthr = omp_get_num_threads();
  int istart = id*N/Nthr;
  int iend = (id+1)*N/Nthr;
  for (int i=istart; i<iend; i++) {
    a[i]=b[i]+c[i];
  }
}` | `#pragma omp parallel for schedule(static)
{`
  | `for (int i=0; i<N; i++) {
    a[i]=b[i]+c[i];
  }
` |
Programming Model – Concurrent Loops

- Used to assign each thread an independent set of iterations
- Threads must wait at the end
- Can combine the directives:
  - #pragma omp parallel for
- Only simple kinds of for loops:
  - Only one signed integer variable
  - Initialization: var=init
  - Comparison: var op last
  - Increment: var++, var--,
    var+=incr, var-=incr, etc.

Load balancing
- If all the iterations execute at the same speed, the processors are used optimally. If some iterations are faster than others, some processors may get idle, reducing the speedup
- We don’t always know the distribution of work, may need to re-distribute dynamically

Granularity
- Thread creation and synchronization takes time. Assigning work to threads on per-iteration resolution may take more time than the execution itself! Need to coalesce the work to coarse chunks to overcome the threading overhead
- Trade-off between load balancing and granularity!
Controlling Granularity

- `#pragma omp parallel if (expression)`
  - Can be used to disable parallelization in some cases (when the input is determined to be too small to be beneficially multithreaded)
- `#pragma omp num_threads (expression)`
  - Control the number of threads used for this parallel region

Programming Model – Loop Scheduling

- `schedule` clause determines how loop iterations are divided among the thread team
  - `static([chunk])` divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default `[chunk]` is `ceil( # iterations / # threads )`
  - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default `[chunk]` is 1
  - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation
The function TestForPrime (usually) takes little time but can take long, if the number is a prime indeed.

Solution: use dynamic, but with chunks.
Work sharing: Sections

```
answer1 = long_computation_1();
answer2 = long_computation_2();
if (answer1 != answer2) { ... }
```

- How to parallelize? These are just two independent computations!

```
#pragma omp sections
{
    #pragma omp section
    answer1 = long_computation_1();
    #pragma omp section
    answer2 = long_computation_2();
    if (answer1 != answer2) { ... }
}
```

Sections

- The **SECTIONS** directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

- Independent **SECTION** directives are nested within a **SECTIONS** directive.
  - Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible that for a thread to execute more than one section if it is quick enough and the implementation permits such.
Example

#include <omp.h>
#define N 1000

main ()
{
  int i;
  float a[N], b[N], c[N], d[N];
  /* Some initializations */
  for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
  
  #pragma omp parallel shared(a,b,c,d) private(i)
  {
    #pragma omp sections
    {
      #pragma omp section
      for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
      #pragma omp section
      for (i=0; i < N; i++)
        d[i] = a[i] * b[i];
    } /* end of sections */
  } /* end of parallel section */
}
Data Sharing

- Shared Memory programming model
  - Most variables are **shared by default**
  - We can define a variable as **private**

```c
// Do this part in parallel
#pragma omp parallel private(nthreads, tid)
{
    printf( "Hello, World!\n" );
    if (tid == 0)
    {
        ...
    }
}
```

Programming Model – Data Sharing

- Parallel programs often employ two types of data
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)

- **PThreads:**
  - Global-scoped variables are shared
  - Stack-allocated variables are private

- **OpenMP:**
  - shared variables are shared
  - private variables are private

```c
int bigdata[1024];

void* foo(void* bar) {
    int tid;
    #pragma omp parallel \
    shared ( bigdata ) \
    private ( tid )
    {
        /* Calc. here */
    }
}
```
Programming Model – Data Sharing

- **private:**
  - A copy of the variable is created for each thread.
  - No connection between the original variable and the private copies
  - Can achieve the same using variables inside {}

```c
Int i;
#pragma omp parallel for private(i)
for (i=0; i<n; i++) { ... }
```

Programming Model – Data Sharing

- **Firstprivate:**
  - Same, but the initial value is copied from the main copy
- **Lastprivate:**
  - Same, but the last value is copied to the main copy

```c
int idx=1;
int x = 10;
#pragma omp parallel for 
firstprivate(x) 
lastprivate(idx)
for (i=0; i<n; i++) {
    if (data[i]==x) idx = i;
}
```
Thread private

- Similar to private, but defined per variable
  - Declaration immediately after variable definition.
  - Must be visible in all translation units. Persistent between parallel sections.
  - Can be initialized from the master's copy with
  - #pragma omp copyin
- More efficient than private, but a global variable!

```c
int data[100];
#pragma omp threadprivate(data)
...
#pragma omp parallel for copyin(data)
for (...)"
```

