Multithreading Programming

Introduction to High Performance Computing Systems (CS1645)

Esteban Meneses

Spring, 2014
Shared-memory Systems

- There is a single shared address space for all processors.
- All processors share the same view of memory.
Types of Shared-memory Systems

- **Uniform Memory Access (UMA)**: memory is equally accessible to all processors with the same performance (bandwidth and latency).

- **Cache-coherent Non Uniform Memory Access (ccNUMA)**: memory is physically distributed but appears as a single address space. Performance (bandwidth and latency) is different for local and remote memory access.

- Copies of the same cache line may reside in different caches. Cache coherence protocols guarantees consistency all time (for UMA and ccNUMA).

- Cache coherence protocols do not alleviate parallel programming for shared-memory architectures.

*Source: Prof. Dr. G. Wellein*
Thread

• “An independent stream of instructions that can be scheduled to run as such by the operating system”.

• Components:
  ❖ Stack pointer.
  ❖ Registers.
  ❖ Scheduling properties.
  ❖ Set of pending and blocked signals.
  ❖ Thread specific data.
Process vs Thread

**Process**

- User Address Space
  - **stack**
    - main()
    - routine1()
    - routine2()
  - **text**
    - routine1 var1
    - var2
  - **data**
    - arrayA
    - arrayB
- Stack Pointer
  - Prgm. Counter
  - Registers
  - Process ID
  - Group ID
  - User ID
- Files
  - Locks
  - Sockets

**Thread**

- User Address Space
  - **stack**
    - routine2() var1
    - var2
    - var3
  - Stack Pointer
    - Prgm. Counter
    - Registers
  - Process ID
    - User Group ID
  - Files
    - Locks
    - Sockets

Source: Blaise Barney, LLNL
POSIX Threads (PThreads)

- Historically, different vendors had different implementations of threads.
- Components of standard:
  - Types in C programming language.
  - Interface for function calls.
  - Header files.
  - Library with implementation.
## Performance

<table>
<thead>
<tr>
<th>Platform</th>
<th>fork()</th>
<th></th>
<th></th>
<th>pthread_create()</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
<td>real</td>
<td>user</td>
<td>sys</td>
</tr>
<tr>
<td>Intel 2.6 GHz Xeon E5-2670 (16 cores/node)</td>
<td>8.1</td>
<td>0.1</td>
<td>2.9</td>
<td>0.9</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>Intel 2.8 GHz Xeon 5660 (12 cores/node)</td>
<td>4.4</td>
<td>0.4</td>
<td>4.3</td>
<td>0.7</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>AMD 2.3 GHz Opteron (16 cores/node)</td>
<td>12.5</td>
<td>1.0</td>
<td>12.5</td>
<td>1.2</td>
<td>0.2</td>
<td>1.3</td>
</tr>
<tr>
<td>AMD 2.4 GHz Opteron (8 cores/node)</td>
<td>17.6</td>
<td>2.2</td>
<td>15.7</td>
<td>1.4</td>
<td>0.3</td>
<td>1.3</td>
</tr>
<tr>
<td>IBM 4.0 GHz POWER6 (8 cpus/node)</td>
<td>9.5</td>
<td>0.6</td>
<td>8.8</td>
<td>1.6</td>
<td>0.1</td>
<td>0.4</td>
</tr>
<tr>
<td>IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)</td>
<td>64.2</td>
<td>30.7</td>
<td>27.6</td>
<td>1.7</td>
<td>0.6</td>
<td>1.1</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4 (8 cpus/node)</td>
<td>104.5</td>
<td>48.6</td>
<td>47.2</td>
<td>2.1</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>INTEL 2.4 GHz Xeon (2 cpus/node)</td>
<td>54.9</td>
<td>1.5</td>
<td>20.8</td>
<td>1.6</td>
<td>0.7</td>
<td>0.9</td>
</tr>
<tr>
<td>INTEL 1.4 GHz Itanium2 (4 cpus/node)</td>
<td>54.5</td>
<td>1.1</td>
<td>22.2</td>
<td>2.0</td>
<td>1.2</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Source: Blaise Barney, LLNL
Parallel Program Design

• Considerations:
  ❖ Parallel programming model.
  ❖ Problem partition.
  ❖ Load balancing.

