Chapter 1

Jaccard Coefficients

Contributed by Neil Butcher

1.1 Introduction

Jaccard Coeffcients is a proposed High Performance Computing (HPC) benchmark that is used in a wide variety of real world applications. The Jaccard benchmark is used as a way to define similarity between the neighborhood of two nodes. The definition of a neighborhood of a node is the adjacent vertices to a specific node. The Jaccard metric was originally introduced as a way to detect communities in botanical species [3]. This idea has been further expanded to other community detection algorithms[5], [1] and for other purposes. The Jaccard coefficient has been used by Wikipedia [2] to determine the relationship between web-pages based upon common authors of pages. Jaccard is an interesting problem because it replicates the complexity of real world graph problems without being overly complex.

The example in figure 1.1 shows a problem that is best solved by using Jaccard coefficients. Often an insurance company will attempt to determine reliability of a person before offering them insurance. There are many ways that the reliability of a person can be determined, for instance people they have shared a residence with. This can be represented as a graph problem, that can in fact be solved by Jaccard! This is one of many examples of problems that can be solved using the Jaccard coefficient. [4]

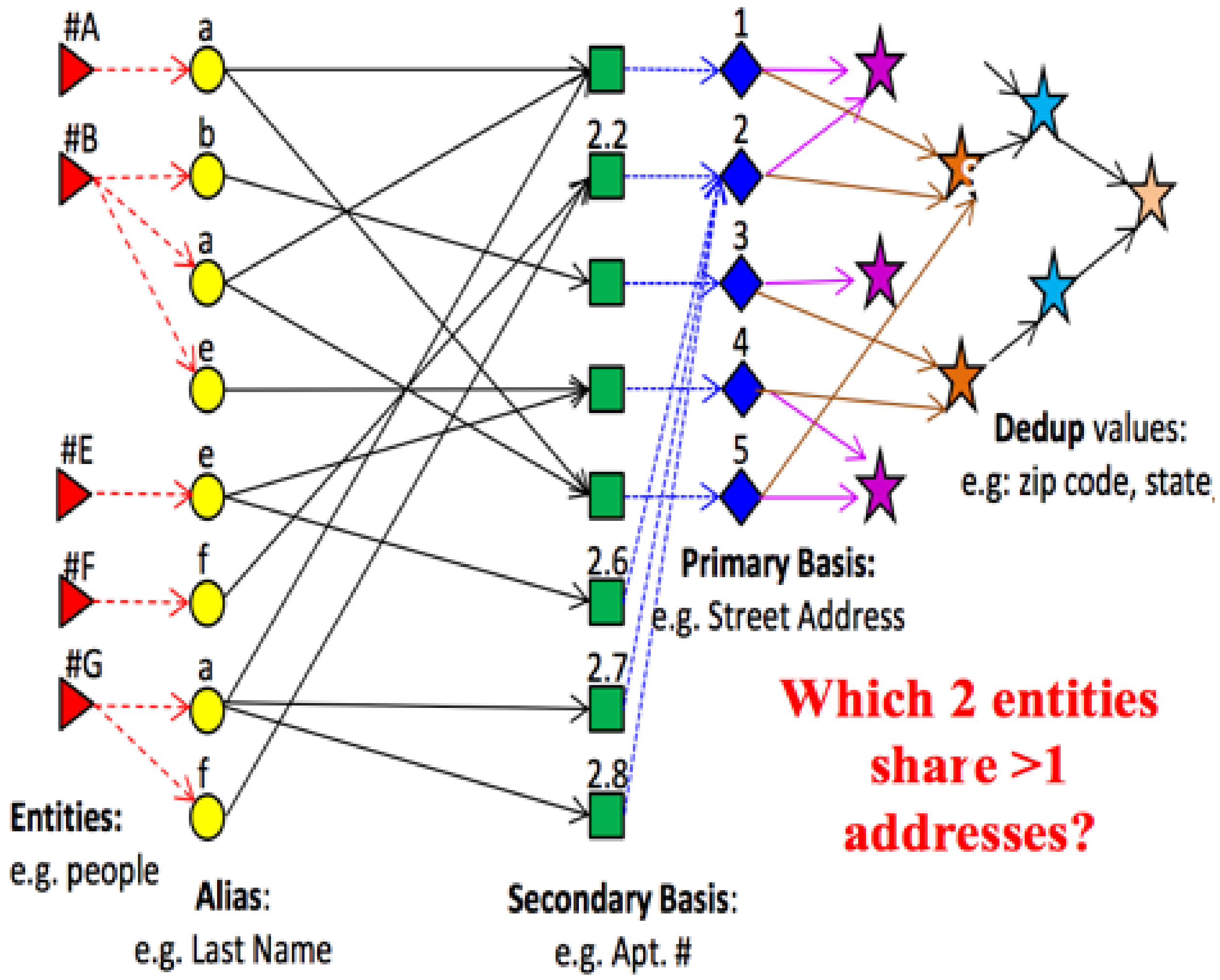

Figure 1.1: Example in which Jaccard Computations a relevant

1.2 Basic Definition

The Jaccard coefficient gives a value that represents the similarity of the neighborhoods of two vertices. Given a pair of vertices U and V we represent Jaccard as the intersection of the neighborhoods of U and V, divided by the union of the neighborhoods of the two vertices. From a computational standpoint most work comes from computing the intersection of U and V. This is because to compute the union you can simply take the size of both the neighborhoods, add them together and then subtract by the size of the intersection, so that the shared vertices will only be counted once.

For examples look at Figure 1.2. The figure shows a simple graph in which it is easy to demonstrate Jaccard computations. For example we demonstrate how to compute the Jaccard for vertex A and vertex D. The intersection of their neighborhoods is the single node, vertex B. The size of the union is two, nodes C and B. Note that despite that both A and D are neighbors of B, we only count B as one node in the union. This makes the Jaccard value 1/2.

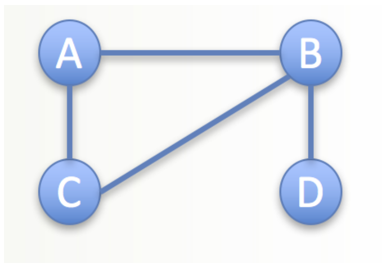

Figure 1.2: Small graph with simple Jaccard computations

1.3 How to Compute Jaccard

