Laboratory Exercise 3

—

Introduction to OpenMP Programming

Multiprocessor Systems, DV2415

Håkan Grahn
Blekinge Institute of Technology

February 19, 2009

The objective of this laboratory exercise is to study how applications can be parallelized using OpenMP, and do comparisons with the previous laboratory exercise on Pthreads.

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 a computer system with several processors, how it can be programmed using OpenMP, 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. This is the same machine as we used for the Pthreads laboratory exercise, so you shall use the same user account and password as before. 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_omp`.

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 -fopenmp`". The compiler flag "`-O2`" turns on code optimization in the compiler and the compiler flag "`-fopenmp`" enables the compiler to understand OpenMP directives.

Home Assignment 2. Basic OpenMP directives

Study how you shall use the following four OpenMP directives, i.e., how to use:

- `#pragma omp parallel [clause list]`
- `#pragma omp for [clause list]`
- `#pragma omp sections [clause list]`
- `#pragma omp section`

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 the menu "Start − > All Programs − > Internet" on the BTH PC-machines.
Copy relevant files to your account on *kraken* by executing the command `/home/hgr/bin/init_omp`.

End of task 1.

3 Parallelization of matrix multiplication

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

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.

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.
- Parallelize only the outermost `for`-loop using the directives `#pragma omp parallel for`.
- Think about which variables that need be declared as `private` and which that need be declared as `shared`.

Task 3. Parallel matrix multiplication

Compile and link your parallel version of the matrix multiplication program using the command "`gcc -O2 -fopenmp`".
We will now measure how much faster the parallel program is 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)?

How difficult was it to parallelize the matrix multiplication program using OpenMP as compared to when using Pthreads?

End of task 3.

Task 4. Parallelization of the initialization

Parallelize the initialization of the matrices also, i.e., the function `init_matrix`. 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.

Task 5. Nested parallelism

The next thing that we will study is how nested parallelism impacts the performance of the matrix multiplication application.
Turn on nested `for`-loop parallelism in the main program, i.e., to the directive `#pragma omp for`.

How fast did the program execute now?

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

End of task 5.

4 Changing granularity

In the previous tasks we let the compiler decide the number of parallel threads to be created and also the chunk size of each work package.

Your task now is to vary the number of threads, the schedule, and the chunk size of the `for`-loop and study the different execution times.

Task 6. Changing granularity

Rewrite your application so you can change the number of threads, schedule, and chunk size for the matrix multiplication program. You can do this by adding the clauses `num_threads` and `schedule(<scheduling_class>, <chunk>)` to your OpenMP directives `#pragma omp parallel for`.

What 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?

For which combination of number of threads, schedule, and chunk size did you get the shortest execution time?

End of task 6.

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();
}
```