POSIX THREADS AND OPENMP
(SHARED MEMORY PARADIGM)

Presenters: James Baxter Thorne and Qun Li
Instructor: Prof. Ajay Kshemkalyani
Nov. 20 2012
OUTLINE

Part I (by Qun):
- Thread basics
- POSIX thread API

Part II (by James):
- OpenMP
POSIX Threads
COMMONLY USED MODELS FOR SHARED ADDRESS SPACE PROGRAMMING PARADIGMS

- Process based models (less suitable)
  - All memory is local unless explicitly specified or allocated as shared (protection vs. overheads)

- Light-weight process/thread models (preferred)
  - All memory is global and can be accessed by all the threads (much faster manipulation)
  - POSIX thread API/Pthreads: low level

- Directive based model (extension of the preferred)
  - Concurrency is specified in terms of high-level compiler directives (facilitating creation and synchronization of threads)
  - OpenMP has emerged as a standard
WHAT ARE THREADS?

- A *thread* is a single stream of control in the flow of a program.

Consider a program like:

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product( get_row(a, row),
                        get_col(b, col));
```

There are $n^2$ independent iterations for the for-loop. Such an independent sequence of instructions is a thread. They can be scheduled concurrently on multiple processors.

```c
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread( dot_product( get_row(a, row),
                                        get_col(b, col)));
```
WHY THREADS?

- Software portability
  - Developed on serial machines and run on parallel machines without changes.

- Latency Hiding
  - While one thread is waiting, other threads can use CPU

- Scheduling and load balancing
  - Rid the programmer of the burden of explicit scheduling and load balancing

- Ease of programming, widespread use
  - Easier to write than MPI, widespread acceptance
POSIX Thread (a.k.a Pthread) API

- IEEE specifies a standard 1003.1c-1995, POSIX API which has emerged as the standard threads API, supported by most vendors.
- Provides a good set of functions that allow for the creation, termination, and synchronization of threads. However, these functions are low-level and the API is missing some high-level constructs for efficient data sharing.
THREAD BASICS: CREATION AND TERMINATION

- Pthreads provide two basic functions for specifying concurrency in a program:

```c
#include <pthread.h>
int pthread_create (
    pthread_t *thread_handle, const
    pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg);

int pthread_join (
    pthread_t thread,
    void **ptr);
```

- The function `pthread_create` invokes function `thread_function` as a thread
- The function `pthread_join` waits for the termination of the thread.
**Thread Basics: Synchronization Primitives**

- When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.

- Consider:
  
  ```c
  /* each thread tries to update variable best_cost as follows */
  if (my_cost < best_cost)
    best_cost = my_cost;
  ```

- Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are 50 and 75 at threads t1 and t2.

- Two problems (race condition):
  - Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75!
  - The value 75 does not correspond to any serialization of the threads.
**Mutual Exclusion**

- The code in the previous example corresponds to a critical segment; i.e., a segment that must be executed by only one thread at any time.
- Critical segments in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. To access the shared data, a thread must first tries to acquire a mutex-lock.
  - Initialize mutex-locks to the unlocked state
  - Before entering a critical section, a thread tries to acquire a mutex-lock
    - If granted, it goes ahead
    - Otherwise, there is another thread currently in the critical section and no other threads will be allowed in
  - When a thread leaves a critical section, it must unlock the mutex-lock.
MUTUAL EXCLUSION

- We can now write our previously incorrect code segment as:
  ```c
  pthread_mutex_t minimum_value_lock;
  ...
  main() {
    ...
    pthread_mutex_init(&minimum_value_lock, NULL);
    ...
  }
  void *find_min(void *list_ptr) {
    ...
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
      minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
  }
  ```
OVERHEADS OF LOCKING

- Overheads of locking:
  - Locks represent serialization points since critical sections must be executed by threads one after the other, resulting in idling.
  - Encapsulating large segments of the program within locks can lead to significant performance degradation.

- It is often possible to reduce the idling overhead associated with locks using an alternate function, `pthread_mutex_trylock`.