❖ Communication.
❖ Data and control dependencies.
❖ Synchronization points.
❖ Memory and I/O issues.

Source: Blaise Barney, LLNL
Shared-memory and Threads

Source: Blaise Barney, LLNL
PThread API

- `pthread_create(thread, attr, start_routine, arg)`
- `pthread_exit(status)`
- `pthread_cancel(thread)`
- `pthread_attr_init(attr)`
- `pthread_attr_destroy(attr)`
- `pthread_join(thread, status)`

Source: Blaise Barney, LLNL
PThreads Life Cycle

Main Program

```c
pthread_create(&thread,NULL,proc,&args);
```

Thread

```c
void proc(args){
...
return(status);
}
```

```c
pthread_join(thread,status);
```

- **Detached threads:**
  - Join is not needed.
  - OS destroys thread when it terminates.
  - A parameter in creation of thread indicates it is detached.
Locks

- Declare a lock: `pthread_mutex_t mutex;`
- Declare a mutex attribute: `pthread_mutexattr_t mta;`
- Initializing an attribute:
  - `pthread_mutexattr_init(&mta);`
  - `pthread_mutexattr_settype(&mta, PTHREAD_MUTEX_RECURSIVE);`
  - `pthread_mutexattr_setname_np(&mta, "My Mutex");`
- Initialize a mutex:
  - `pthread_mutex_init(&mutex, NULL); // use defaults`
  - `pthread_mutex_init(&mutex, &mta); // use attributes`
- Enter and release:
  - `pthread_mutex_lock(&mutex);`
  - `pthread_mutex_unlock(&mutex);`
- Try Lock without block: `pthread_mutex_trylock(&mutex);`
- Release resources:
  - `pthread_mutex_destroy(mutex);`
  - `pthread_mutexattr_destroy(&mta);`
Exercise

- Think about different ways to parallelize a matrix multiplication program with threads.

```c
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        for (k=0; k<L; k++)
            C[i][j] += A[i][k]*B[k][j];
```
PThread Hello World

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

void* hello(void* threadID) {
    long id = (long) threadID;
    printf("Hello World, this is thread %ld\n", id);
    return NULL;
}

int main(int argc, char argv[]) {
    long t;
    pthread_t thread_handles[NUM_THREADS];

    for(t=0 ; t<NUM_THREADS; t++)
        pthread_create(&thread_handles[t], NULL, hello, (void *) t);
    printf("Hello World, this is the main thread\n");
    for(t=0; t<NUM_THREADS; t++)
        pthread_join(thread_handles[t], NULL);
    return 0;
}
```

gcc -pthread hello.c
Example: Computing $\pi$

```c
int inTheCircle=0, numtrials=1000000;
double x, y;
for (i=0; i<numTrials; i++){
    x = rand();
    y = rand();
    if((x*x + y*y) < 1.0)
        inTheCircle++;
}
pi = 4*inTheCircle/numTrials;
```
Parallel Computation of $\pi$

```c
for(t=0; t<threads; t++)
    int localTrials = numTrials/threads;
    pthread_create(&threadHandles[t],
                   NULL, Thread_pi, (void *) &t,
                   (void *) &localTrials);

