AC 2008-1059: AN ARGUMENT AND EXAMPLE FOR THE EARLY INTRODUCTION OF THREADS

Dorian Yeager, Grove City College
An Argument and Example for the Early Introduction of Threads

Introduction

Teaching concurrency to undergraduates has always been a challenge, partly because actual experience with multiprocessor machines has not always been easy to arrange. We are now moving into a time when such machines will be the rule rather than the exception, but teaching concurrency is still difficult. Most of the principles used in designing for concurrent execution have traditionally been taught in the operating systems class, a class which is normally not encountered until fairly late in a student’s academic career. This paper argues that students need exposure to concurrency much earlier and that those topics should be covered in the core courses so that students obtain early, positive experience with threaded applications. A particular design for a set of assignments is presented, using Mandelbrot iteration because it is a compute-bound application for which students can see remarkable speedup and because the end result is an attractive, highly motivating piece of software that students can get excited about.

Motivation

Many courses taught in Computer Science (CS) and Computer Engineering (CE) on the undergraduate level must deal with issues of concurrency. Any course which requires students to produce graphical user interfaces would do well to place that topic in the context of threading, for the simple reason that some event responses initiate heavy computations. If the user interface becomes frozen because of a menu choice or some other user manipulation, the user’s satisfaction level plummets. If the computations so initiated are made to take place in a separate thread, then the interface itself remains responsive.

Another place where concurrency is extremely valuable is in a computer gaming course, where the graphics are animated and involve entities on the screen that appear to be acting independently and simultaneously. The most natural way to achieve this effect is with separate threads. However, if students have not mastered threading before they enter this type of course there will need to be a major effort devoted to their accommodating this topic on the front end of the course.

Much accommodation must be made by the student before he or she can be said to be proficient with threading. Ultimately the student must deal with some tricky and difficult issues to achieve proper synchronization and communication, but early in the study of the topic it is useful and appropriate to shield the student from such issues and give time for mastering the mechanics of dividing up a problem and consolidating the results produced by multiple threads.

Making room in the core of the CS or CE major for a solid introduction to threading will make it much more natural to discuss the more difficult issues involved as they occur in advanced
If the students can be given a heavily compute-bound application to attack with some success early on, and if they can see the benefits of the approach, they stand a much better chance of understanding and appreciating the need to master the more difficult aspects of the topic as their study progresses.

**Mandelbrot Iteration**

The application chosen for introducing this author’s students to threading was a graphical browser for examining the contours of a Mandelbrot iteration function $f$. The specific function studied maps the complex plane into the set of non-negative integers, and is therefore appropriately styled a “step function”. We define, for each complex number $c$, $f(c) = 0$ if $|c| \geq 3$; otherwise, if the iteration below terminates then the value $f(c)$ will be the terminal value of the integer variable $n$:

$$
n := 0
\quad z := 0
\quad \text{while } |z| < 3 \text{ do:}
\quad \quad z := z^2 + c
\quad \quad n := n + 1
\quad \text{end while}
$$

Finally, if the above iteration does not terminate before some prearranged number MAXIT of iterations, then $f(c)$ is given the value MAXIT.

The **Mandelbrot set** is the set of all complex numbers $c$ for which $f(c)$ will always be MAXIT no matter how large that constant is chosen to be. In other words, the Mandelbrot set is the set of complex numbers $c$ for which the iteration above never terminates. The Mandelbrot set has a number of interesting mathematical properties which make it fascinating to view on-screen. The addition to the display of color bands determined by the levels of the step function $f$ makes for some truly breathtaking views.

