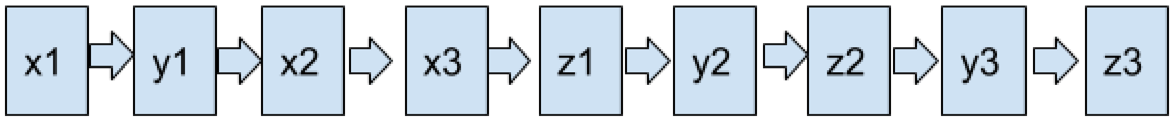
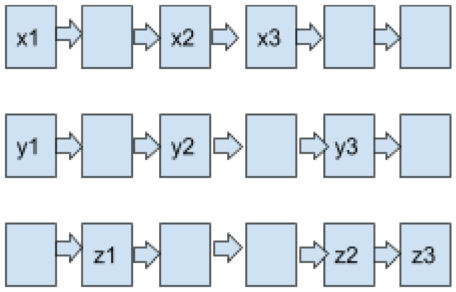
Will Suitor
Multicore - HW 1

1. Concurrency is a more loosely-defined than parallelism. Concurrency only requires events to happen within the same, arbitrarily-long time frame while parallelism requires events to be happening literally at the same time.

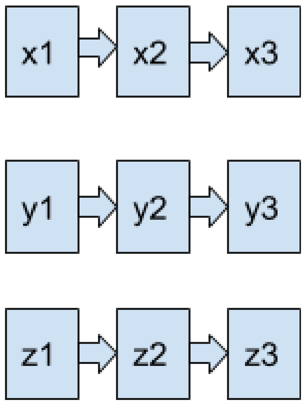
For example, we have three programs x, y and z, each with three instructions. Then on a single processor, concurrency could look like:



Here, x and y are concurrent, y and z are concurrent, but x and z are not concurrent.
Running the programs concurrently on three processors could look like:



OR



Now, all three programs qualify as being run concurrently in both diagrams. In both, all three programs are begun within the first two time units. So, while concurrency does not necessarily require execution at exactly the same time, the events just need to happen in the same time frame.
Running the three programs in parallel on three cores would look like diagram 3. The three programs are being run in parallel because the instructions are being run at the ‘same’ time. Essentially, x1 and y1 are actually occurring at the same time when run in parallel but that is not necessarily true when run concurrently.
From this, we can say that the idea of running programs in parallel on a single core is impossible. A single core can only execute one instruction at a time.

