Watts-Strogatz Experiments on CArray and KokkosCarrays

Calvin Roth

July 22, 2022

1 Introduction

In this report I detail the methods and results of the floyd test. In this test I generate 1000 node watts-Strogatz graph with different parameters, run floyd warshall over this graph, and report the average shortest path distance over the graph.

1.1 But what is a Watts-Strogatz Graph

A Watts-Strogatz graph is a random graph model introduced in the paper Collective dynamics of small-world networks by Watts and Strogatz. This graph type has many interesting properties such has the clustering coefficient(how many 3 vertex triples are connected triangles) and for this experiment the average shortest distance has been well studied.

A watts-Strogatz graph, forever more in this report a WS graph, is created by the following process with parameters n, k, and p. 1. Make an empty n vertex matrix
2. For each node i connect i with a directed edge to it's nearest 2k neighbors. This forms a ring lattice. 3. For each edge rewire it with probability p to and random vertex 3b. For practical reasons we dis allow self loops and repeated edges

The elegance of this graph is showing that in a world where most people or entities have a lot of local connections even a very low number of "far" edges to anywhere in the network can dramatically decrease the average shortest path in the network. This sort of graph is common in network. The behavior that many networks exhibit of having a surprisingly small average shortest path is a phenomenon that this relatively simple model hopes to capture.

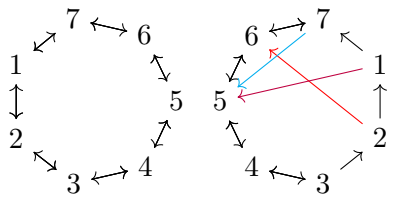

Figure 1: On the left, ring lattice with 7 elements where each node is connected to it's nearest 1 neighbor. On the right we replace three edges with random edges. In particular we replaced (1,2) with (1,5), (2,3) with (2,6) and (7,1) with (7,5). This is a simple showcase of the Watts-Strogatz random graph process.

2 Expected distance calculation

When there is no rewiring, that is each node is connected to the k nearest nodes on each side, calculating the average distance is easy. First we observe that we only ever have to go half way around the ring lattice to get to any other node so we will calculate the average distance of the $N/2$ nodes clockwise from us. The first K are only one step away, the next K are 2 steps, and so on. The last bunch will be $ceiling(\frac{N}{2k})$ steps away. So we get that the average distance is

$$\frac{2}{N}\sum_{i=1}^{ceiling(\frac{N}{2k})} k \approx \frac{2}{N}k\frac{N(N+1)}{8k^2} \tag{1}$$

$$= \frac{N+1}{4k} \tag{2}$$

$$\approx \frac{N}{4k} \tag{3}$$

On the other extreme if p=1 then this graph looks like an erdos renyi graph where each node has a fixed out degree. The expected distance in an Erdos-Renyi graph model is logarithmic in terms of N so we should expect something similar.

3 The experiments

The primary expect is to test MATAR CArrays and kokkos enabled CArrays for speed with varying problem size. In this test case we initialize a WS graph by assigning the local edges, flipping a weighted coin for every edge corresponding to the rewire probability in the times we do rewire we change this edge from a local edge to a random edge. There were no safeguards put in place to make sure double edges are prevented and in theory this makes walks slightly longer. After initializing this graph, we run the Floyd-Warshall algorithm to compute the shortest path length for each pair of vertices. This is an $O(n^3)$ algorithm. The full pseudo code of Floyd-Warshall is provided below.

```
G = // adjacency matrix of graph
res = // Graph with 0 on diagonal, values of G on off diagonal and infinity elsewhere
for k in range(0,n):
    for i in range(0,n):
        for j in range(0,n):
            res(i,j) = min(res(i,j), res(i,k) + res(k,j)
return res
```

After we have computed the shortest distance between every two nodes we compute the average distance by computing a reduced sum over all the entries of res and then dividing by n^2. To parallelize this code we compute the i and j loops in any order but the loop over k is ran sequentially. Note that in kth outer loop $res(i,k) = min(res(i,k), res(i,k) + res(k,k)) = res(i,k), res(k,j) = min(res(k,j), res(k,k) + res(k,j) = res(k,j)$ i.e. in the kth iteration the kth row and column don't change. This is essential to correctness of the code in parallel. The parallel version of this code written in C++ with MATAR is

```
 for(k = 0; k < n_nodes; k++){
        FOR_ALL(i, 0, n_nodes,
                j, 0, n_nodes, {
                 if(i != j){
                    int dist1 = res(i,k) + res(k,j);
                    int dist2 = res(i,j);
                    if(dist1 < 0){
                        dist1 = INT_MAX;
                    }
                    if(dist2 < 0){
                        dist2 = INT_MAX;
                    }
                    if(dist1 < dist2){
                        res(i,j) = dist1;
                    }
                }
        });
}
```

The checks for negatives are because we are using INT_MAX instead of $+\infty$, if we add something to INT_MAX if it less than 0.5∗INT_MAX we will get a negative number which we correct immediately by resetting it to INT_MAX. This is trying to mimic the rule that $\infty + x = \infty$ for all x.