Ultimately the students were expected to (a) associate with each pixel in a window some complex number $c$, (b) compute $f(c)$ for each pixel, and (c) color the pixel in a specific fashion to indicate the value of $f(c)$. It is step (b) that makes this a heavily compute-bound problem, and it is the fact that all these computations are completely independent that makes this a highly partitionable problem.
**Stages of the Project**

The relatively large number of techniques and difficulties that present themselves during the process of building a Mandelbrot browser make that process a marvelous illustration of the power of object-oriented design. Careful division of labor into areas of responsibility governed by strategically designed classes make the problem much clearer and more tractable. There are four classes that suggested themselves to this author, and on which the first four stages of the project assigned to the students were based in turn:

(a) Class *ThreadedExecutionManager*. This class is nothing more than a list of thread functions and their operands. The list is empty at construction time, and an *add* method is available for adding a new thread function and its operand. There is also an *executeThreadsAndWaitForCompletion* method which makes a pass through the list, fashions a thread from each element of the list, begins each thread so that all threads execute in parallel, and then waits for all threads to terminate. A call to *getElapsedTime* returns the number of milliseconds used by the last execution of the list of threads.

(b) Class *MandelbrotIterationCounter*. This class is given, at construction time, a rectangular grid of points in the complex plane, each point corresponding to a complex number \( c \). The grid is specified in the constructor by (1) a lower-left-hand point \( z_0 = x_0 + iy_0 \), (2) a count of the number of \( x \) values in the grid, (3) a count of the number of \( y \) values in the grid, (4) a \( \Delta x \) value, and (5) a \( \Delta y \) value. A static thread function is associated with this class which receives as its only parameter a pointer to an object of the class. When the thread function is called, it uses a bounded modification of the iteration above on every point of the grid and in this way fashions a matrix (actually a vector of vectors) of integer counts and places it at class scope. In other words, for each valid pair of subscripts \((k, l)\), the count \( \text{count}_{kl} \) is \( f(c_{kl}) \), where \( f \) is the function defined above and \( c_{kl} = x_0 + k\Delta x + f(y_0 + l\Delta y) \). This matrix of counts can then be fetched using the method *getCounts*.

(c) Class *ThreadedMandelbrotIterationCounter*. An object of type *ThreadedMandelbrotIterationCounter* is instantiated in the same way as a *MandelbrotIterationCounter*, i.e. with exactly the same parameters. The difference is that once such an object has been instantiated it may (1) automatically subdivide the larger grid into smaller rectangular grids with the member function call *setNumberOfIntervals*, and (2) perform the iteration on all the grids in parallel and return a single vector of vectors for the entire grid. For example, if \( tmic \) is a *ThreadedMandelbrotIterationCounter* object then the call \( tmic.setNumberOfIntervals(2, 4) \) will split the rectangle into eight subrectangles by subdividing the \( x \) interval into two parts and the \( y \) interval into four parts. The subsequent call \( tmic.executeThreadsAndGetCounts() \) will instantiate eight
MandelbrotIterationCounter objects and will then use a ThreadedExecutionManager object to start all eight threads iterating, wait for them to finish, and finally construct and send back a vector of vectors with the counts for the larger grid. To demonstrate the effects of threading, the call `tmic.getTimeSpentIterating()` will return the elapsed time in seconds from the time the threads began to execute until the time they all finished.

(d) Class PixelColorComputer. This class will appear to the students at first to have nothing to do with the preceding classes. Ultimately it will be used to paint a picture of the Mandelbrot iteration function, but its description and the way it is tested give no strong indication of that. It incorporates a small array of base colors and a mapping of the non-negative integers into a set of colors defined by that array. This mapping is dependent on the base colors and on a positive integer step count \textit{step}. The colors are broken down in two ways: (i) into red, green, and blue components, and (ii) into \textit{step} intermediate colors equally spaced between any two consecutive base colors, using modular arithmetic for wrap-around. To compute an intermediate color requires three interpolations between base colors, one for each of the red, green, and blue components. The students were required to furnish a default array of specific basic colors but were also required to have the class randomly regenerate its base colors and step size. With explicit directions, this class is fairly easy to code and the students not only succeed but have quite a bit of fun with it.

The public interfaces of all of the classes above were specified by the author, and each student was required to furnish an implementation. The work proceeded in five different phases, and at the completion of each the students were given the option of continuing with their own source code or using the solutions posted by the author.

