SUMMER 2015 OOP 345 PROJECT

MAKING THINGS GO FAST

THREADING A COMPUTATIONAL INTENSIVE APPLICATION

Modern processors often have more than one core. A program utilizes the available compute power if it can effectively compute in parallel using all available cores.

Some jobs consist of performing multiple calculations were each individual calculation can be performed independently of the other calculations to be done. Each calculation does not depend on the results of other calculations. Each calculation is unique. This class of problems are called ***embarrasing parallel***. Calculations can be done in parallel using cores (or compute units) on the same or different computers. See https://en.wikipedia.org/wiki/Embarrassingly_parallel.

Mandelbrot fractal images are embarrasing parallel, trivial, but are computationally intense, to compute. They can take a long time to compute. See https://en.wikipedia.org/wiki/Mandelbrot_set.

Essentially one counts the number of times squaring a complex number and adding a constant, over and over again, is well behaved. Once the magnitude of the value is greater is greater than 2, the process diverges. The magnitude rapidly grows toward plus or minus infinity.

The count before the series diverges is the interesting property of a Mandelbrot image. Assigning a colour to the count produces an image.

You do not need to know how complex numbers work to calculate a Mandelbrot image.

Here is some non-complex number pseudo code from wikipedia, https://en.wikipedia.org/wiki/Mandelbrot_set, that calculates a Mandelbrot image:

```
for each pixel (Px, Py) on the the Nx x Ny pixel screen, do:
{
      x0 = xMin + (xMax-xMin) * Px/(Nx-1)
      y0 = yMin + (yMax-yMin) * Py/(Ny-1)
      x = 0.0
      y = 0.0
      iteration = 0
      max_iteration = 1000
      while ( x*x + y*y < 2*2  AND  iteration < max_iteration )
      {
            temp = x*x - y*y + x0
            ytemp = 2*x*y + y0
            x = xtemp
            y = ytemp
            iteration = iteration + 1
```

```
      }
      color = palette[iteration]
      plot(Px, Py, color)
  }
```

The **x*x + y*y < 2*2** expression is the same as calculting **sqrt(x * x + y * y) < 2** without making a **sqrt** call.

If you are aware of complex arithmetic, the same thing using the std::complex arithmentic class would be coded:

```
#include <complex>

for each pixel (Px, Py) on the the Nx x Ny pixel screen, do:
{
      std::complex<double> c0 (
            xMin+(xMax-xMin)*Px/(Nx-1),
            yMin+(yMax-yMin)*Py/(Ny-1));
      std::complex<double> p(0.0, 0.0);
      iteration = 0
      max_iteration = 1000
      while ( std::norm(p) < 2*2  AND  iteration < max_iteration )
      {
            p = p * p + c0;
            iteration = iteration + 1
      }
      color = palette[iteration]
      plot(Px, Py, color)
  }
```

The first phase in this project is to write a program that generates a Mandelbrot image.

The second phase is to speed it up using threads.

The third phase, noticing some threads finish early while other thread are still computing, is to use a thread pool to balance the work between the threads.

The fourth phase (optional) is to play around with your code and make it more interesting.

That's it for the summer 2015 OOP345 project.

PHASE ONE - CREATE A MANDELBROT PROGRAM

Write a program that accepts 8 command line paramters.

Parameter 1 is the output image file name.

Parameter 2 is the image width (number of columns) - an integer

Parameter 3 is the image height (number of rows) - an integer

Parameter 4 is the maxIteration count - an integer

Parameter 5 is xMin - a double

Parameter 6 is yMin - a double

Parameter 7 is xMax - a double

Parameter 8 is yMax - a double

Create a class **'Image'.** It manages 24-bit red-green-blue colour images and knows how to save the image data to a disk file. For colour images, 24-bit colour means each of the three red, green, and blue colours are 8-bit, for a total of 24-bits for the pixel. 'Pixel' is a contraction of 'picture element'. It is the colour of one dot in an image at a image coordinate (x,y). Coordinate x, lies between 0 and the number of columes in the image. The y-coordinate lies between 0 and the number of rows in the image.

