Summer School Parallel Programming
From Multi-core to Many-core and beyond

Day 2

Multi-Core Computing with OpenMP

Roy Bakker
R.Bakker@uva.nl

Roeland Douma
R.J.Douma@uva.nl

Clemens Grelck
C.Grelck@uva.nl

July 10, 2012

1 Introduction to OpenMP

1.1 Skeleton

A very simple skeleton for OpenMP code is the following (lets call it foo.c):

```
#include <omp.h>

int main()
{
    omp_set_num_threads(8);

    return 0;
}
```

We need to explicitly set the number of threads we want to use since we are executing on the DAS cluster. In order to compile your code run: `gcc -fopenmp foo.c -o foo`.

You can now run it on the DAS with the command: `prun -v -np 1 ./foo`.

For an overview of OpenMP you can view the slides and/or the OpenMP Specification[1].

1.2 Hello World

Of course your first program should be hello world. But as we are using OpenMP we want each thread to say hello. Write a simple program that starts a parallel section where each thread says hello and prints out his/her thread id. Also make sure that the master thread (and only the master thread) prints the total number of threads in the parallel section.

1.3 Hello World 2.0

Now lets make Hello World slightly less trivial. Extend your code so that each thread says hello and that it is thread m of n. But you can only request the number of active threads only once in the entire program.

1.4 Summing

Consider the code example in Fig. 1.

Exercise 1 Parallelize the second for-loop with the use of a critical section.

Exercise 2 Parallelize the second for-loop and use the reduction operator.

Exercise 3 Compare your two implementations. Can you explain the difference in performance?

[1] http://openmp.org/wp/openmp-specifications/

```
const int n = 1000;
int arr[n];
int i;
int result = 0;

for(i = 0; i < n; i++) {
    arr[i] = i;
}

for (i = 0; i < n; i++) {
    result += arr[i];
}
```

Figure 1: Simple reduction code in (sequential) C

2 Image filters

Also today we have the image filter framework available. Again available at `http://www.science.uva.nl/~bakkerr/summerschool/`.

However today you can play around with OpenMP. Add pragmas at several places and see what gives you the best performance. Explain why.

3 Mandelbrot

Fig. 2 shows a simple sequential C program that generates a Mandelbrot image, similar to the code used during the lecture.

You can compile this code with `gcc -fopenmp mandel.c -o mandel`. To run the code use: `prun -v -np 1 mandel > mandel.ppm`. To show your image run `display mandel.ppm`.

For your comfort there is also a "framework" available at: `http://www.science.uva.nl/~bakkerr/summerschool/`.

Exercise 1 Examine the code and try to parallelize it with OpenMP pragmas.

Exercise 2 Parallelizing the outer loop is often a good idea. However this might require to reorganize some of the code. Try to do this.

Exercise 3 Try to parallelize you reorganized code as much as possible and experiment with different setup. Which once gives you the best performance?

Exercise 4 Experiment with the different scheduling techniques that OpenMP provides. Can you measure performance differences? Which choice gives you the best runtimes?

Exercise 5 Add a dithering step to the Mandelbrot example analogous to the example in the lecture. Can you measure a difference in performance if you use the nowait clause?

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

void put_pixel(int red, int green, int blue)
{
    fputc((char)red,stdout);
    fputc((char)green,stdout);
    fputc((char)blue,stdout);
}

int main()
{
    double x,xx,y,cx,cy;
    const int itermax = 10000;    /* how many iterations to do */
    const double magnify=2.0;     /* no magnification          */
    const int hxres = 1000;       /* horizonal resolution      */
    const int hyres = 1000;       /* vertical resolution       */
    int iteration;

    /* header for PPM output */
    printf("P6\n");
    printf("%d %d\n255\n",hxres,hyres);

    for (int hy=1; hy<=hyres; hy++)  {
        for (int hx=1; hx<=hxres; hx++)  {
            cx = (((float)hx)/((float)hxres)-0.5)/magnify*3.0-0.7;
            cy = (((float)hy)/((float)hyres)-0.5)/magnify*3.0;
            x = 0.0;
            y = 0.0;
            for (iteration = 1; iteration < itermax; iteration++) {
                xx = x*x-y*y+cx;
                y = 2.0*x*y+cy;
                x = xx;
                if (x*x+y*y>100.0) {
                    iteration = 999999;
                }
            }
            if (iteration<99999) {
                put_pixel(0,255,255);
            } else {
                put_pixel(180,0,0);
            }
        }
    }
    return 0;
}
```

Figure 2: Simple Mandelbrot implementation in C