Lab #10: The Mandelbrot Set with MPI

PCSE 2015

1 Mandelbrot set

NOTE: See Section 8.3 of the textbook for more information on this assignment.

If you've never heard the name Mandelbrot set, you probably recognize the picture. Its formal definition is as follows:

> A point c in the complex plane is part of the Mandelbrot set if the series x_n defined by
>
> $$\{x_0 = 0 x_{n+1} = x_n^2 + c$$
>
> satisfies
>
> $$\forall_n : |x_n| \leq 2.$$

It is easy to see that only points c in the bounding circle $|c| < 2$ qualify, but apart from that it's hard to say much without a lot more thinking. Or computing; and that's what we're going to do.

In this set of exercises you are going to take an example program `mandel_serial.cxx` and extend it to use a variety of MPI programming constructs. This program has been set up as a master-worker model: there is one master processor (for a change this is the last processor, rather than zero) which gives out work to, and accepts results from, the worker processors. It then takes the results and construct an image file from them.

1.1 Invocation

The `mandel_serial` program is called as

```
ibrun -np 123 mandel_serial steps 456 iters 789
```

where the `steps` parameter indicates how many steps in x, y direction there are in the image, and `iters` gives the maximum number of iterations in the `belong` test.

If you forget the parameter, you can call the program with

```
mandel_serial -h
```

and it will print out the usage information.

1.2 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates

```
class circle {
 public :
  circle(double stp,int bound);
  void next_coordinate(struct coordinate& xy);
  int is_valid_coordinate(struct coordinate xy);
  void invalid_coordinate(struct coordinate& xy);
```

and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound. It returns zero if the series from the given starting point has not diverged, or the iteration number in which it diverged if it did so.

```
int belongs(struct coordinate xy,int itbound) {
  double x=xy.x, y=xy.y; int it;
  for (it=0; it<itbound; it++) {
    double xx,yy;
    xx = x*x - y*y + xy.x;
    yy = 2*x*y + xy.y;
    x = xx; y = yy;
    if (x*x+y*y>4.) {
      return it;
    }
  }
  return 0;
}
```

In the former case, the point could be in the Mandelbrot set, and we colour it black, in the latter case we give it a colour depending on the iteration number.

```
  if (iteration==0)
    memset(colour,0,3*sizeof(float));
  else {
    float rfloat = ((float) iteration) / workcircle->infty;
    colour[0] = rfloat;
    colour[1] = max((float)0,(float)(1-2*rfloat));
    colour[2] = max((float)0,(float)(2*(rfloat-.5)));
  }
```

We use a fairly simple code for the worker processes: they execute a loop in which they wait for input, process it, return the result.

```
void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
  MPI_Status status; int ntids;
  MPI_Comm_size(comm,&ntids);
  int stop = 0;

  while (!stop) {
    struct coordinate xy;
```

```
    int res;

    MPI_Recv(&xy,2,MPI_DOUBLE,ntids-1,0, comm,&status);
    stop = !workcircle->is_valid_coordinate(xy);
    if (stop) res = 0;
    else {
      res = belongs(xy,workcircle->infty);
    }
    MPI_Send(&res,1,MPI_INT,ntids-1,0, comm);
  }
  return;
}
```

A very simple solution using blocking sends on the master is given:

```
class serialqueue : public queue {
private :
  int free_processor;
public :
  serialqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
    free_processor=0;
  };
  /* Send a coordinate to a free processor;
     if the coordinate is invalid, this should stop the process;
     otherwise add the result to the image.
  */
  void addtask(struct coordinate xy) {
    MPI_Status status; int contribution;

    MPI_Send(&xy,2,MPI_DOUBLE,
             free_processor,0,comm);
    MPI_Recv(&contribution,1,MPI_INT,
             free_processor,0,comm, &status);
    if (workcircle->is_valid_coordinate(xy)) {
      coordinate_to_image(xy,contribution);
      total_tasks++;
    }
    free_processor++;
    if (free_processor==ntids-1)
      // wrap around to the first again
      free_processor = 0;
  };
```

Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

1.3 Bulk task scheduling

The previous section showed a very inefficient solution, but that was mostly intended to set up the code base. If all tasks take about the same amount of time, you can give each process a task, and then wait on them all to finish. A first way to do this is with non-blocking sends.

Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with `MPI_Waitall` to finish before

you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.

You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```
class bulkqueue : public queue {
public :
  bulkqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
```

You will also have to override the `complete` method: when the circle object indicates that all coordinates have been generated, not all workers will be busy, so you need to supply the proper `MPI_Waitall` call.

1.4 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

Code a solution which uses scatter to distribute data to the worker tasks, and gather to collect the results. Is this solution more or less efficient than the previous?