An image consistes of a number of rows or lines. Each row or line contains pixels.

The constructor should accept the width and height of the image.

The class needs member functions which store and access image individual pixels and a function which writes the pixel data to an image disk file with a given filename.

We need to know how to save a 24-bit colour image to disk. There are many image file formats to choose from:

- JPEG – http://www.ijg.org/.
- TIFF – http://www.remotesensing.org/libtiff/.
- PNG – http://www.libpng.org/pub/png/.
- BMP - http://www.kalytta.com/bitmap.h, or drdobbs.com/architecture-and-design/184409517.
- PPM – http://netpbm.sourceforge.net/doc/ppm.html.
- and others

PPM is a minimalist colour graphics format. It is easy to write PPM files. Writing JPEG, TIFF, PNG, and BMP image files is more involved.

Pick a 24-bit colour image format. May I suggest PPM. It is an ASCII format. You can open the image file in a text editor and look at the pixels.

PPM file writers are simple:

1. **create the file.**
2. **write a 3 line header.**

   Line 1: -->P3\n<-- 3 characters: 'P' '3' '\n'

   line 2: -->width ' ' height \n<-- "width space height\n" in ascii

```
line 3: -->255\n<-- 4 characters '2' '5' '5' '\n'
```

3. **write the width*height pixels out in space-separated ascii values for red, green, and blue. The colour order is R1 G1 B1 R2 G2 B2 R3 G3 ...**
4. **close the file.**

See the PPM spec at http://netpbm.sourceforge.net/doc/ppm.html for more information

You are welcome to use any open source C++ library to handle writing image files. If you use a library, you need to learn how to use it. This can take a lot of time and be painful depending on the quality of the documentation for the library.

The next issue is to pick one. Googling "c++11 image format library" only returned 59 million hits.

A popular C++ library is Cimg, http://cimg.eu/reference/group__cimg__overview.html.

It is a single header file of templated classes.

DIY PPM code is so short, it is far easier than using a library. Using CImg is better since it supports multiple formats. Your choice. I would start with DIY PPM code.

Create a class '**Mandelbrot**' which manages Mandelbrot images.

Derive '**Mandelbrot**' from class '**Image**'.

The '**Mandelbrot**' constructor should accept the 5 Mandelbrot parameters: maxIteration, xMin, yMin, xmax, yMax. which specify a Mandelbrot image.

Create a member function **`'Calculate()'`** which calculates interation counts of the given size. **Mandelbrot** stores interation counts in a local table.

It has a member function which uses the std::transform algorithm to transforms count data into a image data to be stored in a instance of class Image.

For each pixel (row,column) in the image, calculate the number of iterations before it diverges.

Store the iteration count in a local table variable.

A simple way to generate a colour is to calculate:

```
unit32_t icolor = (double)iter / (double)max_iter * (1u << 24);
```

Other methods of generating colours from the iteration counts exist. See the wikipedia article.

Use a std::vector to store the image color data;

To covert **`icolor`** to a RGB colour, calculate

```
red   = icolor         & 0xff;
green = (icolor >>  8) & 0xff;
blue  = (icolor >> 16) & 0xff;
```

Similarily, unions can be used

```
      union uPixel {
            uint32_t pix32;
            struct {
                  uint8_t red;
                  uint8_t green;
                  uint8_t blue;
            } pixComponents;
      };

union uPixel pix;
pix.pix32 = icolor;
red   = pix.pixComponents.red;
green = pix.pixComponents.green;
blue  = pix.pixComponents.blue;
```

To measure how long things take, use the Timer class from the mid term test:

```
class Timer { // C++11 chrono needs "#include <chrono>".
  std::chrono::time_point<std::chrono::high_resolution_clock> Start;
  std::chrono::time_point<std::chrono::high_resolution_clock> stop;
public:
  void Start() { start  = std::chrono::high_resolution_clock::now(); }
  void Stop()  { stop   = std::chrono::high_resolution_clock::now(); }
  uint64_t nsecs() {
    typedef std::chrono::duration<int,std::nano> nanosecs_t;
    nanosecs_t duration_get
      ( std::chrono::duration_cast<nanosecs_t> (stop-start) ) ;
    uint64_t ns = duration_get.count();
    return ns;
  }
 };
```

Try running your program on the fast machines in room TEL2107.