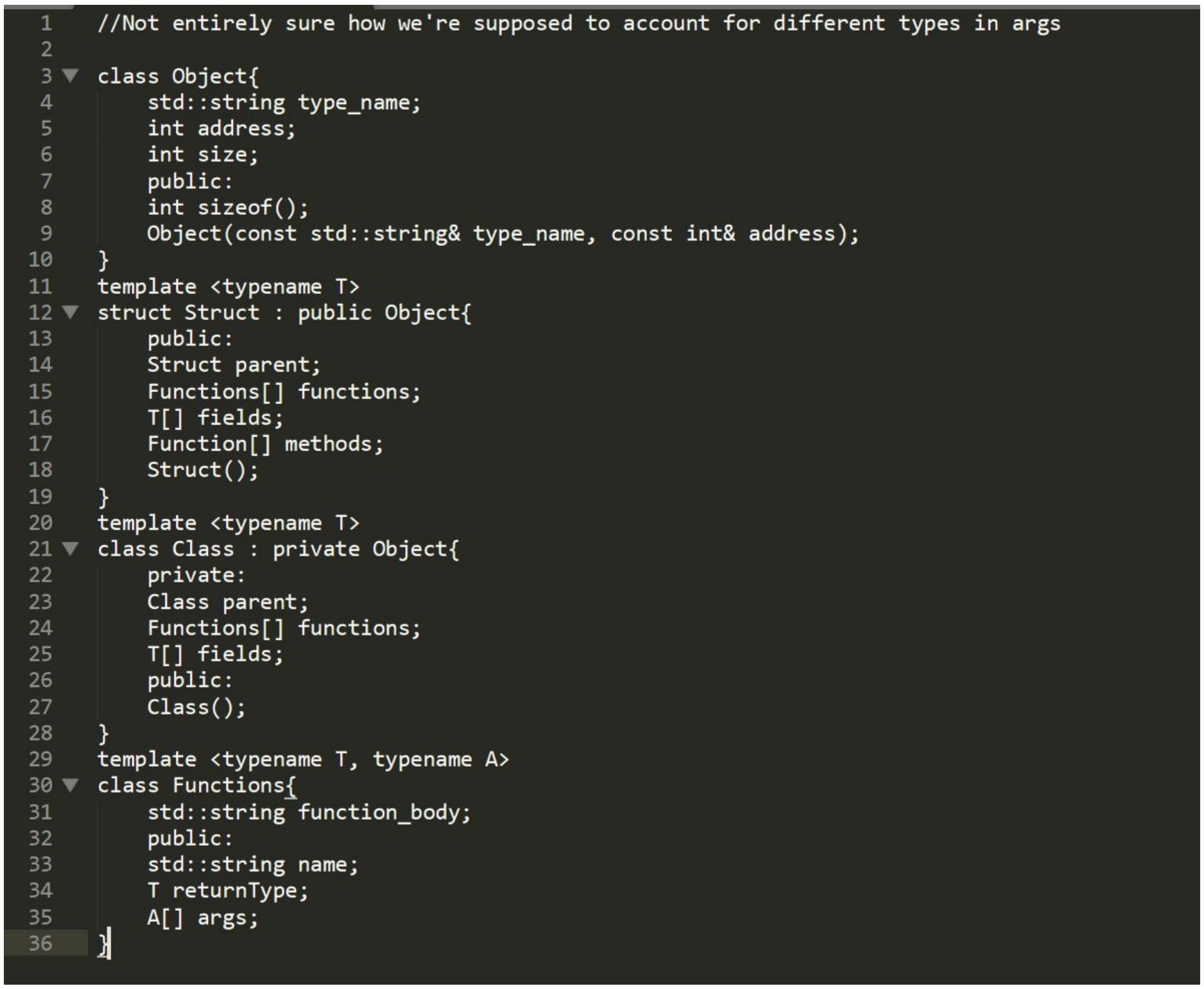
2. I would use the logP model. The logP model is optimized for systems where message-passing is the most integral piece of the system. This question is essentially specifying a system in which there is little computation over the system but a lot data needing to be moved around. BSP breaks large computations down into multiple pieces which would be superfluous here (especially the synchronization). There is no suggestion of one shared memory, so the

PRAM model would not make sense either. Therefore, it seems like logP would be the best option.

3. Speed-up = 1/ (F + (1-F)/P)
a. 1 CPU - Speed-up = 1/(.37 + (.63)/1) = 1
b. 2 CPUs - Speed-up = 1/(.37 + (.63)/2) ~= 1.46
c. 4 CPUs - Speed-up = 1/(.37 + (.63)/4) ~= 1.90
d. 8 CPUs - Speed-up = 1/(.37 + (.63)/8) ~= 2.23
e. 12 CPUs - Speed-up = 1/(.37 + (.63)/12) ~= 2.37
f. 16 CPUs - Speed-up = 1/(.37 + (.63)/16) ~= 2.44
g. Infinite CPUs - Speed-up = 1/(.37 + 0) ~= 2.70

4. Pipelining is essentially breaking up individual instructions into component, sequential pieces. This allows multiple instructions to be run "at once" where one piece of 5 different instructions can be run within a cycle. Hazards force this process to halt and wait for one specific instruction to complete before continuing the pipeline. This obviously reduces the IPC because instead of completing 1 instruction per cycle (assuming a constant flow of new instructions), the pipeline has to wait for 1 instruction to go all the way through its five pieces. This means parts of the CPU has to lie dormant for a few cycles, reducing the amount of instructions completed per cycle.

5.



```
//Not entirely sure how we're supposed to account for different types in args

class Object{
    std::string type_name;
    int address;
    int size;
    public:
    int sizeof();
    Object(const std::string& type_name, const int& address);
}
template <typename T>
struct Struct : public Object{
    public:
    Struct parent;
    Functions[] functions;
    T[] fields;
    Function[] methods;
    Struct();
}
template <typename T>
class Class : private Object{
    private:
    Class parent;
    Functions[] functions;
    T[] fields;
    public:
    Class();
}
template <typename T, typename A>
class Functions{
    std::string function_body;
    public:
    std::string name;
    T returnType;
    A[] args;
}
```