Introduction to Parallel Programming: POSIX Threads Lab

Stéphane Zuckerman

23 January, 2018

Preliminaries

Building a PThread program.

```
cc $CFLAGS -o $PROGRAM source1.c source2.c -lpthread # etc.
```

... Where `$PROGRAM` is the desired name for the resulting application, and `$CFLAGS` contains the usual compilation flags. I suggest you use the following:

```
cc -Wall -Wextra -pedantic -g -std=c11 -o $PROGRAM source1.c source2.c -lpthread # etc.
```

Running a PThread program.

```
./$PROGRAM
```

Advice to Write PThread Programs.

- Always create a data structure that will contain all the data required by a given thread. Each thread must be passed a specific instance of that data structure in most cases.
- Usually, the `main` function of a PThread program tends to only perform the same basic steps:
  1. Process the program's inputs/parameters
  2. Allocate shared variables
  3. Create threads (declare `pthread_t` descriptors; declare arguments for each thread to be created; call `pthread_create`)
  4. Wait for threads to terminate (`pthread_join`, and possibly collect the return value of each thread that terminated)

  $\Rightarrow$ Avoid performing computations in this function

1 PThread Programming

1. Build a PThread program where each thread sends a "token" to another POSIX thread where its ID is one above. Each process must print its rank number before passing the token to the next process. If the process with the highest rank receives the token, it sends it back to process zero, and the program ends.

2. Implement a dot product (see explanations below) using a critical section (hint: use `pthread_mutext_t`, `pthread_mutext_lock`, `pthread_mutext_unlock`,

   and possibly `pthread_mutext_trylock`)

3. Implement a matrix-vector product (we provide the sequential code to compute it at the end of this handout). The `main` function is in charge of initializing the matrix and the vector to multiply.

4. Write a program which spawns N POSIX threads. The overall goal is to read from a large array of unsigned integers (initialized randomly in the `main` function), and have $N-1$ threads perform partial sums of sections of the array. Once their are done summing their associated segment of unsigned integers, each thread must then write the result to `partial_sums.txt`. The N^{th} thread waits until all threads are done writing to this file. **Implement this exercise using two variants: (1) using busy-waiting for the final thread, and (2) using condition variables**. Once it is done, the N^{th} thread reads all partial sums and computes the final sum. here are a few functions that may be of help (use `man` to get documentation on each function):

   - I/O functions: `fread`, `fwrite`, `fprintf`, `fgets`
   - Converting a string to an unsigned: `strtoul` (usually preceded by a call to `fgets`)
   - Signaling an event is fulfilled between POSIX threads: `pthread_cond_t`, `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

2 Stencil Codes

Stencil codes are used in many scientific computations, *e.g.*, to compute partial differential equations (such as heat diffusion on a plate using Laplacians, or computational fluid dynamics), or perform image processing (where one pixel is modified according to its neighbors). Usually the computation stops when the solution converges toward a good-enough precision, or it has reached a specific number of computation steps. We show the code for a 5-point 2D stencil below.

```
static inline void swap(void** p1, void** p2) {
    void* tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}

void
FivePointStencil(int tsteps, int n_rows, int n_cols,
                 double *p_old, double *p_new)
{
    double (*m_old)[n_cols] = (double (*)[n_cols]) p_old,
           (*m_new)[n_cols] = (double (*)[n_cols]) p_new;

    while (tsteps-- > 0) {
        for (int i = 1; i < n_rows-1; ++i)
            for (int j = 1; j < n_cols-1; ++j)
                m_new[i][j] = ( m_old[i-1][j] + m_old[i+1][j]
                              + m_old[i][j-1] + m_old[i][j+1] )
                            / 4;
        swap(m_old,m_new);
```

```
    }
}
```

Adapt the sequential code to make it work using POSIX threads.

Appendix

Dot Product

The dot product (also called the *scalar product*) takes two vectors $V1_n$ and $V2_n$, and returns the sum of the products of corresponding elements in $V1$ and $V2$:

$$s = V1 \cdot V2 = \sum_{i=1}^{n} v_{1_i} v_{2_i}$$

The corresponding code is given below.

```
double dotproduct(double *v1, double *v2, int n)
{
    double sum = 0.0;
    for (int i = 0; i < n; ++i)
        sum += v1[i]double * v2[i];
    return sum;
}
```

Matrix-Vector Product

A matrix-vector multiplication is defined as the product of a matrix $A_{M,N}$ with a column vector b_N. The result is stored in a row vector c_M: $c_M = A_{M,N} \times b_M$. The sequential code to compute such a product is given below. Each computed element of c is the result of a dot product between a row in A and the column vector b.

```
#include <string.h>
void matvec(int n_rows, int n_cols, int max_rows, int max_cols,
            double A[max_rows][max_cols],
            double b[max_cols], double c[max_rows])
{
    memset(c,0,sizeof(double)*n_rows);
    for (int i = 0; i < n_rows; ++i)
        for (int j = 0; j < n_cols; ++j)
            c[i] += A[i][j] * b[j];
}
```