There are many free image viewers. **Geeqie** is popular on Linux systems, **xnview** on Windows. With IOS you only need to click on the file and I understand IOS will display the image.

PHASE TWO – ACCELERATE IT WITH THREADING

Divide the work to be done by the number of cores on your computer. We will discuss how trivial it is to thread a for loop in class.

Write a new member function **CalculateThreaded** for class '**Mandelbrot**'.

Function **`std::thread::hardware_concurrency()`** returns the number of cores on your machine. It sometimes returns 0 if there is only one.

Spawn threads to do the work.

Measure the speed up due to threading.

Measure each thread start and stop time.

Print out the time spent in each thread. Pay attention to the finish time for threads. Calculate a utilization factor: time threads are busy / available time.

PHASE THREE – BALANCE THREADS WITH A THREAD POOL

You should observe some threads finish before other threads. The core is idle if the thread has finished. An idle core is not doing anything useful.

The goal in threading is to use all the cores all the time.

Break the job down into many small chunks. The chunks are placed in a queue. A thread takes a job off the job queue, runs it and then goes back to the queue to get the next job. When the queue is empty and all threads have completed, there is nothing left to do, and the job has finished.

That is what a thread pool is. Conceptually, a thread pool is simple. So is the the coding:

- Start the thread pool. Create N threads.
- Each thread starts in a sleeping state.
- Add jobs to the job queue.
- When a job is added to the queue, wake up one of the threads to handle the job.
- When a thread finishes processing a job, it tries to get another job from the job queue.
- If there are no more jobs to be queued, the thread pool manager will request threads to exit. If so, and the job queue is empty, the thread exits.
- When all threads have exited, the job is complete.

See https://en.wikipedia.org/wiki/Thread_pool_pattern.

A thread pool requires the use of mutexes and condition_variables. Both were discussed in class as well as in the scs.senecac.on.ca/~oop345 course notes.

A mutex is a lock method. It says “if the item to be locked is busy, wait your turn. If it not busy, mark it busy (lock it) and go ahead”. One needs to take care to remember to unlock a locked item.

We went over some attic mutex examples in class: lock/unlock/try_lock, and a smart mutex lock, which is similar to a smart pointer.

```
Basic mutex:
std::mutex> myMutex;
myMutex.lock();
...  what happens to the locked mutex if this code returns or throws?
...  what happens if this code calls a function that throws?
myMutex.unlock();

Same thing with a smart mutex:
```

```
{
      std::unique_lock<std::mutex> myLock(myMutex);
      ...  return or throw automatically unlocks the mutex
}
```

Smart mutexes and smart pointers (std::unique_ptr) make a lot of sense. Use them!

See http://www.cplusplus.com/reference/mutex/mutex/ and
http://www.cplusplus.com/reference/mutex/unique_lock/

CONDITION VARIABLES

A condition_variable says “wait (sleep) here until someone says to go”. More than one thread may be waiting on the condition_variable. One can wake up all threads waiting or only one thread. The operating system will decide which thread to wake up in the only one thread wakeup case.

Condition_variable use a mutex. Some operating systems occasionally wake up a thread before a wake up call. These events are called spurious wakeups. It is impotant to see that it is time to wake up. If not, it is a spurious wakeup, go back to sleep (wait).

