Tutorial 01 – POSIX Threads

ECE 459: Programming for Performance

Saeed Nejati
snejati@uwaterloo.ca

University of Waterloo

January 23, 2015

Quick Reference

- **API header:** `#include <pthread.h>`
- **Compiling:** `gcc -pthread source.c -o exec`
- **Main functions:**
  - **Creating threads:** `pthread_create`
  - **Waiting for another thread to finish:** `pthread_join`
  - **Exiting a thread:** `pthread_exit`
  - **Initializing a Mutex:** `pthread_mutex_init`
  - **Locking a Mutex:** `pthread_mutex_lock`
  - **Unlocking a Mutex:** `pthread_mutex_unlock`

```c
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
int pthread_join(pthread_t thread, void **retval);
void pthread_exit(void *retval);
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Section 1

An example use of Pthread

An example: Computing Sum of $\phi(n)$

- **Task:** Compute the sum of ϕ of numbers in a given range [l, h]
- ϕ is the euler's totient function
- We use a naive implementation for ϕ and don't care about optimizing the function itself

Sum of $\phi(n)$, version 1

- This is a simple serial implementation of the task.
- We considered the range to be [2, 30000]

```c
#include <stdio.h>

int gcd(int a, int b) {
    return a < b ? gcd(b, a) : b == 0 ? a : gcd(b, a % b);
}

int phi(int n) {
    int i, r = 0;
    for( i=1; i<n; i++ )
        if ( gcd(n, i) == 1 ) r++;
    return r;
}

int main() {
    int n;
    long long sum = 0;
    int low = 2;
    int high = 30000;

    for( n=low; n<=high; n++ )
        sum += phi(n);

    printf("%lld\n", sum);

    return 0;
}
```

Complete code can be found in *codes/serial.c*

Sum of $\phi(n)$, version 2

1. Calculation of $\phi(n)$ for each n could be done independent of each other (look into `main`)
2. Thus we can do it in parallel
3. We consider that we have a fixed number of threads (8).
4. Now we need to split the task and send it to threads

Sum of $\phi(n)$, version 2

- Let's split the interval into equal sized sub-intervals
- Thread routines accept one argument, so we represent a sub-interval as a pair structure: [first, last]

```
typedef struct { int first, last; } pair;

    pthread_t tid[THREAD_COUNT];                    /**< ID of worker threads */
    pair intervals[THREAD_COUNT];                   /**< Sub-intervals that are passed to
                                                         threads */
    int low = 2;                                    /**< Input Interval lower value */
    int high = 30000;                               /**< Input Interval higher value */
    int job_size = (high - low + 1) / THREAD_COUNT;  /**< Number of numbers in the sub-interval
                                                         for each thread */
    int remaining = (high - low + 1) % THREAD_COUNT; /**< Number of remaining jobs which should
                                                         be distributed over other sub-
                                                               intervals */

    int i, first = low, last;

    for( i=0; i<THREAD_COUNT; i++ ) {
        intervals[i].first = first;
        intervals[i].last = first + job_size - 1;

        if ( remaining > 0 ) {
            intervals[i].last++;
            remaining--;
        }

        first = intervals[i].last + 1;
    }
```

Sum of $\phi(n)$, version 2

- We should sum up the results and because all threads are going to update this sum variable, we need to protect it as well
- Consider these two variables are in a scope that are accessible by all threads

```
long long sum;
pthread_mutex_t sumLock;
```

- and before starting threads, we initialize these two like this:

```
sum = 0;
pthread_mutex_init(&sumLock, NULL);
```

- Now we can fire up our threads and wait for all of them to complete their tasks

```
for( i=0; i<THREAD_COUNT; i++ ) {
    pthread_create(&tid[i], NULL, threadRoutine, &intervals[i]);
}

for( i=0; i<THREAD_COUNT; i++ )
    pthread_join(tid[i], NULL);
```

Sum of $\phi(n)$, version 2

- At last, our `threadRoutine` would be something like this:

```
void* threadRoutine(void* arg)
{
    pair p = *((pair*)arg);
    int n;

    for( n=p.first; n<=p.last; n++ ) {
        pthread_mutex_lock(&sumLock);
        sum += phi(n);
        pthread_mutex_unlock(&sumLock);
    }

}
```

- **Note:** we cast the `void*` pointer to argument, to the appropriate type and then reference it to get the value of the job we need to work on

---

Complete code can be found in *codes/parallel1.c*

Quick Reminder

UNIX `time` command: `time [options] command [arguments...]`

- **real:** The elapsed real time between invocation and termination of process
- **user:** The user CPU time
- **sys:** The system CPU time (the time spent in system calls on behalf of process)

Execution Results

The results of running these two versions on a machine with Intel Core i7 CPU, 16M ram was:

- Version 1:
  ```
  real 0m38.191s
  user 0m38.180s
  sys 0m0.002s
  ```
- Version 2:
  ```
  real 0m38.214s
  user 0m38.708s
  sys 0m0.075s
  ```

But why the parallel version didn't make any improvements?

Sum of $\phi(n)$, version 2, look-back

- Take a look at this part of code (in `threadRoutine` function):

```
for( n=p.first; n<=p.last; n++ ) {
    pthread_mutex_lock(&sumLock);
    sum += phi(n);
    pthread_mutex_unlock(&sumLock);
}
```

- We are protecting the `sum` variable, which is a correct thing to do
- BUT, we are locking the main heavy computation part in the critical section
- No two threads go into Critical sections at the same time
- Therefore Critical sections are executed in a serial fashion
- Our parallel version is no better than a serial code + thread creation and mutex handling overhead! (look at the difference in `sys` time again)

Sum of $\phi(n)$, version 2.1

- **TIP:** Do as much as possible with your thread local data, then publish your results to global scope
- A fix to previous version could be:

