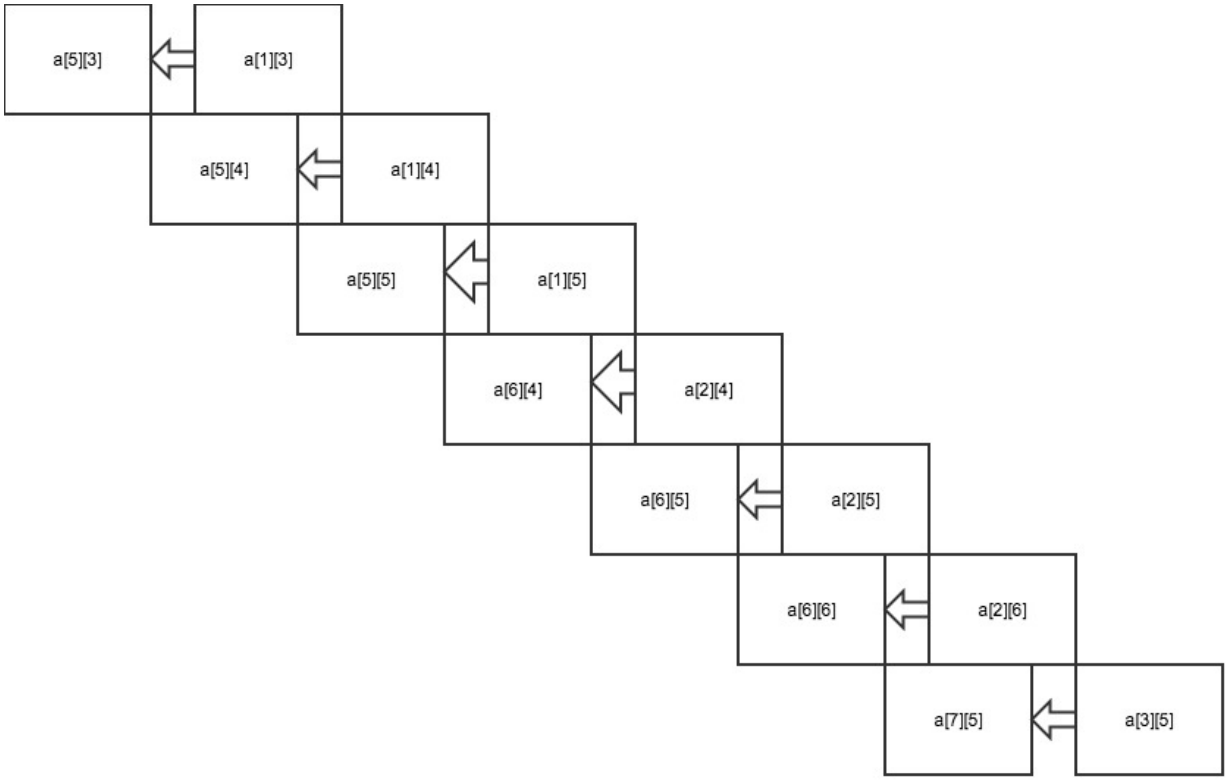
Jack Ziesing
hw 2

1.



2. This nested loop allows for the outer loop to be paralyzed as a map while the inner loop should remain less paralyzed. The inner loop can be broken into chunks of three to get better performance and paralyzation, but maybe still a map.

3.

```
cilk_for i = 5 to 1000 do
        cilk_for j = i-2 to i do
                a[i][j] = a[i-4][j]
        end
end
```

Patterns

1. The stencil seems good because not only did we use it in a lab to blur an image, but its properties seem fitting. The stencil is fitting because it paralyzes by a map with knowledge of its

neighbors, which is good because thats how we blur an image and close access to the properties we are changing help performance.

2. The gather or scatter algorithm seems right for this problem because it does exactly what is described. Gather may be better because it is making a subset of the original data, but both keep a list of indices so either seem fine.

Map Pattern

1. A map is very desirable when the algorithm or loop has little or no dependencies. In this sense if each iteration of a loop can be computed independently without reliant on another iteration in the loop. The map executes loops like these very fast, most likely faster than any other parallel algorithm.

2.

```
cilk_for (i=1; i<=n; i++) {
        int sum;
        for(j=1; j<=m; j++) {
                sum += a[i][j] * b[j];
        }
        c[i] = sum;
}
```

3. Fusing maps together is faster and better because it can be thought about as a reduction. If you are able to chunk the map up into smaller pieces and create a tree of these chunks and then compute the chunks in parallel and combine each level of the tree until the entire solution is done you will get better performance than if you performed multiple maps one after another.

Collective Pattern

1. This is going to be somewhat like a mergesort, except you will divide each word into its own array and then work up from there. You combine two arrays making a new subset of the elements, but in sorted order, then you work up a tree and keep merging until you have one array that holds all the elements in sorted order.

2. The tiling happens when the problem/data is broken into smaller parts. If we are performing a reduce or scan, and our problem is too big to fit into cache, tiling helps out because it divides the program up to allow parallel computations on chunks to work through the length of the problem.

3. The danger with paralyzing this is that some computations are dependent on each other, or rather produce different answers based on the order of the computation. For example if we add then multiply we get different results than if we multiply and then add.