Pi = 4*inTheCircle/numTrials;
```

**Main Program**

```c
void* Thread_pi(void *rank, int numTrials){
    int myRank = (int) (*rank);
    double x, y;
    for (i=0; i<numTrials; i++){
        x = rand();
        y = rand();
        if((x*x + y*y) < 1.0)
            inTheCircle++;
    }
}
```

**Thread**
```c
void* Thread_pi(void *rank, int numTrials) {
    int myRank = (int) (*rank);
    double x, y;
    for(i=0; i<numTrials; i++) {
        x = rand();
        y = rand();
        if((x*x + y*y) < 1.0) {
            pthread_mutex_lock(&mutex)
inTheCircle++;
pthread_mutex_unlock(&mutex);
        }
    }
}
```
void* Thread_pi(void *rank, int numTrials) {
    int myRank = (int) (*rank);
default myinTheCircle = 0;
    double x, y;
    for (i = 0; i < numTrials; i++) {
        x = rand();
        y = rand();
        if ((x*x + y*y) < 1.0) {
            myinTheCircle++;
        }
    }
pthread_mutex_lock(&mutex);
inTheCircle += myinTheCircle;
pthread_mutex_unlock(&mutex);
}
Condition Variables

- A condition variable (cv) allows a thread to block itself until a specified condition becomes true.
- When a thread executes `pthread_cond_wait(cv)`, it is blocked until another thread executes `pthread_cond_signal(cv)` or `pthread_cond_broadcast(cv)`.
- `pthread_cond_signal()` is used to unblock one of the threads blocked waiting on the condition variable.
- `pthread_cond_broadcast()` is used to unblock all the threads blocked waiting on the condition variable.
- If no threads are waiting on the condition variable, then a `pthread_cond_signal()` or `pthread_cond_broadcast()` will have no effect.
Condition Variables for Synchronization

- A condition variable always has a mutex associated with it. A thread locks this mutex before issuing a wait, a signal or a broadcast.
- While the thread is waiting on a condition variable, the mutex is automatically unlocked, and when the thread is signaled, the mutex is automatically locked again.
- PThreads library provides the following functions for condition variables:
  - `int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);`
  - `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);`
  - `int pthread_cond_signal (pthread_cond_t *cond);`
  - `int pthread_cond_broadcast (pthread_cond_t *cond);`
  - `int pthread_cond_destroy (pthread_cond_t *cond);`
Barriers

- PThreads library provides support for a basic set of operations. Higher level constructs can be built using basic synchronization constructs.
- A barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex, and a condition variable:
  - A single integer (counter) is used to keep track of the number of threads that have reached the barrier.
  - If the count is less than the total number of threads, the threads execute a condition wait.
  - The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast and resets the count to zero (to prepare for the next barrier).
Barrier Type Definition

typedef struct{
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b){
    b->count = 0;
    pthread_mutex_init (&(b->count_lock), NULL);
    pthread_cond_init (&(b->ok_to_proceed), NULL);
}

void mylib_barrier(mylib_barrier_t *b, int thread_count){
    pthread_mutex_lock (&(b->count_lock));
    b->count++;
    if(b->count == thread_count) {
        b->count = 0;
        pthread_cond_broadcast(& (b->ok_to_proceed));
    } else
        pthread_cond_wait(& (b->ok_to_proceed), &(b->count_lock));
    pthread_mutex_unlock(& (b->count_lock));
}
Using Barriers in the Code

```c
mylib_barrier_t my_barrier; /* declare a barrier */
int main(int argc, char *argv){
    mylib_init_barrier(my_barrier); /* initialize the barrier */
    double A[5], B[5], C[5]; /* global, shared variables*/
    for(i=0;i<5;i++) A[i]=B[i]=1;
    for(i=0;i<4;i++) pthread_create(...,Proc,int i);
    for(i=0;i<4;i++) pthread_join(...,Proc,...);
    Print the values of C ;
}
void Proc(int threadID){
    int k;
    B[threadID+1] = 2 * A[threadID];
    mylib_barrier ( my_barrier, 4); /* call the barrier */
    C[threadID] = 2 * B[threadID] ;
}
```
Exercise

- Implement the parallel sum algorithm using PThreads.

```c
shared float v[n];
int value;

for(i=1; i<=log(n); i++)
  if(ID mod 2^i == 2^i-1)
    value = v[ID] + v[ID-2^i-1]
  barrier
  v[ID] = value
  barrier
```
void* sum(void* threadID){
    long id = (long) threadID;
    int i, value, stages = (int) log2(NUM_THREADS);
    for(i=1; i<=stages; i++){
        if(id % (int)pow(2,i) == (int)pow(2,i)-1){
            value = vector[id] + vector[id-(int)pow(2,i-1)];
        }
        mylib_barrier(&my_barrier,NUM_THREADS);
        vector[id] = value;
        mylib_barrier(&my_barrier,NUM_THREADS);
    }
    return NULL;
}
Summary

- Multithreading is an efficient mechanism to provide parallelism.
- Granularity in synchronization has an impact on performance.
- Locks and condition variables are the building blocks of synchronization operations.
- Floating-point arithmetic is not associative, in general.
Acknowledgements

- POSIX Thread Programming by Blaise Barney at LLNL.
- Lecture notes by Rami Melhem.