There are many different ways that a Jaccard can be computed. The most efficient way to compute a single Jaccard value depends on how the data is formatted. Given neighborhoods that are sorted based on vertex id greatly simplifies computing the intersection. To do compute given a sorted neighborhood we iterate through each neighborhood simultaneously, checking for collisions. We only iterate on the list that currently has the smaller vertex id. Then once we have counted all the intersections we can compute the union in constant time given that we know the size of the neighborhoods. This results in a complexity of O(M), where M is the size of the neighborhood larger of the two neighborhoods. In the case that the neighborhoods are not sorted by vertex id we have two choices. We can sort the neighborhoods first O($Mlog(M)$), then we use our previous sorted neighborhoods algorithm. The other option is the just do a brute force comparison of the neighborhoods determine the intersection, which has a complexity of O(M^2). This may be more efficient then sorting, given that it may be a rather large constant to perform the sort.

Algorithm 1 JaccardPair

```
1: Given a pair of vertices U, V with sorted neighbor lists
2: intesect starts at 0
3: Nv is the current neighbor of V, starting with the first value in the list
4: Nu is the current neighbor of U, starting with the first value in the list
5: While Nv or Nu can still iterate
6: if Nv == Nu then
7:     intersect++
8: if Nv greater then Nu then
9:     Nu++
10: if Nu greater then Nv then
11:     Nv++
12: End While
```

Often real applications that want to utilize Jaccard coefficients require a Jaccard value for all pairs of vertices. The obvious solution is to just perform a brute force comparison and compute on all nodes. This gives a complexity of O(N^2), where N is the number of vertices in the graph. This can be further improved if we attempt to find a way to 'ignore' the '0' value Jaccard coefficients. This essentially comes down to ignoring vertices that do not share a two-hop path. If they do not share a two-hop path there it is impossible that they have any intersecting vertices. The pseudocode that performs a search of just two hop paths can be seen below. A summary of the concept is that starting from each vertex check all two hop paths, if you haven't computed the Jaccard yet on any two hop path you find, then compute it. Since this algorithm revolves around finding two hop paths the most benefit will be seen in the cases in which a graph is fairly sparse. We start by going through each of the verticies which is O(N). Then for each vertex we go through each of it's neighbor's neighborhoods. This results in a time complexity of O(M^2), where M is the average size of a neighborhood. Then of course we have to accumulate for each unique two hop path we find.

