Laboratory Exercise 2

—

Introduction to Pthreads Programming

Multiprocessor Systems, DV2544

Håkan Grahn
Blekinge Institute of Technology

April 11, 2016

The objective of this laboratory exercise is to study how applications can be parallelized using POSIX threads, and study some of those problems that may occur.

Home Assignment 1. Preparations

Read through this laboratory manual and do the **home assignments** in this manual.

End of home assignment 1.

1 Introduction

A computer system consists of one or several processors. Today, most processors are so called multicore processors, i.e., the processor contains several cpu cores internally. In this laboratory we will study computer systems with several processors, a so called multiprocessor system, how these can be programmed, and study some issues that affect the performance.

2 Practical details

The computer that we will use in the laboratory is called *kraken.tek.bth.se* and is a Linux server with 2 quad-core processors, i.e., it is in essence an 8 cpu machine, with 16GB primary memory. You will have a new user account on the machine, and the lab assistant will give you the password to your account. The source code files and programs that are necessary for the laboratory are copied or linked to your account by executing the command `/home/hgr/bin/init_mp`.

When measuring the execution time, you shall use `/usr/bin/time`. Further, if nothing else is said/written, you shall compile your programs using "`gcc -O2`". When you link a parallel program written using pthreads, you shall add "`-pthread`" to the compile command in order to link the pthreads library to your application.

3 Parallelization of a simple application

We start by parallelizing a simple application, *matrix multiplication*. A sequentiall version of the program is found in the file `matmul_seq.c`.

Home Assignment 2. Useful manual pages

Read the manual pages for the following functions (hint, e.g., write `man pthread_create` on a Linux system):

- `pthread_create`
- `pthread_exit`
- `pthread_join`
- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

End of home assignment 2.

Task 1. Log in

Log in on *kraken.tek.bth.se* using the username and password that have been provided to you. Use `ssh` (Secure Shell Client) to connect to *kraken*. Secure Shell Client is found under "Programs − > Internet" on the BTH PC-machines.
Copy relevant files to your account on *kraken* by executing the command `/home/hgr/bin/init_mp`.

End of task 1.

Task 2. Sequential matrix multiplication

Compile and execute the program `matmul_seq.c`.

How long time did it take to execute the program?

End of task 2.

Home Assignment 3. Parameter passing

Sending parameters to a new thread can be tricky. Read the man page for `pthread_create` and write the code for the situations below.

How does it look in the main program of `matmul` when you send a parameter `int a` to the function `init_matrix`?

How shall you recieve the parameter `a` in the function `init_matrix`?

End of home assignment 3.

We will now parallelize the matrix multiplication program using threads. Parallelize the program according to the following assumptions:

- Parallelize only the matrix multiplication, but not the initialization.
- A new thread shall be created for each row to be calculated in the matrix.
- The row number of the row that a thread shall calculate, shall be passed to the new thread as a paramater to the new thread.
- The main function shall wait until all threads have terminated before the program terminates.

Task 3. Parallel matrix multiplication

Compile and link your parallel version of the matrix multiplication program. Don't forget to add `-pthread` when you link your program. We will now measure how much faster the parallel program is as compared to the sequential one.
Execute the program on 8 processors and measure the execution time.

How long was the execution time of your parallel program?

Which speedup did you get (as compared to the execution time of the sequential version)?

End of task 3.

Task 4. Parallelization of the initialization

Parallelize the initialization of the matrices also, i.e., the function `init_matrix`. Use one thread to initialize each of the rows in the matrices `a` and `b`. Compile, link, and execute your program.

How fast did the program execute now?

Did the program run faster or slower now, and why?

End of task 4.

4 Changing granularity

In the previous tasks we have had a large number of threads (1024) to perform the matrix multiplication. An alternative would be to let each thread calculate several rows in the resulting matrix.

Task 5. Changing granularity

Rewrite your program so it only creates as many threads as there are processors to execute on, i.e., when we run the program on 8 processors, we create 8 threads that each calculate 1024/8 = 128 rows in the matrix. Execute your program and measure the execution time.

Which is the execution time and speedup for the application now?

Do these execution times differ from those in Task 3? If so, why? If not, why?

Does it make any difference now if `init_matrix` is parallelized or not?

End of task 5.

5 Effect of false sharing

In a multiprocessor with shared memory, the different threads communicate by reading and writing to shared variables. The main memory is divided into blocks and these blocks can be moved closer to the processor by copying them to the cache memory. In a multiprocessor can several processors have a local copy of the same memory block. In order to guarantee that the memory content is consistent when one processors updates its copy, all other copies are invalidated upon the write. The mecahnism that handles this is cacalled a cache coherence protocol.

The invalidation of cached memory blocks as a result of writes can have unexpected effects on the performance. In the file `false_sharing.c` you find a program with three global variables, `a`, `b`, `c`. These variables are updated a number of times of three concurrent threads, i.e., thread 1 (`inc_a`) updates variable `a`, thread 2 (`inc_b`) updates variable `b`, and thread 3 (`inc_c`) updates variable `c`.

Task 6.

NOTE: Compile the program with `gcc` only, i.e., **no** optimization at all! Optimization removes the interesting effects in this simple example.

How fast did the program run (execution time)?

End of task 6.

Task 7.

Rewrite the program so that the variable `a` is local in the function `inc_a`.

How fast did the program run this time?

End of task 7.

Task 8.

Rewrite the program so that the variable `b` is local in the function `inc_b`.

How fast did the program run this time?

End of task 8.

Task 9.

Finally, rewrite the program so that the variable `c` is local in the function `inc_c`.

How fast did the program run this time?

Which variables were probably allocated in the same memory block?

Did the execution time decrease when the last variable (`c`) became local? Why/why not?

End of task 9.

Appendix: Source code listings

```
/*****************************************************************************
 *
 * Sequential version of Matrix-Matrix multiplication
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
          /* Simple initialization, which enables us to easy check
           * the correct answer. Each element in c will have the same
           * value as SIZE after the matmul operation.
           */
          a[i][j] = 1.0;
          b[i][j] = 1.0;
        }
}

static void
matmul_seq()
{
    int i, j, k;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
          printf(" %7.2f", c[i][j]);
      printf("\n");
    }
}

int
main(int argc, char **argv)
{
    init_matrix();
    matmul_seq();
    //print_matrix();
}
```

```
/*****************************************************************************
 *
 * false_sharing.c
 *
 * A simple program to show the performance impact of false sharing
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define LOOPS (2000*1000000)

volatile static int a, b, c;

static void *
inc_a(void *arg)
{
    int i;
    //int a;
    printf("Create inc_a\n");
    while (i++ < LOOPS) a++;
    pthread_exit(0);
}

static void *
inc_b(void *arg)
{
    int i;
    //int b;
    printf("Create inc_b\n");
    while (i++ < LOOPS) b++;
    pthread_exit(0);
}

static void *
inc_c(void *arg)
{
    int i;
    //int c;
    printf("Create inc_c\n");
    while (i++ < LOOPS) c++;
    pthread_exit(0);
}

int
main(int argc, char **argv)
{
    int rc, t;
    pthread_t tid_a, tid_b, tid_c;

    rc = pthread_create(&tid_a, NULL, inc_a, (void *)t);
    rc = pthread_create(&tid_b, NULL, inc_b, (void *)t);
    rc = pthread_create(&tid_c, NULL, inc_c, (void *)t);
    pthread_join(tid_a, NULL);
    pthread_join(tid_b, NULL);
    pthread_join(tid_c, NULL);
}
```