4 Time results

For the time section the rewire probability is 0.00 and we connected to the 12 nearest nodes(i.e the 6 nearest on each side of the node). Timing was done a volta-x86 machine with an NVIDIA Tesla V100 with 32 gb of memory, a Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz,

Note that since this algorithm is $O(n^3)$ a doubling of the node size should correspond to an 8x the amount of work being done so the parallel version of the code is a better than it first appears. The speedups gained for moving to different architectures were on average 27.8x speedup going from serial to GPU, 11.7x speedup from serial to CPU, and a 2.4x speedup going from CPU to GPU.

5 Analysis of Average distance when P=0

In this section, we sill connect to the 12 nearest nodes and we set the rewire probability to 0 in order to test our math that the average distance scales linearly with the number of nodes in the graph. We see that it does.

We see there is no effect on correctness when the code is ran in serial vs parallel which is good.

6 Analysis of average distance changing P

Here we vary the rewire probability of any given edge. There is no reason to run the code both serially and in parallel so we just run in parallel. Parameters are node size= 4000 and connect to the 12 nearest neighbors.

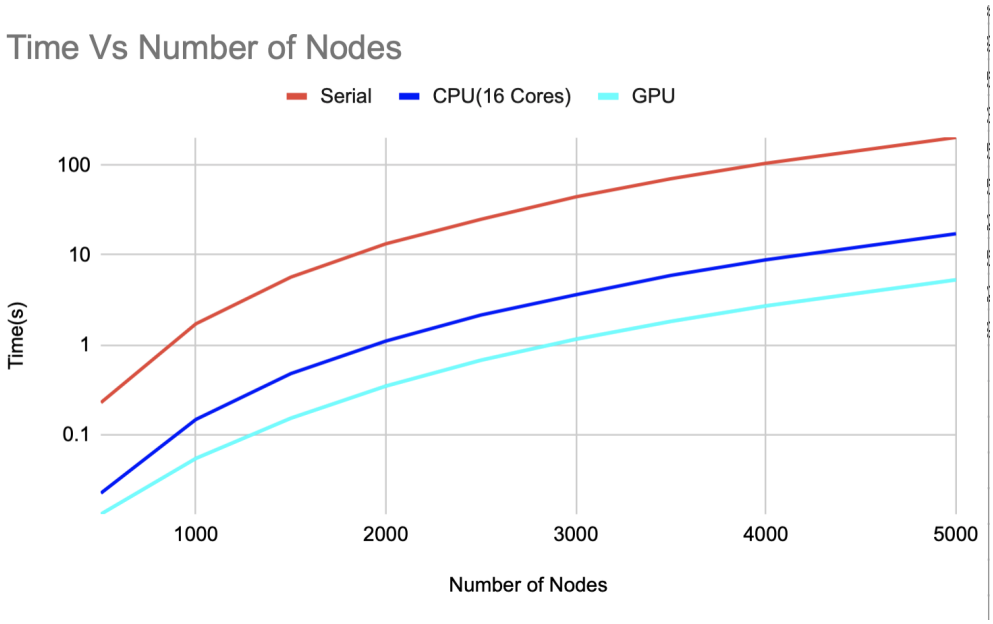

Figure 2: Comparison of times taken when ran serially, on the gpu, and on the cpu with 16 threads

