**Synthesis Questions**

1. Evaluate your performance in terms of speed-up and cost. Chart your results. Be sure to run the programs with different numbers of processors and different coordinate ranges.

I ran the program with 1080 rows, and 1920 columns, with x ranging between -1.5 and -1, and y ranging between -1 and -0.5. I took 5 measurements for each permutation. While there was speedup with the static method, the speedup was greater in the dynamic method, so long as more than two processors were used. This is expected, because the dynamic method on two processors really does the algorithm on one processor, with the added overhead of communication between the worker and manager. The dynamic allocation method used was very efficient up to 16 processors.

1. Is the Mandelbrot set application a good choice for parallelization? Why or why not?

The Mandelbrot set is a great choice for parallelization. The exact same calculation is run multiple times on numerous elements.

1. Is the speed up what you expected? Why or why not? What are some possibilities to improving the speed up?

The static method doesn’t have a whole lot of room for improvement. One thing I did find in the nested loops is that cy is unnecessarily assigned in the inner loop. cy can be assigned once for each outer loop, with cx changing in the inner loop. That saves a lot of floating-point multiplications and divisions. The biggest thing slowing it down, though, is that several processors end up being idle for most of the time while one or two end up doing the most work.

1. Is there any advantage to the dynamic allocation over static? Justify your answer.

Dynamic allocation allows for load balancing, which makes the program take less time overall. There is less wasted processor time, making the program more efficient.

5.      Did you use collective communication methods in your implementation? Why or why not?

I used the gather method in my static implementation. This is because gathering up the data from all of the processors is the last step, once it has all been calculated out. The dynamic approach I took does not require this, because the manager retrieves each row as it is completed. This adds communication overhead, but allows for an overall faster process.

1. Pseudo code you implementation. Could you reduce the message passing overhead? Is there a better way to partition the work among the processors?

FOR each row assigned to me

interpolate to find cy

FOR each col in the row

interpolate to find cx

find color using Mandelbrot function

put color into picture[row][col]

Gather the results to master node

There isn’t really much message passing overhead in this implementation. The only time the processors talk is to gather the results. That being said, there are better ways to partition the work, which I did with my dynamic approach.

7.      What portion of processing is sequential? Does the sequential portion scale with data set size? Which law (Amdahl’s or Gustafson’s) is more appropriate in this case?

The only portion of this program that is sequential is writing the results to file, which does scale with data set size. The total workload does not scale with the number of processors, so Amdahl’s law applies here.

8.      Suppose we wanted to calculate PI to thousands of digits of accuracy. Unfortunately doubles only support sixteen digits of precision. How could we overcome this problem?

Use a digit extraction method, and assign digits to an array.

9.      If we use parallel program to manipulate an image being displayed. For example, suppose we want to rotate an image in three dimensions. Do you see a bottleneck that could arise?

Not so sure. If the manipulation were statically spread across processors, then some processors may have to do more work than others.

10.  For the following row-order matrix, what is single dimension offset of the 11, 88, and 53? What formula did you use to calculate the offsets?  
  
int A[3,2,5] = { { {14,3,8,2,1}, {22,17,18,14,95} },

{ {99, 47,33,22, 11}, {16, 88, 77, 66, 55} },

{ {44,45,46,47, 48}, {55,54,53,52,51} } }

11: A[1,0,4]: 1\*(2)\*(5) + 0\*(5) + 4 = 14: A[14]

88: A[1,1,1]: 1\*(2)\*(5) + 1\*(5) + 1 = 16: A[16]

53: A[2,1,2]: 2\*(2)\*(5) + 1\*(5) + 2 = 27: A[27]