```c
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex_lock);
```

- `pthread_mutex_trylock` attempts to lock on mutex-lock.
  - If granted, it returns 0
  - Otherwise, it returns EBUSY and allows the thread to do other work and poll the mutex for a lock.

- `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.
CONDITION VARIABLES FOR SYNCHRONIZATION

- `pthread_mutex_trylock` introduces the overhead of polling for availability of locks.
- A natural solution: condition variable.
  - It indicates the availability of space
    - If available, it signals the thread
    - Otherwise, it suspends the execution of the thread and the thread does not need to poll the lock.
By design, Pthreads provide support for a basic set of operations. Higher level constructs can be built using basic synchronization constructs

- Read-Write locks
  - In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
  - A read lock is granted when there are other threads that may already have read locks.
  - If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
  - If there are multiple threads requesting a write lock, they must perform a condition wait.

- Barriers
  - A barrier call is used to hold a thread until all other threads participating in the barrier have reached the barrier.
THREADS

OpenMP

- Open Multiprocessing
  - API for **shared memory** multiprocessing
    - Address space
    - Compiler extension
- Directive based
  - Preprocessor parallelizes code, inserts POSIX calls
- Support
  - C, C++, Fortran
  - Most common operating systems / architectures
**OpenMP - Management**

- OpenMP Architecture Review Board (OpenMP ARB)
  - nonprofit technology consortium

- Permanent Members:
  - AMD
  - Convey Computer
  - Fujitsu
  - IBM
  - Microsoft
  - NVIDIA
  - Oracle Corp.
  - CAPS-Enterprise
  - Cray
  - HP
  - Intel
  - NEC
  - Texas Instruments
  - The Portland Group, Inc.

- First released October 1997 for Fortran, C/C++ 1998
ABSTRACTION

- Handles details
  - Uses default values / environment variables
  - Mutexes
  - Condition variables
  - Data scope
  - Initialization for multiprocessing

- Mapping threads to cores
  - Handled by OS / runtime environment
  - Portability
  - Runtime changes
```c
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

The corresponding Pthreads translation is:

```c
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (........., internal_thread_fn_name, ....);
    for (i = 0; i < 8; i++)
        pthread_join (.........);
    // rest of serial segment
}
void *internal_thread_fn_name (void *packaged_argument) {
    int a;
    // parallel segment
}
```

**Figure 7.4** A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

Source: http://www-users.cs.umn.edu/~karypis/parbook/
MINIMAL WORK TO PARALLELIZE

- `#include <stdio.h>`

  ```c
  int main(void)
  {
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
  }
  ```

- `gcc -fopenmp hello.c -o hello`
BASIC DIRECTIVES / DATA HANDLING

- `#pragma omp parallel [clauses]`
  - `{ thread block }
- `num_threads (2)`
- Data privacy for variables
  - private (vars)
  - shared (vars)
  - firstprivate & lastprivate
    - initializes at start / remains in main thread at end
  - default (private)
  - reduction (op:vars)
    - `+, -, *, &, |, &&, ||`
    - Treated as local in threads, master gets reduction
EXAMPLE

#pragma omp parallel \
  default (private) \
  shared (a) \
  reduction (*: product) \
  num_threads (p * 2)
{
  /* code to be parallelized across threads*/
}

- Concurrency
- Synchronization
- Data handling

**Conditional Parallelization**

- `#pragma omp parallel if (expr)`
  
  ```
  { parallel block }
  ```

- Parallelizes if `expr == TRUE`
  - `expr` can contain runtime variables

- E.g.
  
  ```
  #pragma omp parallel if (n >= (2 * p)) num_threads (2)
  { parallel block }
  ```
CONCURRENT TASKS

- **for** directive
  - Split parallel iteration spaces amongst threads

```c
#pragma omp parallel for [omp clauses]
for(i=0; i<n; i++){
    /* code not in order across threads */
    #pragma omp ordered
    { forced in order execution, i = x before i = x + 1}
```

- Each thread gets a section of iterations
  - (i = 0-3, 4-7, etc)
CONCURRENT TASKS

- **section** directive
  - Specify a section for a thread

```c
#pragma omp parallel sections [omp clauses]
{
    #pragma omp section {
        runs in 1 thread }
    #pragma omp section {
        runs on a different thread }
}
```
SCHEDULING