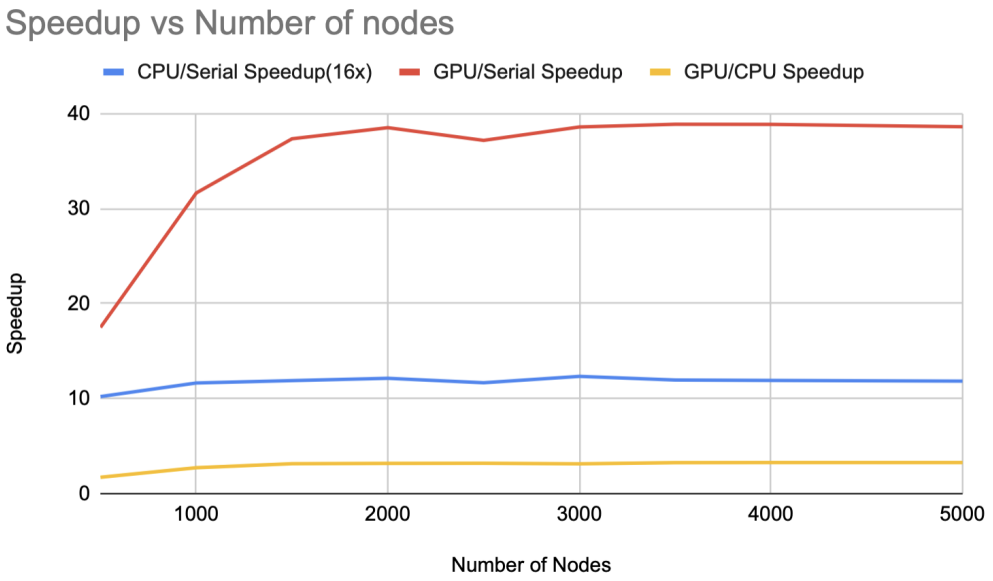

Figure 3: Speedups experienced from moving from serial to GPU, serial to CPU, and CPU to GPU. Aside from small values of n the speedups we get are nearly constant. Actual data in github. GPU and CPU speedups are truncated because we the largest serial calculation we did was 4000x4000.