The first phase was not due until the fifth week of class and was preceded by lectures from Stroustrup’s C++ book\textsuperscript{3} and by supplementary lectures on using threads in a Microsoft Windows setting, as well as by problem-solving activities which exercised their skills at analysis, design, and coding. To complete Project 1 the students were given a very simple thread function and told to tabulate and graph the time consumed while doing the equivalent amount of work with one, two, four, five, eight, ten, and sixteen threads. Students who did not own a multiprocessor laptop were told to use a lab machine. Linear speedups were observed and noted by the students, and all were surprised to see that the overhead was light enough that they were not able to observe it by increasing the number of threads.

The second phase invited students to experiment with Mandelbrot iteration and threading, constructing multiple threads for evaluating the function on specific grids by crafting client code. They were given several small grids with which to test their classes, and they were given the correct arrays of counts with which to compare their results. They were then encouraged to try their classes on larger grids and note the amount of time consumed with different breakdowns of
a grid into subregions. That process of breaking down a grid was then given more structure by phase 3, in which the splitting of the rectangular grid into subrectangles is encapsulated by the `setNumberOfIntervals` method.

In phase 4, the students were told to implement and test the `PixelColorComputer` class by trying out different ways of computing non-negative integers corresponding to each pixel in a window and then using a `PixelColorComputer` object to furnish a color to paint that pixel. Essentially the students were generating very colorful contour maps of functions of two variables. The students found this a very entertaining way of experimenting with color, and generally had no idea that this project had anything to do with the other projects.

Finally, phase 5 brought together all the work in the previous phases. A single-document, single-view MFC$^2$ project was fashioned in which the client area of the window was painted in a fashion similar to that of phase 4, using a `PixelColorComputer` object to furnish the colors but using the Mandelbrot function $f$ (as implemented in `ThreadedMandelbrotIterationCounter`) described above to associate an integer with each pixel. The completed work had to (a) correctly display the function on a specified domain using colors chosen by the student (but always using black for the Mandelbrot set itself), (b) react to a mouse click by generating a new display with the point corresponding to the mouse click at the center and with a reduced domain, causing a “zoom” effect, (c) display on the status bar the number of seconds consumed by the computation, and (d) provide menu choices that increased or decreased the number of threads used and the value of MAXIT. This project allowed the students to examine in detail the Mandelbrot set and the interesting plateaus surrounding it as defined by the function $f$. The students were dazzled by the effect produced, and many have commented that it was their most compelling project ever.

The students were invited to embellish their work in phase 5. They were shown more sophisticated ways to manage the display, and many incorporated those techniques. It was pointed out to them that replacing the single view object with a list of views and traveling back and forth in the list required only routine modification, and the program so modified allowed previously generated views of the Mandelbrot set to be recalled using menu choices or accelerator keys, so that the effect was very much like that of a browser. Several students implemented this suggestion also.

**Tools**

The project was assigned to students proficient in the use of Microsoft Visual Studio 2005 and the C++ language, and those were the tools they used to complete it. A Java or C# environment is certainly useable, but may be a bit slow. Native code and operating system-defined threads will give students a better feel for the potential benefits of threading.
The Win32 API was used for the creation and maintenance of threads, and the Microsoft Foundation Classes (MFC) library was used for the graphical aspects of the project, partly because it is object-oriented but mostly because it took less time to develop the essential skills needed, since MFC provides object-oriented “wrappers” that hide much of the inessential details of the Win32 interface.

Conclusions

In previous offerings of this course a text editor project had been used, and that project had some of the same benefits as the Mandelbrot project in terms of illustrating the power of object-oriented design. However, it did not capture the imagination of the students and did not incorporate threading or graphics. The addition of these two components rejuvenated the project aspect of the course and made the course much more enjoyable for the students, as well as teaching them a broader set of skills. Moreover, the students were shown the advantages of threading quite graphically when they observed the difference in the time it took to compute a new display with and without threading.

All source code and project descriptions are freely available from the author at Grove City College, 100 Campus Drive, Grove City, Pennsylvania 16127, email dpyeager@gcc.edu.

Bibliography