Synchronization

- What should the result be (assuming 2 threads)?

```c
X=0;
#pragma omp parallel
X = X+1;
```
Synchronization

- 2 is the expected answer But can be 1 with unfortunate interleaving
- OpenMP assumes that the programmer knows what he is doing
- Regions of code that are marked to run in parallel are independent If access collisions are possible, it is the programmer's responsibility to insert protection

Synchronization

- Many of the existing mechanisms for shared programming
- OpenMP Synchronization
  - Nowait (turn synchronization off!)
  - Single/Master execution
  - Critical sections, Atomic updates
  - Ordered
  - Barriers
  - Flush (memory subsystem synchronization)
  - Reduction (special case)
Single/Master

- `#pragma omp single`
  - Only one of the threads will execute the following block of code
  - The rest will wait for it to complete
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region
  - Applicable to parallel for sections

Single/Master

- `#pragma omp master`
  - The following block will be executed by the master thread
  - No synchronization involved
  - Applicable only to parallel sections

```c
#pragma omp parallel
{
  do_preprocessing ();
  #pragma omp single
  read_input ();
  #pragma omp master
  notify_input_consumed ();
  do_processing ();
}
```
Critical Sections

- #pragma omp critical [name]
  - Standard critical section functionality
- Critical sections are global in the program
  - Can be used to protect a single resource in different functions
- Critical sections are identified by the name
  - All the unnamed critical sections are mutually exclusive throughout the program
  - All the critical sections having the same name are mutually exclusive between themselves

```c
int x=0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
  x++;
}
```
Ordered

- #pragma omp ordered statement
- Executes the statement in the sequential order of iterations
- Example:

```c
#pragma omp parallel for ordered
for (j=0; j<N; j++) {
    int result = j*j;
    #pragma omp ordered
    printf ("computation(%d) = %d\n" ,j ,
    result ) ;
}
```

Barrier synchronization

- #pragma omp barrier
- Performs a barrier synchronization between all the threads in a team at the given point.
- Example:

```c
#pragma omp parallel
{
    int result = heavy_computation_part1 ()
;
    #pragma omp atomic
    sum += result ;
    #pragma omp barrier
    heavy_computation_part2 (sum) ;
}
Explicit Locking

- Can be used to pass lock variables around (unlike critical sections!)
- Can be used to implement more involved synchronization constructs
- Functions:
  - `omp_init_lock()`, `omp_destroy_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`
  - The usual semantics
- Use `#pragma omp flush` to synchronize memory

Consistency Violation?

```c
#pragma omp parallel for \
shared(x) private(i)
for( i=0; i<100; i++ ) {
    #pragma omp atomic
    x++;
}
printf("%d",x); /* 100 */
```

```c
#pragma omp parallel for \
shared(x) private(i)
for( i=0; i<100; i++ ) {
    #pragma omp atomic
    omp_set_lock(my_lock);
    x++;
    omp_unset_lock(my_lock);
}
printf("%d",x); /* 96 !!! */
```
Consistency Violation?

```c
#pragma omp parallel for \
shared(x) private(i)
for( i=0; i<100; i++ ) {
    #pragma omp atomic
    x++;
}
printf("%i",x); /* 100 */
```

```c
#pragma omp parallel for \
shared(x) private(i)
for( i=0; i<100; i++ ) {
    omp_set_lock(my_lock):
    x++;
    #pragma omp flush
    omp_unset_lock(my_lock):
}
printf("%i",x); /* 100 */
```

Reduction

```c
for (j=0; j<N; j++) {
    sum =
    sum+a[j]*b[j];
}
```

- How to parallelize this code?
  - sum is not private, but accessing it atomically is too expensive
  - Have a private copy of sum in each thread, then add them up
  - Use the reduction clause!
  - `#pragma omp parallel for reduction(+: sum)`
  - An operator must be used: `+`, `-`, `*`...
Synchronization Overhead

- Lost time waiting for locks
  - Prefer to use structures that are as lock-free as possible!