- **schedule** clause of **for** directive
  - `#pragma omp for schedule (static [, chunk_size])`
  - **static**
    - iterations per thread set at start of loop
  - **dynamic**
    - chunk allocated to a thread when idle
  - **guided**
    - Reduce chunk size to evenly distribute to threads
  - **runtime**
    - Environment variable determines scheduling & chunk size, so can be set at runtime
EXAMPLE MATRIX MULTIPLICATION

```c
#pragma omp parallel default(private) shared (a, b, c, dim) \ 
  num_threads(4)
#pragma omp for schedule(static)
for (i = 0; i < dim; i++) {
  for (j = 0; j < dim; j++) {
    c(i,j) = 0;
    for (k = 0; k < dim; k++) {
      c(i,j) += a(i, k) * b(k, j);
    }
  }
}
```
Figure 7.5  Three different schedules using the static scheduling class of OpenMP.

a) 4 threads  b) schedule(static, 16)  c) schedule(static)
schedule(static)  On each for loop
Nested Parallelism enabled
setenv OMP_NESTED TRUE

Source: http://www-users.cs.umn.edu/~karypis/parbook/
SYNCHRONIZATION

- Check process name
- barrier directive
  - All threads in that parallel will wait
- single directive
  - Arbitrary single thread executes section
  - By default other threads wait at end of block
- master directive
  - just master executes, rest do not wait
SYNCHRONIZATION

- critical directive
  
  #pragma omp critical [(name)]
  
  Ensures only one thread executes at a time

- Can give names, so different threads can execute different code with protection

- flush directive
  
  Forces variable to be written to or read
# Example Producer / Consumer

```c
#pragma omp parallel sections {
    #pragma parallel section { /* producer thread */
        task = produce_task();
        #pragma omp critical (task_queue)
        {
            insert_into_queue(task);
        }
    #pragma parallel section { /* consumer thread */
        #pragma omp critical (task_queue)
        {
            task = extract_from_queue(task);
        }
        consume_task(task);
    }
} 
```
## OpenMP & Open MPI

<table>
<thead>
<tr>
<th>Implementation</th>
<th>OMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task level parallelism</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Data decomposition support</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Complex parallel patterns</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Broadly applicable generic parallel patterns</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Scalable nested parallelism support</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Built-in load balancing</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Affinity support</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Static scheduling</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Concurrent data structures</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Scalable memory allocator</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>I/O dominated tasks</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>User-level synchronization primitives</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Compiler support is not required</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cross OS support</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
**Matrix Multiplication**

- Sequential program – 30 lines of code
- OpenMP – 31 lines

```c
#pragma omp parallel for default(none) shared(a,b,c)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            for (int k = 0; k < size; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
```
Matrix Multiplication
Open MPI

- 75 lines of code

```c
int main(int argc, char* argv[])
{
    int rank, nproc;
    int istart, iend;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                a[i][j] = (float)i + j;
                b[i][j] = (float)i - j;
                c[i][j] = 0.0f;
            }
        }
    }

    MPI_Bcast(a, size*size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(b, size*size, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Bcast(c, size*size, MPI_FLOAT, 0, MPI_COMM_WORLD);

    istart = (size / nproc) * rank;
    iend = (size / nproc) * (rank + 1) - 1;

    multiply(istart, iend);
}```
MATRIX MULTIPLICATION
OPEN MPI (CONTINUED)

```
MPI_Gather(c + (size/nproc*rank),
           size*size/nproc,
           MPI_FLOAT,
           c + (size/nproc*rank),
           size*size/nproc,
           MPI_FLOAT,
           0,
           MPI_COMM_WORLD);

if (rank == 0) {
    if (size % nproc > 0) {
        multiply((size/nproc)*nproc, size - 1);
    }
}