There are four calls related to condition_variables we need to use:

1. create a condition_variable
```
std::condition_variable cv;
std::mutex> myMutex;
bool ready = false;  // if true, not a spurious wakeup event
```

2. wait here
```
{
      std::unique_lock<std::mutex> smartLock(myMutex);

      // if 'ready' false, spurious wakeup, need cv.wait to go
      // back to sleep
      auto f = [ready] () {return ready;};
      cv.wait(smartLock, f);  // 'f' returns true if authentic wakeup
}
```

3. wake up one thread
```
{
      std::unique_lock<std::mutex> smartLock(myMutex);
      ready = true;
      cv.notify_one();
}
```

4. wake up all threads
```
{
      std::unique_lock<std::mutex> smartLock(myMutex);
      ready = true;
      cv.notify_all();
}
```

That's it. That's all we need to know about condition_varibles.

THREAD POOL PSEUDO CODE

```
Class ThreadPool {
private:
      vector<thread>          threads;
      queue<function<void()> jobQueue;
      mutex                   mtx;
      condition_variable      cv;
      bool                    quit;
      bool                    stopped;
      void                    Run();  // note this is a private member function
public:
      ThreadPool(int numberOfThreads);
      ~ThreadPool();
      void addJob(function<void()> job);
      void ShutDown();
};

ThreadPool::ThreadPool(int numberOfThreads)
  : quit(???), stopped(???)
{
      for(int t=0; t < numberOfThreads; t++)
            threads.emplace( &ThreadPool::Run, this );

       // NOTE: The above line is tricky:
      // how to add a function pointer to a member function of a class
      // to a STL container, so I am giving you the exact line of code

      // Notice it needs to know about the member function as well as 'this'
      // which is the pointer to class data.
}

ThreadPool::~ThreadPool()
{
      if not stopped, call ShutDown
}

ThreadPool::ShutDown()
{
      {
            lock the thread pool
            set the 'quit' flag
      }

      wake up all threads

      join all threads  // wait for threads to exit

      set the stopped flag
}
```

```
ThreadPool::addJob(a 'void(void)' function pointer 'job')
{
      if 'stopped', throw “thread pool shutdown, not accepting jobs”
      if 'quit',    throw “thread pool shutting down, not accepting jobs”

      {
            lock the thread pool
            push 'job' onto the 'jobQueue'
      }
      wake up one thread
}

ThreadPool::Run() // note this is a private member function
{
      declare a job 'void(void)' function pointer 'job'

      while(true)
      {
            {
                  lock the thread pool

                  wait on the condition variable for a job to arrive
                        to prevent sperious wakeups, code a lambda
                        function which returns true if either
                              - jobs are available (use job.empty() )
                              - or the quit flag is set

                  if the quit flag is set and there aren't any jobs
                        return

                  get the job off the job queue
                        job = jobQueue.???
                        jobQueue.???

            }

            run 'job'
      }

}
```

Notice the thread pool code manages **`void(void)`** functions. If you need to pass a function with arguments to the thread pool, use **`std::bind`** to create a **`void(void)`** function pointer which can be passed to the thread pool.

Write a new member function **CalculateThreadPool** for class '**Mandelbrot**' which uses a thread pool.

The ThreadPool class is a useful general purpose thread pool. Code it as two files, **ThreadPool.h** and **ThreadPool.cpp.** Use it elsewhere during your C++ programming.

(OPTIONAL) PHASE FOUR – MAKE IT MORE FLEXIBLE

Make your Mandelbrot program more interesting:

- Add support for other image file formats.
- The Mandelbrot image calculates the interation count before the value of a variable becomes larger than 2*2=4 for the equation

  F(Z) = (Z * Z) + C,

  where

  Z = Zx + i*Zy, i = sqrt(-1)
  C is a complex constant C=Cx + i*Cy

  A Mandelbrot image is specific case of class of functions called a Julia Set.

  See https://en.wikipedia.org/wiki/Julia_set. Scroll down to the “Examples of Julia sets” to see images of other Julia set images for some other math functions.

  Try modifying your Mandelbrot code to generate some Julia Set images based on other math fuctions.
- Make your program display the image on the screen. I suggest using SDL as the leanest minimal overhead graphics display system.

INSPIRATION

Here is an inspirational link on high performance computing:

https://software.intel.com/en-us/articles/what-is-code-modernization.