MPCS 51040 – C Programming

Lecture 10 – Optimization & Parallel Programming

Dries Kimpe

November 28, 2016

General Info

- HW6 grading in progress
- Quiz: see repository
- Project

Project

Project

Any questions/help needed?

Reminder:

- Review early
- Clarifications/questions: use piazza!

Profiling

What is profiling?

In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, ***profiling information serves to aid program optimization****.*
Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a profiler (or code profiler).
Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

What is the relationship to parallel programming?

- We want to direct our efforts so to have the most impact; In other words, we want to speed up the part of the code where most of the processing time is spent.
- Profiling is (one of) the methods to find those code areas.

Profiling

gprof

How does it work?

- Compiler instruments your code to collect information **at runtime** about which functions are being called and by whom.
  Note: causes (limited) slowdown and disturbs program.
- At certain intervals, **during the execution of your program** the current *program counter* is examined to determine which *function* is currently being executed.

Some caveats:

- We use gprof because it is readily available – it is not necessary the best tool for the job. . .
- gprof is not well suited for profiling multi-threaded programs.
- Compiler optimization can interfere (see later).

`https://sourceware.org/binutils/docs/gprof/`

gprof

```
today% gprof game gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 58.63      6.26     6.26 97951385     0.00     0.00  searchDir
 19.84      8.37     2.12 331558518     0.00     0.00  get
  9.66      9.40     1.03  8894885     0.00     0.00  board_has_winner
  4.88      9.92     0.52        9     0.06     1.17  rate_board
  1.69     10.10     0.18  8894866     0.00     0.00  board_unplay
  1.31     10.25     0.14  8894875     0.00     0.00  board_play
  1.13     10.37     0.12 17789741     0.00     0.00  set
  0.94     10.47     0.10 17620300     0.00     0.00  board_can_play_move
  0.89     10.56     0.10                             main
  0.61     10.63     0.07       10     0.01     0.01  board_destroy
  0.33     10.66     0.04       18     0.00     0.00  board_can_play
  0.09     10.67     0.01  8894886     0.00     0.00  board_get_width
  0.09     10.68     0.01  4405075     0.00     0.00  other_player
  0.00     10.68     0.00       13     0.00     0.00  player2string
  0.00     10.68     0.00       11     0.00     0.00  agent_describe
  0.00     10.68     0.00       11     0.00     0.00  board_get_height
  0.00     10.68     0.00       11     0.00     0.00  board_print
  0.00     10.68     0.00       11     0.00     0.00  computer_agent_describe
  0.00     10.68     0.00        9     0.00     1.17  agent_play
  0.00     10.68     0.00        9     0.00     0.00  board_duplicate
  0.00     10.68     0.00        9     0.00     1.17  computer_agent_play
  0.00     10.68     0.00        9     0.00     0.00  next_player
  0.00     10.68     0.00        3     0.00     0.00  readnum
  0.00     10.68     0.00        2     0.00     0.00  agent_destroy
  0.00     10.68     0.00        2     0.00     0.00  computer_agent_create
  0.00     10.68     0.00        2     0.00     0.00  computer_agent_destroy
  0.00     10.68     0.00        2     0.00     0.00  new_computer_player
  0.00     10.68     0.00        2     0.00     0.00  select_player
  0.00     10.68     0.00        1     0.00     0.00  board_create
  0.00     10.68     0.00        1     0.00     0.00  create_board
```

How to use it?

1. Compile and link with `-pg` (in addition to normal options)
2. Run the program as usual. Program must exit 'cleanly'.
3. At exit, file `gmon.out` is written in the **current directory**.
4. Analyze `gmon.out` using the `gprof` tool.
   `gprof <programexecutable> <gmonfile>`

- Compile/ link hw6
- Generate profile
- View using `gprof`
- Demonstrate on optimized binary

Analyzing GProf Output

Flat Profile

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 58.63      6.26     6.26 97951385     0.00     0.00  searchDir
 19.84      8.37     2.12 331558518     0.00     0.00  get
  9.66      9.40     1.03  8894885     0.00     0.00  board_has_winner
  4.88      9.92     0.52        9     0.06     1.17  rate_board
```

- Shows how much time was spent in each function (total), in descending order.
- Order by time *not inclusive* of time spent in children (i.e. other functions called by this function) – see *cumulative* for inclusive time.

https://sourceware.org/binutils/docs/gprof/Flat-Profile.html

Anaylzing GProf Output

Call Graph

Shows each function, which functions it called and which functions called it.