Algorithm 2 Jaccard

```
1: for each Vertex V do
2:     for each Neighbor N of V do
3:         for each Neighbor J of N do
4:             intersect[V][J]++
```

Jaccard can also be computed by using the Graph Basic Linear Algebra Subroutines (GraphBLAS). The GraphBLAS library is a simple C interface that represents graphs as matrices and performs matrix operations to perform graph algorithms. Jaccard can easily be transformed to a matrix operation, the intersection of a all pairs of nodes can be done by a matrix multiply operation. A graph can easily be converted to a matrix by making an NxN matrix G and given an edge from Av to Au we set G[v][u]=1 and G[u][v]=1. Then we perform GxG, the values in the results matrix will end up representing the intersection of the neighborhoods of the corresponding nodes (G[u][v] is the intersection of u and v). Now that the value for the intersection of two neighborhoods is known, computing the union is: Adding the size of the two neighborhoods and subtracting the size of the intersection: —Nu— + —Nv— - —I(u, v)—. Having computed both the intersection and union of the two neighborhoods computing the Jaccard is now just a division. Implementing this is as simple as performing a matrix-matrix multiply and then iterating through the results. There are many different parallel implementations of GraphBLAS that currently exist. This is a scalable and simple way to implement Jaccard.

1.4 Dataset Used

For the purposes of this project we would like to adjust the density of the problem. Being able to adjust the density of the graph can lead to different computational bottlenecks. Dense graphs will result in iterating through larger neighborhoods in turn raising the memory dependence of the problem. The algorithms we develop in this work target a sparse graph and so increasing the density gives us a strong idea of our weaknesses. We chose the RMAT graph generator test the efficiency of our code. The RMAT graph generator is convenient because it allows us to produce large and diverse types of graphs and test the efficiency of our solutions on a wide variety of problem types. The main downside of RMAT is it is not emblematic of real world datasets, and can often have impactful structural differences from real world graphs.

In conducting experiments to observe the performance characteristics of different Jaccard algorithms there are a wide variety of data sets to choose from. The simplest choice is the RMAT graphs. These graphs are a dramatic simplification of real world problem, but are easy to demonstrate strong scaling behaviour. The RMAT graphs are artificial graphs produced for benchmarks, most notably Graph500. This makes RMAT graphs an interesting data point because one of the goals of developing Jaccard codes is to implement in as a benchmark alongside the BFS benchmark in Graph500.

Jaccard was initially developed as a community detection metric. This makes it an obvious candidate for graphs with clear and noticeable communities. Previous work computes the similarity between Wikipedia pages. These data sets are available and make an obvious comparison point for any new work that is performed. There are also many other SNAP datasets that exist that have many clearly defined networks. For the purposes of this project we chose to focus on RMAT graphs to emphasize scalability and simplicity.

1.5 A Sequential Jaccard Implementation

We have developed a simple sequential Jaccard implementation and performed experiments to display some baseline performance characteristics. The algorithm we used was shown in Algorithm 2, we only compute the number of non-zero values. This adds in the requirement that we now examine all two-hop paths which given a very dense graph can be a large overhead and may add more work then just computing Jaccard for all-pairs. The results from the sequential implementation can be seen in figure 1.3. The chart shows that as the density of the graph goes up (number of edges) the execution time rises at a dramatic rate as well.

