Short-Answer Questions

1. Consider the number of processors used for a parallel computing application. Explain in detail why it is not always useful to just have as many processors as possible.

2. What is Moore's Law? How does it relate to the development of multicore computers today?

3. Explain how a cluster would typically use both shared memory and distributed memory paradigms. Explain how communication between processes would most likely occur in such a cluster.
4. Many image processing algorithms are parallelizable. Consider the quantization code we wrote, which reduces the number of bits used to represent each channel of color. Describe in detail how you might parallelize this image operation in Python (as in, what parallel processes you would instantiate, what parameters those processes would receive, etc.).
5. Describe the use of Process/Pipe versus Pool/map in Python. How did we use the two paradigms for the algorithms we discussed (in detail)?

6. Consider the code below:

```
from multiprocessing import *

def child(q):
    s = 0
    for i in range(100):
        s = s + random.randint(1, 10)
    q.put(s)

def main():
    q = Queue()

    p1 = Process(target=child, args=(q,), name="p1")
    p2 = Process(target=child, args=(q,), name="p2")

    p1.start()
    p2.start()

    a1 = q.get()
    a2 = q.get()

    print "p1 sum:", a1
print "p2 sum:", a2
```

The intent is that two processes would each add up 100 random numbers, with the sum for each process printed and labeled as shown. There is something wrong with the code in main(), however. Describe how you would change main() to fix the problem. (Do not change child().)

Programming Problems

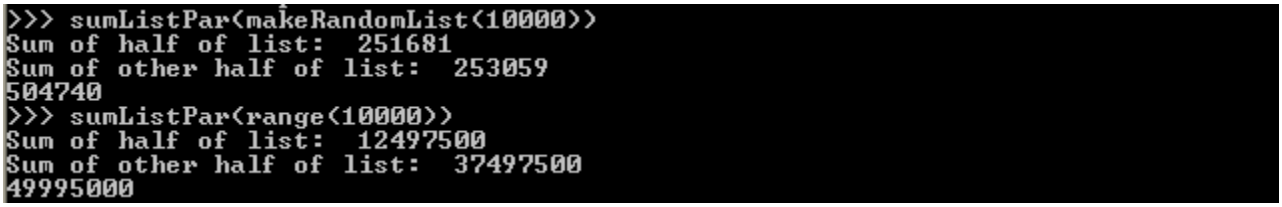
1. Write parallelMergesort.py: There are various ways of approaching a parallel implementation of the mergesort algorithm. A complicated implementation could involve a parallelization of the

merge operation, where two Processes responsible for separate sorted lists swap elements with each other to ensure that the lists don't overlap. Then the lists could simply be concatenated to form the overall sorted list.
You can take a more simple approach, duplicating the kind of parallelism used in the parallelQuicksort.py example. Here, we keep the merge operation as before and rely on the non-parallel mergesort, but simply run more than one instance of mergesort at a time.

2. Write a function sumListPar() that takes a list of numbers and returns the sum. It should follow the following pseudocode:

   def sumListPar(ls):
    start a child process to sum the first half of the list
    start a child process to sum the second half of the list
    print the sum of each half
    return the total sum

   For example:



   Note in the first example that I'm using:

```
import random
def makeRandomList(numNumbers):
    ls = []
    for i in range(numNumbers):
        ls.append(random.randint(1,100))
    return ls
```

   So of course your answers will vary for the 1st example, since it's random. But the second example should match for everyone, except that the order of the halves might be different on different runs of the function.

   Notes:
   - Pass the entire list to each child, along with an indication of the indices it should be adding for the given list.

3. Write a function named restaurant() that will be a "simulation" of a small restaurant. There will be three processes: the main process, a "chef" process, and a "server" process. The main process spawns the chef and server processes. The main process gets a number of meal orders from the user, and passes the list of orders to the chef. The chef loves to take quick naps, so he does the following:

   for each meal:

nap for 1 to 3 seconds
prepare the current meal and send it to the server

For our purposes, the "sent meal" is just a string describing the meal. The server, in the meantime, is waiting to receive one meal (string) after another from the chef, each of which he then "delivers" to the customer (prints a message about the meal).