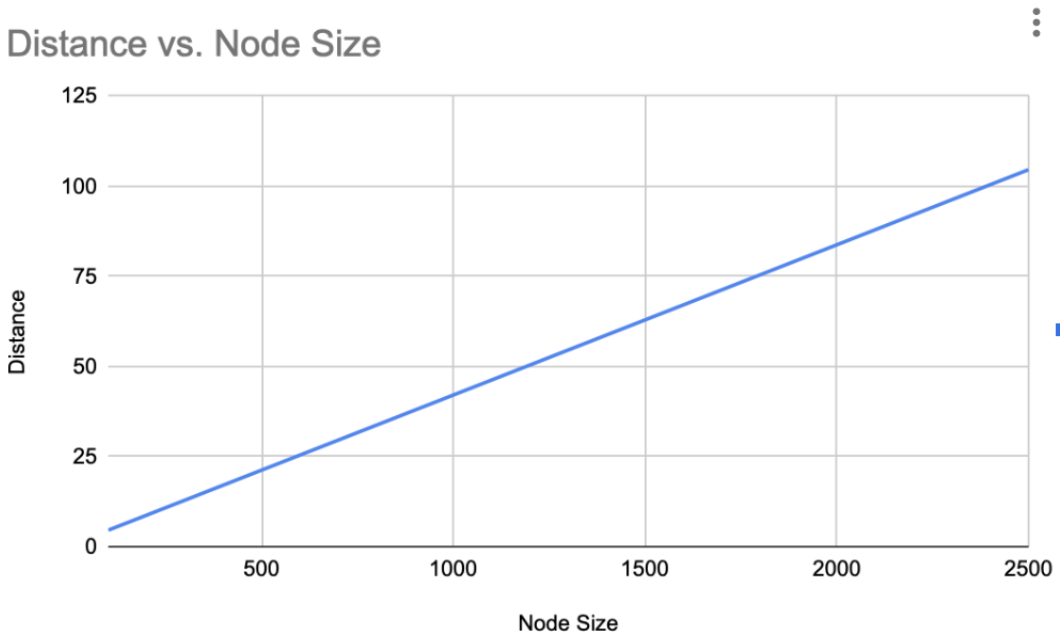

Figure 4: Average distance when ran in serial

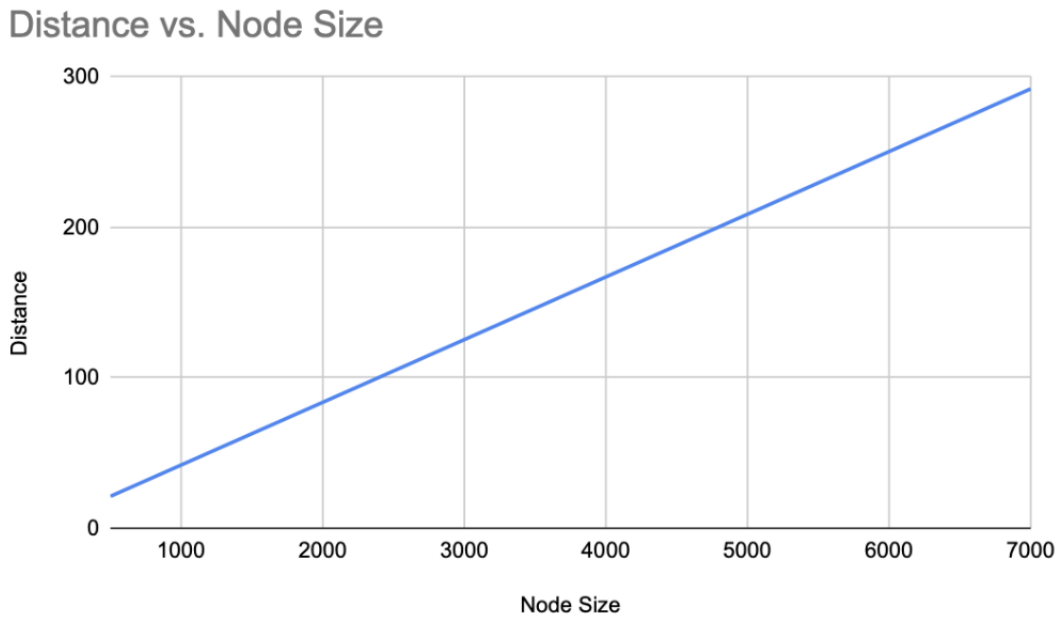

Figure 5: Average distance when ran in parallel

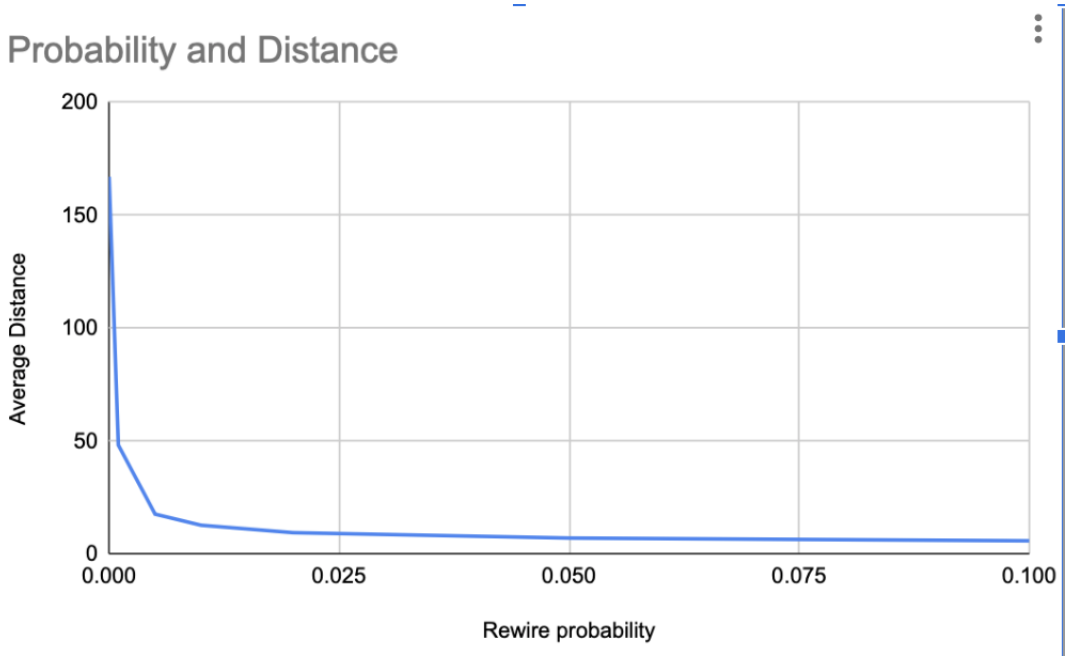

Figure 6: Affect of changing the rewire probability on average distance in the network. The effect even for small probabilities is quite strong

7 Conclusion

In this report we showed that CArrayKokkos is able to utilize parallelism effectively dramatically reducing the runtime and making larger problems feasible. Furthermore we validated the already known behavior of this random graph model. This model gives us reason to believe real world networks such as the facebook friendship network should have a surprisingly low average distance even though most of our friends are local i.e. most peoples friends are mutual friends more frequently than random chance would suggest. Finally, we note that the algorithms in this paper are easily parallelizable and converting from serial to parallel with the exception of random number generation is straightforward.