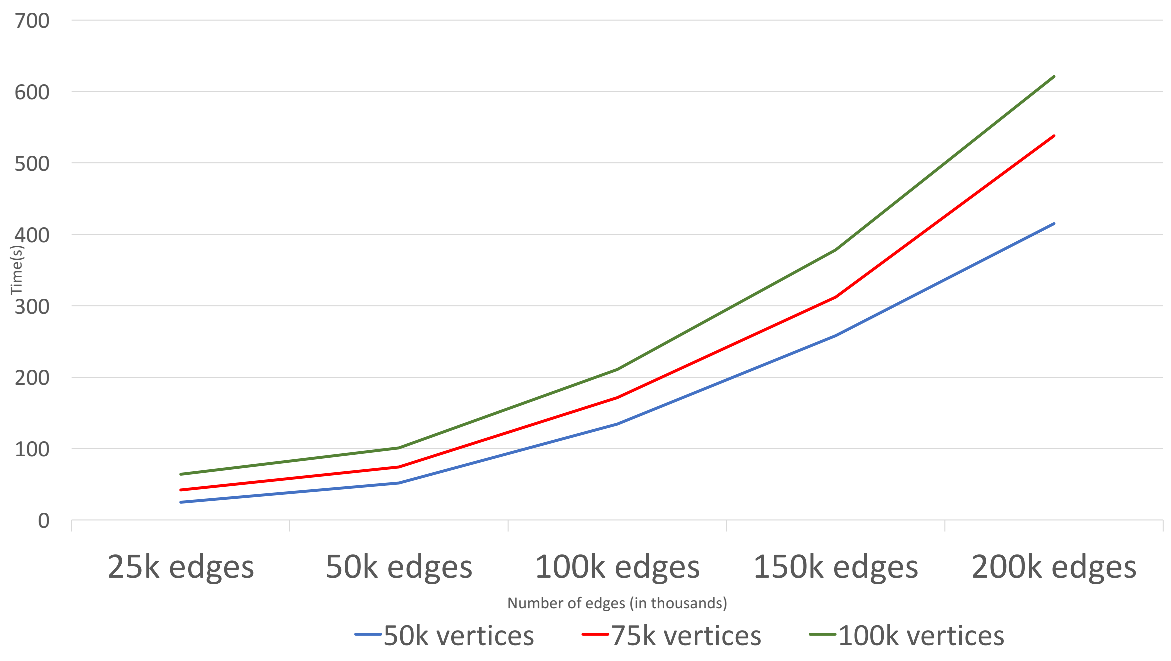

Figure 1.3: Sequential result of Jaccard computations. As we increase the number of threads

1.6 Parallel Jaccard Implementations

There are a wide variety of computational tasks that can utilize the Jaccard coefficient. Since each Jaccard coefficient can be computed independently of the other, parallelizing the computation is fairly straightforward. There however can obviously be multiple caveats to computing the Jaccard coefficient. The way that work is distributed and created is incredibly important. The obvious solution is to just take our sequential implementation and assign the work done on each vertex to a single thread. This has the added overhead of requiring each thread to provide atomic access to a 'intersection' counter. The alternative is to spawn a task for each pair of vertices and compute all pairs of Jaccard. The is bound to perform better given a dense problem.

We utilize the Intel Knights Landing (KNL) chipset which contains multi-channel DRAM (MCDRAM). The MCDRAM provides a no latency benefit over DDR, it does provide more bandwidth then conventional DDR. The downside of MCDRAM is it is significantly smaller then DDR and so often the entirety of the data will not fit in MCDRAM. The most important consideration when deciding if an application can benefit from MCDRAM is whether or not a problem is memory bandwidth bound. Since MCDRAM only provides a bandwidth improvement this will be the determining factor if a program can benefit from using MCDRAM. When considering the memory bandwidth boundedness of Jaccard it is of course important to consider the density of the problem and the algorithm being used. The KNL machine has 68 cores and each core supports up to four hyperthreads.

There are 3 ways that MCDRAM can be configured, Cache, Flat, and Hybrid. In cache mode the MCDRAM acts as a large shared cache in which most recently accessed data is stored. This has the advantage of now you get some straight out of the box usage of the MCDRAM however it is unlikely to be efficient without some code redesign. In Flat mode the data is explicitly allocated by the user and all data movement onto the MCDRAM is explicit. Hybrid mode is where a portion of the MCDRAM acts as a cache and a portion acts as it does in flat mode. There have been a variety of studies in how to effectively use these different modes. The most important issue remains and that is to ensure the application will benefit from the improved bandwidth on the MCDRAM.

Presuming we can efficiently make use of all of the hyperthreads (272 threads) we have to generate enough accesses to exceed the DDR bandwidth (90Gb/s) meaning each threads would have to generate more then 330MB/s of accesses. Likewise in order to get full bandwidth of the MCDRAM (384Gb/s) would require each thread to access 1.4Gb/s. This suggests perhaps it would be more effective to use the CSR format instead of a conventional linked list data structure. However in the case of this work we use a linked list and leave converting to CSR format to future work.