Here's an example:

```
>>> restaurant()
Enter order (<Enter> when done): Chicken fingers
Enter next order (<Enter> when done): Double cheeseburger
Enter next order (<Enter> when done): Medium fries
Enter next order (<Enter> when done): Fish sandwich
Enter next order (<Enter> when done):
>>> Here's your Chicken fingers.  Enjoy!
Here's your Double cheeseburger.  Enjoy!
Here's your Medium fries.  Enjoy!
Here's your Fish sandwich.  Enjoy!

>>>
```

So the user first types in the meals. The list of meals is sent to the chef. The chef sends one at a time to the server, symbolizing the meal being "ready".

Note that only the chef receives a list of the meals from the main process. The server **never** receives such a list!

Remember, the chef sleeps a bit between cooking each meal. So there will be a pause between each "Here's your <meal>. Enjoy!" message, because the server is waiting for the next meal to be ready. The server **never** sleeps, though. (Poor guy!)

You might imagine that we could make this more sophisticated in a computational modeling kind of way, with various servers and chefs and rates of incoming orders, to determine what kind of response time to expect in a given restaurant. We'll just keep things simple for now, though.

Hint: In real life, a chef might eventually say to the server "Hey, that's all the orders we have. We're done." Make something like this happen in your program too, otherwise the server won't know when to stop serving.

The code for getting input from the user uses the append command for lists, which we haven't seen before. So here's the code for that part. Study it a bit - I think you'll be able to see what's happening.

```
orders = []
nextOrder = raw_input("Enter order (<Enter> when done): ")
while nextOrder != "":
    orders.append(nextOrder)
    nextOrder = raw_input("Enter next order (<Enter> when done): ")
```

4. Recall that the Caesar cipher is a code where each letter is shifted 3 "down" the alphabet. For example, all A's become D's, B's become E's, etc. Letters at the end of the alphabet should wrap around, so for example, X becomes A.

   Write a function named encodePar() (that is, "encode in parallel") that will ask the user to type in a message, and then encode that message in parallel, with two child processes. First, a demonstration:

```
>>> encodePar()
Enter message: The eagle has landed!
(first half): t --> w
(first half): h --> k
(first half): e --> h
(second half): h --> k
(first half): e --> h
(second half): a --> d
(first half): a --> d
(second half): s --> v
(first half): g --> j
(second half): l --> o
(first half): l --> o
(second half): a --> d
(first half): e --> h
(second half): n --> q
(second half): d --> g
(second half): e --> h
(second half): d --> g
The encoded message is:  wkhhdjohkdvodqghg
>>>
```

   Note:
   - The encoding changes all letters to lower case, and totally ignores all non-letters. So for example, that first w above is from the T of "The", and the last g is from the last d of "landed".
   - Spawn two processes. Name the first one "first half", and the second "second half". To do this, use the name argument to the Process function, like we saw in class. Do not pass an extra argument in the args tuple specifying the name.
   - The "first half" process handles the first half of the string, while the "second half" process handles the second half.
   - When both child processes are done encoding, they need to send their results back to the main process.
   - Make sure you control access to the shared stdout. No jumbled-up print statements are allowed!
   - It's quite possible that your processes won't "intertwine" in the way mine do above. The reason mine are so intertwined is that I added a time.sleep(.01) call in my encoding loop. Obviously you wouldn't want to do this in "real life", as it would slow things down. But for the purposes of this exercise, it forces a process to give up control of the processor a little sooner than it ordinarily would. If you don't put in a sleep like this, you may get something like:

```
>>> encodePar()
Enter message: The eagle has landed!
(first half): t --> w
(first half): h --> k
(first half): e --> h
(first half): e --> h
(first half): a --> d
(first half): g --> j
(first half): l --> o
(first half): e --> h
(second half): h --> k
(second half): a --> d
(second half): s --> v
(second half): l --> o
(second half): a --> d
(second half): n --> q
(second half): d --> g
(second half): e --> h
(second half): d --> g
The encoded message is:  wkhhdjohkdvodqghg
>>>
```

That would be alright for this exercise. I expect, however, that if you put the sleep in, your output should be intermingled like the first screenshot, so that would be a good thing to check. Depending on your computer, you might experiment with sleeping for longer than .01 seconds. Strings of different lengths can cause some different behavior as well.

