UvA / SURF-SARA
High Performance Computing and Big Data Course

Workshop 6: OpenMP and MPI

Assignments

Clemens Grelck

C.Grelck@uva.nl

June 25, 2014

UNIVERSITY OF AMSTERDAM

1 Getting started with OpenMP

1.1 Skeleton

A very simple skeleton for OpenMP code is the following (let us 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 it on a cluster with batch job submissio system. In order to compile your code run: `gcc -O3 -fopenmp foo.c -o foo`.

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 let us make *Hello World* slightly less trivial. Extend your code so that each thread says hello and that it is thread m of n. As a little challenge, you may request the number of active threads only once in the entire program.

2 Reductions

Consider the code example in Fig. 1.

```
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

Exercise 1 Parallelize the first for-loop.

Exercise 1 Parallelize the second for-loop using a critical section.

Exercise 3 Parallelize the second for-loop using the reduction clause.

Exercise 4 Compare your three implementations. Can you explain the performance difference?

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`. When running the resulting binary, redirect the standard output to a file, e.g. `mandel.ppm`. To show your image run `display mandel.ppm`.

For your comfort the code and a simple makefile are available at:
`http://www.science.uva.nl/~grelck/uva-sara-hpc-2013/`.

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 your reorganized code as much as possible and experiment with different setups. How do you obtain 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?

Exercise 6 Parallelize the middle loop instead of the outer, and play with the collapse clause. Can you measure a performance difference between the three different versions?

4 MPI collective communication

Collective communication is a form of structured communication, where instead of a dedicated sender and a dedicated receiver all MPI processes of a given communicator participate. A simple example is *broadcast* communication where a given MPI process sends a message to all other MPI processes in the communicator. Write a C+MPI function

```
int MYMPI_Bcast( void *buffer,           /* INOUT : buffer address  */
                 int count,              /* IN    : buffer size */
                 MPI_Datatype datatype,  /* IN    : datatype of entry  */
                 int root,               /* IN    : root process (sender)  */
                 MPI_Comm communicator)  /* IN    : communicator  */
```

that implements broadcast communication by means of point-to-point communication. Each MPI process of the given communicator is assumed to execute the broadcast function with the same message parameters and root process argument. The root process is supposed to send the content of its own buffer to all other processes of the communicator. Any non-root process uses the given buffer for receiving the corresponding data from the root process.

```
#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 iter, hy, hx;

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

    for (hy=1; hy<=hyres; hy++)  {
        for (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 (iter = 1; iter < itermax; iter++) {
                xx = x*x-y*y+cx;
                y = 2.0*x*y+cy;
                x = xx;
                if (x*x+y*y>100.0) {
                    iter = 999999;
                }
            }
            if (iter<99999) {
                put_pixel(0,255,255);
            } else {
                put_pixel(180,0,0);
            }
        }
    }
    return 0;
}
```

Figure 2: Simple Mandelbrot implementation in (sequential) C

Exercise 1 Implement the above function `MYMPI_Bcast` using point-to-point communication. Assume a fully-connected network topology, ie every node may send messages to any other node at the same price.

Exercise 2 A particular advantage of collective communication operations is that their implementation can take advantage of the underlying physical network topology without making an MPI-based application program specific to any such topology. For your implementation of broadcast assume a 1-dimensional ring topology, where each node only has communication links with its two direct neighbours with (circularly) increasing and decreasing MPI process ids. While any MPI process may, nonetheless, send messages to any other MPI process, messages may need to be routed through a number of intermediate nodes. Communication cost can be assumed to be linear in the number of nodes involved. Aim for an efficient implementation of your function on an (imaginary) ring network topology.

No experiments are required for this assignment, but describe your code in detail in a short report: how it is adapted to the given network topology and how many atomic communication events (sending a message from one node to a neighbouring node) are required to complete one broadcast.

5 Distributed Mandelbrot (optional)

Parallelise the Mandelbrot code of Fig. 2 to run on scalable architectures using MPI message passing.