Parallel Computing

Practicum / Home work

Week 2: OpenMP - part 1

This document describes the practical assignment for week 2 of Parallel Computing 2016-2017. Complete the exercises below and submit a report by answering questions related to the problems on Nestor. For this week it is sufficient to perform the computations on a local machine. You can do the exercises together with your partner, but both of you have to make the submission. All given code is available under the Assignments folder on Nestor. Please note that the system calls that print the user and timestamp in the end of each program assume that you work with a unix based OS, even though this can be adapted to others. The instructions below will form a base for your submission.

You can find the document that describes the OpenMP standard at
`http://openmp.org/wp/openmp-specifications/`.
We will use OpenMP version 3.0.

Exercise 1: Hello world!

Consider the following simple "Hello world" program in the file `hello.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <unistd.h>
int main (int argc, char **argv)
{
   #pragma omp parallel
   {
      printf ("Hello ");
      sleep(1);
      printf ("world!\n");
   }
   #pragma omp parallel
   printf ("Have a nice day!\n");
   printf ("Have fun!\n");

   printf("\n");
   system("echo $USER");
   system("date");
   return EXIT_SUCCESS;
}
```

The compiler that we will use for the practical exercises is called `gcc`. It is a freely available compiler that supports OpenMP. First we are going to compile the program without telling the compiler that this program is an OpenMP program. An OpenMP compliant C compiler will then ignore all `#pragma omp` directives. In other words, the program is considered a standard sequential C program. Compile the program with the command

```
gcc -Wall -mcmodel=medium -O3 -o hello hello.c
```

This will compile the file `hello.c` into the executable `hello`. Note that this is specified by the compiler flag `-o`. If you omit this flag, the default name is `a.out`. The flag `-Wall` specifies that we want the compiler to warn us about anything the compiler finds suspicious. We advise you to always turn on this flag. The flag `-O3` specifies that we want the compiler to optimize the code at so-called optimization-level 3. We advise you to use `-O3` by default. The option `-mcmodel=medium` tells the compiler that we want to use a memory model that supports more than 4 Gbytes memory. For this program this is clearly irrelevant, but we advise you to get accustomed to using this flag (it does not harm). Compile the program. You will see the following output

```
hello.c: In function main:
hello.c:6: warning: ignoring #pragma omp parallel
hello.c:12: warning: ignoring #pragma omp parallel
```

As you can see the compiler warns you that it ignores all `#pragma omp` directives. You will obtain the executable `hello`. Run the program. The output should be

```
Hello world!
Have a nice day!
Have fun!
```

Now, let us make the program parallel by telling the compiler that we want to use OpenMP. The flag `-fopenmp` is used for this purpose. We recompile as follows

```
gcc -Wall -mcmodel=medium -O3 -fopenmp -o hello hello.c
```

This time the compiler will use the `#pragma omp` directives and will create a parallel program. Next, we will tell the system how many threads we want to use by setting the environment variable `OMP_NUM_THREADS` which controls the number of threads to use[1]. For this exercise we choose 2. Note that the number of threads may exceed the number of available cores, although this is rarely effective.

```
export OMP_NUM_THREADS=2
```

Now run the program again. The output will be in the style of:

```
Hello Hello world!
world!
Have a nice day!
Have a nice day!
Have fun!

s123456
Wed Apr 19 15:30:47 CEST 2017
```

(a) In the output above some lines are interleaved/mingled and others not. Think about reasons why this is the case. Also save your own output, including the user name and time stamp which are automatically printed with the system() call. Answer questions 1-2 on Nestor.

[1]Note for non `bash`-shell users. In this document we assume that you are using the `bash` shell. If you are not using this shell, you should find out your selves how to set environment variables in the shell of your choice.

In the file `hello2.c` you will find the following variation of the program.

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

int main (int argc, char **argv)
{
  #pragma omp parallel
  {
    printf ("Hello world!\n");
    #pragma omp parallel
    printf ("Have a nice day!\n");
  }

  printf ("Have fun!\n");

  printf("\n");
  system("echo $USER");
  system("date");
  return EXIT_SUCCESS;
}
```

(b) Compile the program and run it. Save your output and answer questions 3-4 on Nestor.
(c) Now set the environment variable `OMP_NESTED` by typing

```
export OMP_NESTED=true
```

Run the program again and answer questions 5-6 on Nestor.