5. In this problem, we will use the idea of parallel pipeline computation to compute the solutions to several binomial equations using the quadratic formula. First, let's see what the end result will look like. Then we'll dive into the details.

```
>>> quadPipeline()
What .txt file contains the binomial equations? binomials.txt
(Equation reader): Read: 5x^2 + 23 x + 9

(Coefficient extractor): Extracted: 5, 23, 9
(Quadratic formula computer): Solved: 5x^2 + 23x + 9
(Equation reader): Read: 14.5x^2 + (-12)x + 1

(Equation reader): Read: -2984632 x^2 + 7x + 8

(Coefficient extractor): Extracted: 14.5, -12, 1
(Equation reader): Read: 2.7182818284590 45x^2 + 2.342x + (-7.64)
(Coefficient extractor): Extracted: -2984632, 7, 8
(Quadratic formula computer): Solved: 14.5x^2 + -12x + 1
(Quadratic formula computer): Solved: -2984632x^2 + 7x + 8
(Coefficient extractor): Extracted: 2.71828182846, 2.342, -7.64
(Quadratic formula computer): Solved: 2.71828182846x^2 + 2.342x + -7.64
All solutions have been written to binomials.sol.txt
>>>
```

Note that I typed in the file name "binomials.txt". You can use a file with contents such as:

```
5x^2 + 23 x + 9
14.5x^2 + (-12)x + 1
-2984632 x^2 + 7x + 8
2.7182818284590 45x^2 + 2.342x + (-7.64)
```

Of course, as always, your code should work for any specified .txt file, and should generate a properly-named .sol.txt file.

Format of the Input File

As you can see, the input file (binomials.txt in the example above) is a file formatted in a particular way:

- Each line in the file corresponds to the left-hand side of a binomial equation.
- The right hand side of the equation is assumed to be 0.
- In the formatting of the file, `x^2` means `x` raised to the second power.
- You may assume that there will always be an explicit term for `x^2`, `x`, and the constant. For example, the file would say `4x^2 + 0x + 3` instead of `4x^2 + 3`.
- The operator between the terms will always be +. For example, the file would say `3x^2 + -2x + 1` or `3x^2 + (-2)x + 1` instead of `3x^2 - 2x + 1`. (Note the use of parentheses!)
- There are no specific rules about spaces. For example, the file could say any of the following:

```
3x^2 + 2x + 1
3x^2+2x+1
3x^2 + 2x+1
3x^2 + 2x + 1
3 x^2+2x    + 1
```

etc.

- There are two exceptions to the previous rule. First, `x^2` will always appear together, never as `x ^ 2` or anything like that. Second, if a coefficient is negative, the minus sign will always be right up against the coefficient.
- Multiplication between the coefficient and `x` term will always be implied. There will never be an explicit * symbol.
- Coefficients may be expressed as decimals, but not fractions.
- There are no blank lines in the input file. Not even the last line! (So be careful with this when you create your own file.)
- You may assume that every binomial equation in the file has real solutions. We will not be dealing with complex ("imaginary") numbers here. Please ask me if you need help on this mathematical concept.

Pipeline Design and Communication

In this program, you will have a parent process that will be in charge of the creation of three children, corresponding to three steps in the pipeline:

1) Read the equations from the file
2) Extract the coefficients from each equation
3) Solve the binomial equation using the quadratic formula and write the solutions to a file.

If we call these child processes p1, p2, and p3, then the following directions of communication are required:

p1: send to p2
p2: receive from p1, send to p3
p3: receive from p2

Graphically, we could think of this as:

p1 --> p2 --> p3

This is as we would expect for a pipeline. Note that each arrow above corresponds to a separate queue. You can also think of the treehouse metaphor we used in class: with p2's treehouse between p1's and p3's.

