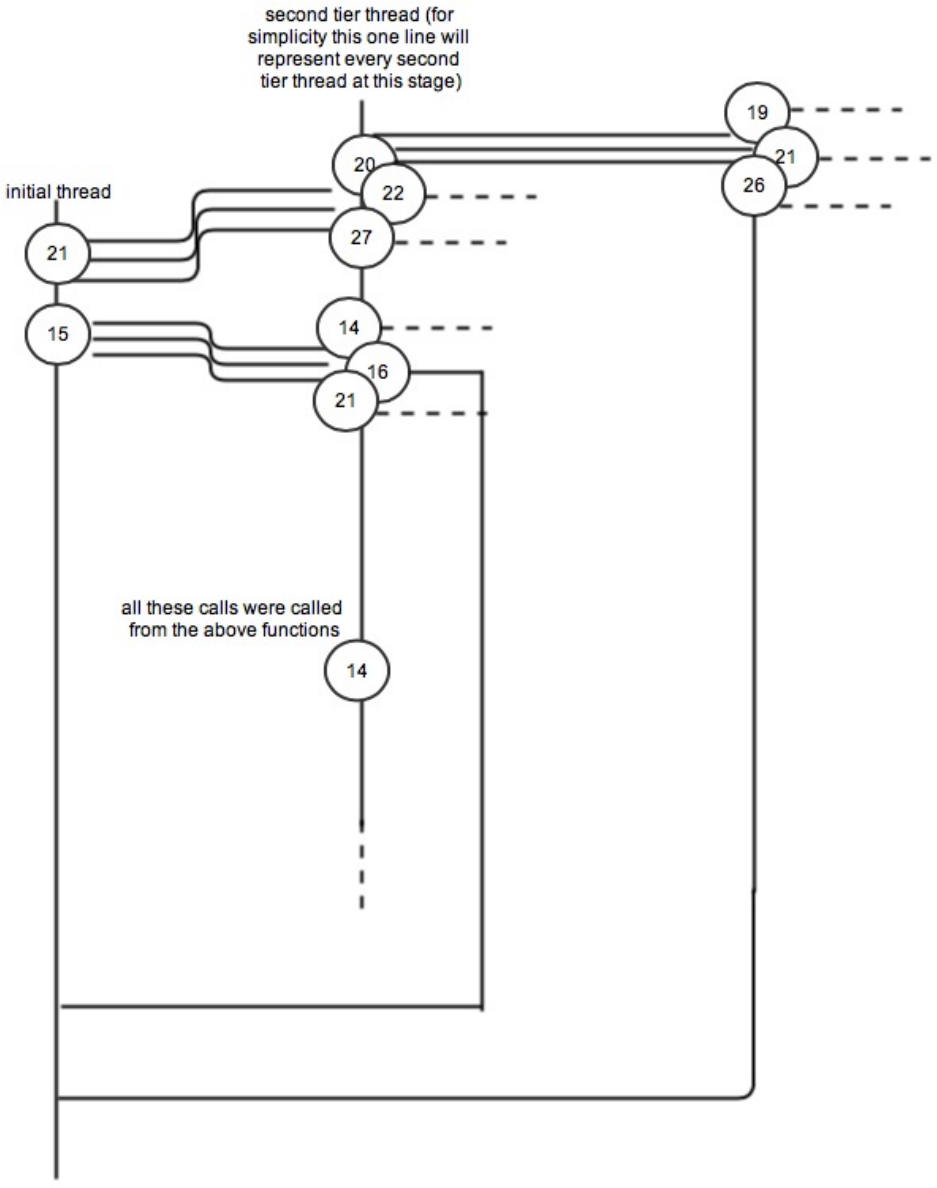
Tallack Graser
HW 4
5/12/14

Fork-Join Pattern

1) The fork-join pattern allows us to spawn off function calls or computations into a new thread allowing the current thread to continue without knowing the result or what is returned. At some point in the program the threads will be joined (each thread waits for all threads to get there) then only one thread continues on with all the results from all the threads.

2)

a)



Once we get to a black square (I'm assuming the call meant to be FloodFill(21, white, black)?) we return back to the previous thread stage (or parent).

b) The amount of concurrency is determined by the size/number of cells needing to be colored. So the total available concurrency can be found by multiplying the length and height of the area and multiplying by .75 (since 25% of the calls are made on the main thread).

c) There are data races in the sense that one cell may be being looked at by two different method calls. One method may be changing the characteristic of the cell, while the other one isn't getting the updated changes. This may cause non-deterministic behavior in exactly our program will end, but will not hamper the accuracy of our program (in this case).

3) Parallel slack is the amount of extra parallelism above the minimum necessary to use the parallel hardware resources. Parallel slack is important to keep in mind when considering fork-join problems because as seen in the above problems, we can create many children extremely fast. Hence, we have to know approximately how many processors (and slack) we have and be careful not to overload our hardware with an overbearing amount of work.

Pipeline Pattern

1) As the number of data items in a pipeline increases, it will eventually pass the number of serial tasks that must be done. It is at this point when the performance of the system will depend on the number of data items (since the parallelism is decreased) and not the parallelism that can be achieved though number of serial tasks.

2)

Stage-Bound workers: A stage-bound worker can be thought of as an assembly line of sorts. An item is worked on by a given worker, then is passed down the line to the next worker and so on. This is easier to implement, but there must be some kind of communication between the workers.

Item-Bound workers: one worker stays with the item from initialization to completion. This allows for little communication since the same entity is working on the same item through out the whole pipeline.

Hybrid workers: As the name implies, this kind of worker is a combination of the previous workers. A hybrid worker may start as a stage-bound worker but may switch to an item-bound worker depending on the characteristics of the item.

*In general I would probably prefer hybrid workers since they would give you the most versatility in what you can do computationally to each item. However, depending on the size/restrictions of the program, I may choose stage-bound workers for simplicity of implementation.