```plaintext
#pragma omp parallel
{
  #pragma omp critical
  {
    ...
  }
  ...
}
```

Summary

- OpenMP is a compiler-based technique to create concurrent code from (mostly) serial code
- OpenMP can enable (easy) parallelization of loop-based code
  - Lightweight syntactic language extensions
- OpenMP performs comparably to manually-coded threading
  - Scalable
  - Portable
- Not a silver bullet for all applications
More Information

- www.openmp.org
  - OpenMP official site

- www.llnl.gov/computing/tutorials/openMP/
  - A handy OpenMP tutorial

- www.nersc.gov/nusers/help/tutorials/openmp/
  - Another OpenMP tutorial and reference

Backup Slides
Syntax, etc
• OpenMP Syntax

• General syntax for OpenMP directives

```c
#pragma omp directive [clause...] CR
```

• *Directive* specifies type of OpenMP operation
  • Parallelization
  • Synchronization
  • Etc.

• *Clauses* (optional) modify semantics of *Directive*

---

• OpenMP Syntax

• PARALLEL syntax

```c
#pragma omp parallel [clause...] CR
  structured_block
```

**Ex:**
```c
#pragma omp parallel
{
    printf("Hello!
");
} // implicit barrier
```

**Output:**
```
Hello! Hello! Hello! Hello!
```

(T=4)
OpenMP Syntax

- **DO/for Syntax (DO-Fortran, for-C)**

  ```c
  #pragma omp for [clause...] CR
  for_loop
  ```

  Ex:
  ```c
  #pragma omp parallel
  {
    #pragma omp for private(i) shared(x) \ 
    schedule(static,x/N)
    for(i=0;i<x;i++) printf("Hello!\n");
  } // implicit barrier
  Note: Must reside inside a parallel section
  ```

OpenMP Syntax

More on Clauses

- **private()** – A variable in private list is private to each thread
- **shared()** – Variables in shared list are visible to all threads
  - Implies no synchronization, or even consistency!
- **schedule()** – Determines how iterations will be divided among threads
  - **schedule(static, C)** – Each thread will be given C iterations
    - Usually T\*C = Number of total iterations
  - **schedule(dynamic)** – Each thread will be given additional iterations as-needed
    - Often less efficient than considered static allocation
- **nowait** – Removes implicit barrier from end of
OpenMP Syntax

- PARALLEL FOR (combines parallel and for)
  
  Ex:
  #pragma omp parallel for shared(x)\
   private(i)\
   schedule(dynamic)
  for(i=0;i<x;i++) {
      printf("Hello!\n");
  
Example: AddMatrix

Files:

(Makefile)
  addmatrix.c       // omp-parallelized
  matrixmain.c      // non-omp
  printmatrix.c     // non-omp
OpenMP Syntax

• ATOMIC syntax

    #pragma omp atomic CR
    simple_statement

Ex:
#pragma omp parallel shared(x)
{
    #pragma omp atomic
    x++;
} // implicit barrier

OpenMP Syntax

• CRITICAL syntax

    #pragma omp critical CR
    structured_block

Ex:
#pragma omp parallel shared(x)
{
    #pragma omp critical
    {
        // only one thread in here
    }
} // implicit barrier
OpenMP Syntax

**ATOMIC vs. CRITICAL**

- Use ATOMIC for “simple statements”
  - Can have lower overhead than CRITICAL if HW atomics are leveraged (implementation dep.)

- Use CRITICAL for larger expressions
  - May involve an unseen implicit lock

OpenMP Syntax

- MASTER – only Thread 0 executes a block
  ```
  #pragma omp master CR
  structured_block
  ```

- SINGLE – only one thread executes a block
  ```
  #pragma omp single CR
  structured_block
  ```

- No implied synchronization
• OpenMP Syntax

• BARRIER
  
  ```
  #pragma omp barrier CR
  ```

• Locks
  
  Locks are provided through `omp.h` library calls
  
  - `omp_init_lock()`
  - `omp_destroy_lock()`
  - `omp_test_lock()`
  - `omp_set_lock()`
  - `omp_unset_lock()`

• OpenMP Syntax

• FLUSH
  
  ```
  #pragma omp flush CR
  ```

• Guarantees that threads’ views of memory is consistent

• Why? Recall OpenMP directives…
  
  • Code is generated by directives at compile-time
    - Variables are not always declared as `volatile`
    - Using variables from registers instead of memory can seem like a consistency violation
  
  • Synch. Often has an implicit flush
    - ATOMIC, CRITICAL
OpenMP Syntax

Functions
omp_set_num_threads()
omp_get_num_threads()
omp_get_max_threads()
omp_get_num_procs()
omp_get_thread_num()
omp_set_dynamic()
omp_[init destroy test set unset]_lock()

Function for the environment

omp_set_dynamic(int)
omp_set_num_threads(int)
omp_get_num_threads()
omp_get_num_procs()
omp_get_thread_num()
omp_set_nested(int)
omp_in_parallel()
omp_get_wtime()