So the first arrow is the transmission of the line, as a string. The second arrow is the transmission of the coefficients, as a list of the form `[a,b,c]`. Yes, you can put a list on a queue with a single `put` operation.

Finally, you'll need the same "DONE" technique we saw in the chef/server example to enable your processes to complete when there is no more input to work with. If you don't do this properly, your processes won't actually complete. Rather, they'll be stuck waiting to get something else from a queue. This would be a problem and would cause you to lose some points.

To receive full credit, you must create three child processes to do these three steps above! So p1 should only read from the file and send the line on. p2 should only extract the coefficients and send them on. p3 should only apply the quadratic formula and write to the solution file.

Printing Progress

Recall that when I say that the processes are *interleaved*, that means that the operating system is executing context switches between them, to give the illusion that the processes are running at the same time even if we're only working with a single-core processor. Alternatively, we could say that the processes are executing *concurrently*. If you are unsure what I'm talking about here, please ask me about this.

You can see in the example that each process needs to give regular reports of what it's doing. I'm requiring this so that we can see the interleaving of the three processes. Since all three processes are printing, you'll need a lock to control access to stdout. Please examine the function below:

```
def printMessage(lock, s):
    lock.acquire()
    print "(" + current_process().name + "): " + s
    lock.release()
```

You are required to use this function, without making any changes whatsoever to it, for the printing in your three processes. This is a nice example of avoiding code duplication by splitting out some functionality into a separate function.

If you use this function correctly, you should not have any `print`s in the functions of your three children processes. They should just call `printMessage` instead.

Note that the code above assumes your processes have names, so please make it so.

Make sure absolutely all of your printing looks exactly like the example run above! The only possible differences, of course, are:
- Your actual interleaving may be different each time you run the program.
- Your results will be different if you provide an input file with different contents.

Interleaving of the Processes

There is one slight complication about how the operating system interleaves the three processes. We're actually not giving the processes very much work to do in this problem, so one process may actually run to completion before a context switch occurs even once to another process! One thing we can do, just for educational purposes here, is purposely make the processes take a little longer by inserting a `time.sleep(.01)` call into each of the child processes' loop. So please be sure to do this, otherwise your output won't appear interleaved as mine does above. You must do this!

It is important you understand what I'm talking about here with interleaving and why the `sleep` call changes things, so again, please make sure you talk with me if you're unclear on this.

Extracting Coefficients

Extracting the coefficients from a line read from a file is not as hard as it might first appear. Some tips:

- First, use `replace` (a string method) to get rid of stuff you don't care about.
- You should use `find` to find important markers like "x^2".
- You might, or might not, find it useful to use slicing to cut off and throw away part of a string once you're done working with it. If you do this in the right way, `find`ing "x" will actually get you the `x` term rather than the `x^2` term...
- Don't forget about `rfind` - that may be helpful too, or maybe not, depending on how you do things. See p. 96 of the text for all the string functions.

General Problem-Solving Strategies

It will be helpful to remember the general strategies we've talked about throughout the semester. In particular, I encourage absolutely everyone to break this problem up into pieces. For example:

1) Write a program that takes a string representing a binomial as input, and extracts the coefficients.
2) Write a program that takes binomial coefficients, and computes the solutions to the corresponding binomial equation.
3) Write a program that creates three child processes: p1, p2, and p3. Make p1 pass a number (or something) to p2, which then passes it to p3, which then prints it.
4) Write a program that does as in #3, but p1 reads numbers (or something) from a file in a loop, passes them one at a time to p2, which receives them in a loop and passes them one at a time to p3, which receives them in a loop and prints them.

Do you see how solving the above four smaller problems is directly related to solving the main problem here? If you don't, please look into this a bit more and talk with me. If you take this problem one step at a time as I've shown above, it will be much easier to handle things, rather than trying to do everything at once and having a huge number of problems everywhere all at once. If you ignore this advice, I promise this problem will be very hard!