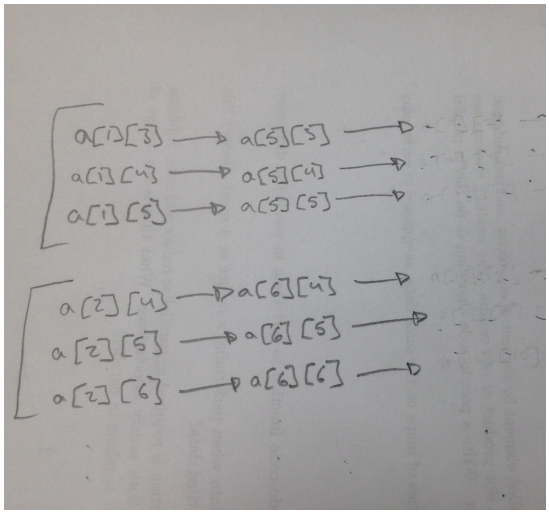
Tallack Graser
CIS 410
HW 2

Dependencies

1)



2) after sketching out the dependencies it doesn't look like there are any dependencies between iterations in the outer loop, hence, we can parallelize the outer loop with a standard map pattern. SInce there are no dependencies between iterations in the inner loop either, we can use another simple map pattern in the second, nested, loop.

3)

```
cilk_for i=5 to 100 do
        cilk_for j=i-2 to i do
                a[i][j]=a[i-4][j];
        end
end
```

*if there were dependencies between the iterations in the outer loop, then the fact that we had three threads available to us would mean that we could perform the all three iterations of the inner loop at the same time.

Patterns

1) We should use the stencil pattern. The stencil pattern is a generalization of a map, and is often/easily used with iterative solvers (this example is basically given to us in the slides).

2)We should use the gather pattern. Gather reads a collection of data, where the output collection shares the same type as in the input, but it shares the same shape as the indices collection.

Map Pattern

1)Map is a desirable target in terms of parallel programming because it is probably the easiest pattern to implement, and increases the performance of loops by a lot (probably even more than any other pattern).

2)

```
cilk_for(int i=0; i<num_of_columns; i++){
        int total=0;
        for(int j=0; j<num_of_rows; j++){
                total=total+(A[i][j]*B[j][0])
        }
        C[i]=total;
}
```

3) The overhead of implementing a map can be relatively high, (if you're not mapping a very high number of elements, sometimes the serial for loop will be faster). Hence, by "fusing" together multiple maps into one, you are limiting the overhead that will be spent on implementing the map, and spending more time on the hard computations.

Collective Patterns

1)The algorithm to do this would be similar to a merge sort. All of the words would be in their own array, then every array would combine with another array, ordering the words in alphabetical order in doing so. Since there are many subprocesses of combing two array at a time (that are all independent of each other) you can do all this combining in parallel, until there is only two arrays left to merge together.

2)Yes, tiling does/can lead to less parallelization, but if you have dependences in the data that must be reduced serially, then tiling gives you a way to still parallelize your code to some degree. This works by gathering all the data that is dependent on something and grouping it together (you'll have multiple groups). Then, you can perform a serial reduction on all these groups in parallel.

3)The dangers of doing floating point addition and multiplication in parallel (althought they are associative) is that they are done with saturation. That is, if the result of a computation is outside the allowable range, the result is clamped to the closest representable range, instead of overflowing like it normally would. This would then not give you the correct solution.