```
void* threadRoutine(void* arg)
{
    pair p = *((pair*)arg);
    int n;

    long long local_sum = 0;
    for( n=p.first; n<=p.last; n++ ) {
        local_sum += phi(n);
    }

    pthread_mutex_lock(&sumLock);
    sum += local_sum;
    pthread_mutex_unlock(&sumLock);
}
```

The updated code can be found in *codes/parallel2.c*

C++11 Thread

- Creating and starting threads in C++11 is easier.
- You can have multiple arguments passed to the thread routine.
- Thread definition would be like:

```
std::thread threads[THREAD_COUNT];
```

- and mutex definition:

```
std::mutex sumLock;
```

- and the loop for starting threads:

```
for( int i=0; i<THREAD_COUNT; i++ ) {
    last = first + job_size - 1;

    if ( remaining > 0 ) {
        last++;
        remaining--;
    }

    threads[i] = std::thread(threadRoutine, first, last);

    first = last + 1; /* Move on to the next sub-interval */
}
```

C++11 Thread

- and finally the thread routine:

```
void threadRoutine(int first, int last) {
    long long local_sum = 0;
    for( int n=first; n<=last; n++ ) {
        local_sum += phi(n);
    }

    sumLock.lock();
    sum += local_sum;
    sumLock.unlock();
}
```

- and you can compile it like this:

```
g++ -std=c++11 -pthread parallel.cpp -o exec
```

Complete code can be found in *codes/parallel.cpp*

Execution Results, again

- Version 1:
```
real 0m38.191s
user 0m38.180s
sys 0m0.002s
```
- Version 2:
```
real 0m38.214s
user 0m38.708s
sys 0m0.075s
```
- Version 2.1:
```
real 0m9.975s
user 0m41.427s
sys 0m0.007s
```

- The results seems much better, but the current speed up is about 3.8X (instead of close to ideal 8X).
- The threads are not doing the same amount of job
- Computing $\phi(n)$ takes more time as n increases
- The 8th thread is doing more computation than the others
- It means that some threads are doing more than 1/8th of the total jobs
- How to balance the job splitting?
  - Dependent on task
  - Very hard to determine
- **An Option:** Break the jobs into smaller piece and when the threads finished their job, let them pick another one (known as Dynamic Scheduling, we won't discuss it here).

Some performance notes

- **Lock granularity:** How ”big” (coarse) or ”small” (fine) are your mutexes?
  - lock your whole structure or fields of a structure?
  - The more fine-grained, the more concurrency
  - but at the cost of more overhead and potential deadlocks
- **Lock ordering:** Make sure your locks are always locked in an agreed order
- **Lock frequency:** Are you locking too often? Reduce such occurrences to exploit concurrency and reduce sync overhead
- **Critical sections:** Take extra steps to minimize critical sections which can be potentially large bottlenecks

Section 2

Helgrind, A thread error detector

How to use it?

- It is part of Valgrind tool set
- You can invoke it like: `valgrind --tool=helgrind your_program [your_program_options]`
- It is a tool for detecting synchronization errors in C, C++ and fortran (which use pthread)
- It can detect:
  - Misuses of pthread API
  - Potential deadlocks arising from lock ordering problems
  - Data races – Accessing memory without adequate locking or synchronization

For more information and complete reference, please refer to: `http://valgrind.org/docs/manual/hg-manual.html`

A sample use

- Consider the following code, which has a data race on variable `var`

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
   var++; /* Unprotected relative to parent */ /* this is line 6 */
   return NULL;
}

int main ( void ) {
   pthread_t child;
   pthread_create(&child, NULL, child_fn, NULL);
   var++; /* Unprotected relative to child */ /* this is line 13 */
   pthread_join(child, NULL);
   return 0;
}
```

- If you compile the code with debugging info, helgrind would be able to address line of code with an error in it.
  `g++ -pthread -g race.cpp`

The code is an example from Helgrind reference document and can be found at *codes/race.cpp* too

A sample use

- Now, this is the result of running the code with Helgrind:
  `valgrind --tool=helgrind ./a.out`
- Helgrind reports two possible data races: one on read

```
Possible data race during read of size 4 at 0x601050 by thread #1
Locks held: none
   at 0x4007C1: main (race.cpp:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x400791: child_fn(void*) (race.cpp:6)
   by 0x4A0A245: ??? (in /usr/lib64/valgrind/vgpreload_helgrind-amd64-linux.so)
   by 0x3147407C64: start_thread (in /usr/lib64/libpthread-2.17.so)
   by 0x3146CF5B9C: clone (in /usr/lib64/libc-2.17.so)
```

A sample use

- and one on write
- It refers to line 13 and line 6 of the code

```
Possible data race during write of size 4 at 0x601050 by thread #1
Locks held: none
   at 0x4007CA: main (race.cpp:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x400791: child_fn(void*) (race.cpp:6)
   by 0x4A0A245: ??? (in /usr/lib64/valgrind/vgpreload_helgrind-amd64-linux.so)
   by 0x3147407C64: start_thread (in /usr/lib64/libpthread-2.17.so)
   by 0x3146CF5B9C: clone (in /usr/lib64/libc-2.17.so)
```