```
                                8894866              rate_board [4]
                 0.52    9.96       9/9              computer_agent_play [3]
[4]     98.2     0.52    9.96       9+8894866 rate_board [4]
                 1.03    8.32 8894875/8894885        board_has_winner [5]
                 0.18    0.12 8894866/8894866        board_unplay [8]
                 0.14    0.06 8894866/8894875        board_play [9]
                 0.10    0.00 17620300/17620300        board_can_play_move [11]
                 0.01    0.00 4405075/4405075        other_player [15]
                 0.01    0.00 8894875/8894886        board_get_width [14]
                              8894866                rate_board [4]
```

https://sourceware.org/binutils/docs/gprof/Call-Graph.html

C11 and Parallel Programming

C11 describes a header (`threads.h`) which provides functions for creating, destroying and synchronizing threads.

Thread support is *optional* in C11, and most compilers do not currently provide C11 thread functionality.

- This is mostly since there have been non-C11 standard specified methods of using threads. For example: POSIX threads (pthreads) described by POSIX.1c standard are available on most common platforms (including Windows, Linux, OS X, Solaris, OpenBSD)

We will be using pthreads (`pthreads.h`)

Review '`man pthreads`', in particular the section 'Thread-safe functions'.

POSIX Threads

Example program

```c
void * PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me,"
           "thread #%ld!\n", tid);
    pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0;t<NUM_THREADS;t++)
    {
      rc = pthread_create(&threads[t],
      NULL, PrintHello, (void *)t);
    }
    // need to join before exiting...
}
```

- Explicitly create and destroy threads.
- Synchronization through mutex, barrier, condition variables.
- Communication via Shared memory – threads can access each others variables.
- Low level interface.
- Supports both MPMD and SPMD.

Thread Creation/Destruction

```c
...
for (int i=0; i<count; ++i)
{
    pthread_create(&threads[i],
          NULL, printHello, (void *)(intptr_t) i);
}

// Join all threads
for (int i=0; i<count; ++i)
{
    // -- BLOCKING -- operation
    void * returncode;
    pthread_join(threads[i], &returncode);
}
...
```

- Program starts out with a single thread
- We manually (explicitly) create additional threads
  - `pthread_create` does not **block** or need to wait for the new thread to start executing – but might.
  - Creating a thread is not a cheap operation.
- Threads end when calling `pthread_exit` or returning from 'initial' function.
- *joining* a thread allows waiting for a a thread to exit (and obtaining a return code from the thread). The call **blocks** until the indicated thread has finished.

See `pt1.c` example program

- Compile and link (`-pthread`)
- Run with multiple thread counts and observe scheduling

Threads

Thread-safe functions (2)

```c
void * found;
bool find_node(node_t * node, void * d)
{
    if (!node)
        return false;
    if (node->data==d)
    {
        found=node;
        return true;
    }
    else if (find_node(node->left, d))
    {
        return true;
    }
    else
       return find_node(node->right, d));
}
```

```c
int pseudorandom()
{
    static_assert(sizeof(int)>=4,
      "proper size int");
#define M 2147483647
```

For a function to be thread-safe, at the very least, any access to a *shared* resource (memory or other) should be properly synchronized/protected.

Examples:

- Concurrent access to **the same** variable.
- Concurrent output to a file or to the screen.

Are these examples thread-safe? Why/Why not?

Thread-safety

Problem Example

```c
...
unsigned int total = 0;

void * printHello (void * threadid)
{
    ++total;
    // or pthread_exit(NULL);
    return NULL;
}
...
```

Question

- What is the problem here?
- How to fix it? (see next slide)

See pt2.c

Mutex

Limiting concurrent access

```c
...
unsigned int total = 0;

pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;

void * printHello (void * threadid)
{
    pthread_mutex_lock(&mutex);
    ++total;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
...
```

Mutexes **limit** concurrency!

(Remember Amdahl's law?)

A *mutex* allows us to have *mutual exclusion*

- Only one thread can **hold** (or lock) a mutex at a time.
- Other threads trying to lock the mutex will **block** (wait) until the mutex is unlocked.
- Only a single waiting thread will succeed in locking the mutex

There are many different kinds of mutexes; They differ on:

- Fairness
- Recursive locking
- Type (reader/writer)
- Use case (expected contention)

See `pt3.c`