MPI_Finalize();
return 0;
```
OPENMP

- Simple
- Automatic thread handling / data decomposition
- Portable
- Easy to have just certain sections parallelized
- No change to code to compile for serial

- Memory architecture limits scalability
- Can be difficult to debug
  - Race conditions / synchronization problems
POWER CONSUMPTION COMPARISON
OPENMP VS MPI

- www.green500.org
- Lists top500 by LINPACK FLOPS per watt
- Only two of top 10 from top500 on green top 10
  - Titan
    - top500:1, green500: 3
  - JUQUEEN BlueGene/Q
    - top500: 5, green500: 5
POWER CONSUMPTION COMPARISON
OPENMP VS MPI

- Can implementing OpenMP like features on each node in a cluster improve power efficiency?

- Algorithm and application greatly impact efficiency

Experiment:
  - Matrix multiplication
  - Shared memory, desktop PC
    - Shared Memory extension for MPI
    - Emulates shared memory behavior
### Power Consumption Comparison

#### OpenMP vs MPI

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP1</td>
<td>Single core Intel P4 CPU @ 2.4GHz, 1GB of DDR RAM @400MHz</td>
</tr>
<tr>
<td>PP2</td>
<td>Dual core AMD Athlon CPU @ 3.13GHz, 4GB of DDR2 RAM @ 1000MHz.</td>
</tr>
<tr>
<td>PP3</td>
<td>Single CRO-NGI ETFOS node (PowerEdge M600 Blade), two Quad-core Intel Xeon Processors E5430 @ 2.66 GHz, 16GB of ECC DDR2 RAM @ 667MHz</td>
</tr>
</tbody>
</table>
Figure 3. PP3 performance
Distributed Memory

- Efficient only with shared address space

- Distributed Memory System
  - Difficult to implement on large clusters
    - K computer – distributed memory cluster

- Local threads and Message Passing
  - Utilize benefits of threading
  - Open MPI support for OpenMP
COMBUSTION CHAMBER SIMULATION
MPI-OPENMP HYBRID

- CPU Cluster with 240 cores
- Two stage parallelization

- Four stroke engine
  - Moving piston
  - Intake / exhaust valves
COMBUSTION CHAMBER SIMULATION
MPI-OPENMP HYBRID

- OpenMP
  - limited by cores with access to virtual address space
- MPI
  - Excessive communication

- MPI used for inter-node
- OpenMP intra-node threads

- Considered using MPI for both levels
# Combustion Chamber Simulation

**MPI-OpenMP Hybrid**

## TABLE I. Hardware description

<table>
<thead>
<tr>
<th>System Name</th>
<th>Plexi</th>
<th>Memo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Cluster</td>
<td>SMP</td>
</tr>
<tr>
<td>Processor type</td>
<td>Intel Xeon X5650</td>
<td>Intel Xeon X7560</td>
</tr>
<tr>
<td>Cores / Processor</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Processor / Node</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td># of Nodes</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>Total Cores</td>
<td>240</td>
<td>64</td>
</tr>
<tr>
<td>Ram / Processor</td>
<td>24 GB</td>
<td>128 GB</td>
</tr>
<tr>
<td>Total RAM</td>
<td>960 GB</td>
<td>1024 GB</td>
</tr>
<tr>
<td>Interconnect type</td>
<td>Quad Infiniband</td>
<td>QPI</td>
</tr>
</tbody>
</table>
## COMBUSTION CHAMBER SIMULATION

**MPI-OpenMP Hybrid**

### TABLE II. Maximum Speedup and # of Cores

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>MPI</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>20.89</td>
<td>41.62</td>
<td>61.91</td>
</tr>
<tr>
<td># of Cores</td>
<td>32</td>
<td>76</td>
<td>192</td>
</tr>
<tr>
<td># of Nodes</td>
<td>1</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td># of Processes per node</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td># of Threads per process</td>
<td>32</td>
<td>1</td>
<td>12</td>
</tr>
</tbody>
</table>
COMBUSTION CHAMBER SIMULATION
MPI-OPENMP HYBRID
KMP STRING MATCHING
MPI & OpenMP

- Exclusive MPI implementation
- MPI & OpenMP (2 threads per node)
  - Uses twice the memory

- OpenMP style threading:
KMP STRING MATCHING
MPI & OPENMP HYBRID

- Hybrid threading

Figure 2 hybrid programming model
KMP STRING MATCHING
MPI & OPENMP HYBRID
THANK YOU FOR LISTENING, QUESTIONS?