Exercise 2: OpenMP Memory Model

The goal of this exercise is to get acquainted with the properties of the *shared memory model* of OpenMP. Each thread can have *private* as well as *shared* variables. Private variables can only be accessed by the thread that *owns* them, while shared variables can be accessed by all threads (concurrently). Note that assigning a value to a shared variable is not atomic by default in OpenMP, so you have to take care of that yourself if atomicity is required.

Exercise 2.1 - Memory.c

Consider the following program to be named `memory.c`:

```
/* file: memory.c */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (int argc, char **argv)
{
  int i=31415;

  printf ("#i=%d\n", i);
  #pragma omp parallel private(i)
  {
    printf (" i=%d\n", i);
    i = 99;
  }
%The optimizer needs to be switched off for the block above.
%(If the optimizer is on the assignment i=99 is done at
%the beginning of the block).

  printf ("#i=%d\n", i);
  #pragma omp parallel firstprivate(i)
  {
    printf (" i=%d\n", i);
    i = 99;
  }
  printf ("#i=%d\n", i);
  #pragma omp parallel shared(i)
  {
    printf (" i=%d\n", i++);
  }
```

```
  printf ("#i=%d\n", i);

  printf("\n");
  system("echo $USER");
  system("date");
  return EXIT_SUCCESS;
}
```

(a) Study the source of the program and try to predict what the output of the program will look like if you run this program after setting `OMP_NUM_THREADS=4`. Next, compile the program, with the optimizer switched off, and check if your prediction was right. Save your output and answer questions 7-8 on Nestor.

Exercise 2.2 - Race.c

Now consider the following program to be named `race.c`:

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

#define N 1000000

int main (int argc, char **argv)
{
  int i, x;
  int histogram[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

  for (i = 0; i < N; i++)
  {
    x = 1;
    #pragma omp parallel sections shared(x)
    {
      #pragma omp section
      {
        int y = x;
        y++;
        x = y;
      }
      #pragma omp section
      {
        int y = x;
        y = 3*y;
```

```
        x = y;
      }
    }
    histogram[x]++;
  }

  for (i = 0; i < 10; i++) {
    printf ("%d: %d\n", i, histogram[i]);
  }

  printf("\n");
  system("echo $USER");
  system("date");
  return EXIT_SUCCESS;
}
```

The program simulates two concurrent processes that both try to modify the shared variable `x`. One thread tries to increment `x` while the other thread tries to multiply `x` with 3. This process is repeated 1000000 times. Compile the program. In this exercise we will turn off the optimization by using the compile command:

```
gcc -Wall -mcmodel=medium -fopenmp -o race race.c
```

(a) Set `OMP_NUM_THREADS` to 1 (i.e. run sequentially). Predict what the output of the program will be. Check your prediction by running it and answer question 9 on Nestor.
(b) Suppose that we run this program concurrently using two threads. Assume that a single assignment is atomic. What are all possible outcomes for `x`? (Hint: the total number of different outcomes is 4). Answer question 10 on Nestor.
(c) Now set `OMP_NUM_THREADS` to 2. Run the program and answer question 11 on Nestor. Next, set `OMP_NUM_THREADS` to 24 and run the program again. Don't be alarmed if the execution takes quite a while due to competition between the threads on the shared variable. Reason for yourselves about the result.

Exercise 2.3 - Shared.c

Now, consider the following program to be named `shared.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define N 100000000

static double timer(void)
{
  struct timeval tm;
  gettimeofday (&tm, NULL);
  return tm.tv_sec + tm.tv_usec/1000000.0;
}

int main (int argc, char **argv)
{
  double clock;
  int i;
  unsigned long long sum;
  clock = timer(); /* start timer */
  sum = 0;
  #pragma omp parallel reduction(+:sum)
  {
    #pragma omp for
    for (i=0; i<N; ++i)
    {
      sum = sum + i;
    }
  }
  clock = timer() - clock; /* stop timer */
  printf ("sum=%llu\n", sum);
  printf ("wallclock: %lf seconds\n", clock);
  printf("************************************\n");
  clock = timer();   /* start timer */
  sum=0;
  #pragma omp parallel shared(sum)
  {
    #pragma omp for
    for (i=0; i<N; ++i)
    {
      #pragma omp critical
      {
        sum = sum + i;
      }
    }
  }
  clock = timer() - clock;    /* stop timer */
  printf ("sum=%llu\n", sum);
  printf ("wallclock: %lf seconds\n", clock);

  printf("\n");
  system("echo $USER");
  system("date");
  return EXIT_SUCCESS;
}
```

(a) Compile and run the program. Run it with `OMP_NUM_THREADS` set to 1, 2, 4, and 8 respectively. Save your results and answer questions 12-14 on Nestor.