We develop a chunking method in which MCDRAM is ran in flat mode. We move a portion of data into the MCDRAM, compute on that portion and then move the next portion of the data in to the MCDRAM. This can be represented very similarly to a consumer/producer problem. We generate a task which is a pair of vertices. The producer takes the neighborhoods of the two vertices and stores them on MCDRAM. The consumer then calculates the Jaccard for this particular pair of vertices. Unfortunately MCDRAM on KNL does not support direct memory access (DMA) which often results in data transfer taking up a considerable amount of compute resources.

We demonstrate the performance of our algorithms on two different RMAT graphs in figure 1.4. The unfortunate observation we tend to observe is that we get little to no performance impact from using the MCDRAM. This could be due to a couple of issues: 1) the problem is not limited by bandwidth 2) The data structure and code doesn't allow for high throughput concurrent data accesses. In extremely sparse problems it is likely that problem is simply not bandwidth bound. However as the problem gets more and more dense we should expect to see improved performance from MCDRAM if we are allowing many memory accesses to be issued using instruction level parallelism.

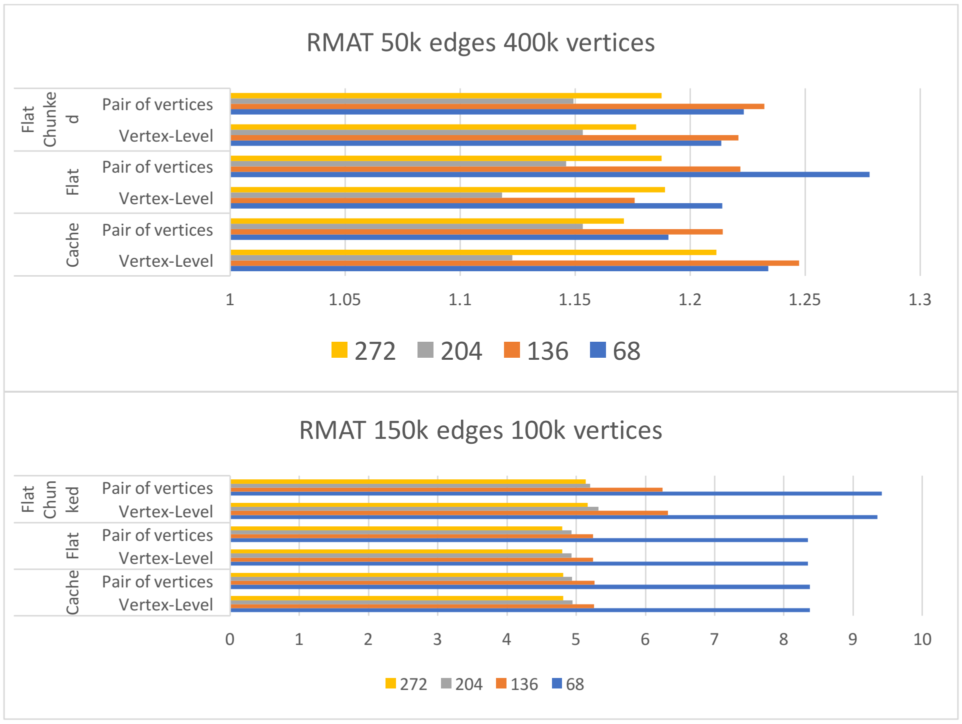

Figure 1.4: Sequential result of Jaccard computations. As we increase the number of threads

1.7 Next Steps

The next step is to provide improved techniques to compute Jaccard while maintaining a higher memory concurrency then the current code provides. This problem takes careful consideration of time and space complexity and how these two will affect the other. It is also of great interest to perform a comparison in which the Graph-BLAS techniques are adapted to perform matrix matrix multiply. It is also of great interest of us to develop a MPI based scaling code which can compute Jaccard coefficients at a much larger scale.

Bibliography

[1] Brian Ball, Brian Karrer, and Mark EJ Newman. Efficient and principled method for detecting communities in networks. *Physical Review E*, 84(3):036103, 2011.

[2] Jacob Bank and Benjamin Cole. Calculating the jaccard similarity coefficient with map reduce for entity pairs in wikipedia. *Wikipedia Similarity Team*, pages 1–18, 2008.

[3] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[4] Peter M Kogge. Jaccard coefficients as a potential graph benchmark. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 921–928. IEEE, 2016.

[5] Chayant Tantipathananandh, Tanya Berger-Wolf, and David Kempe. A framework for community identification in dynamic social networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 717–726, New York, NY, USA, 2007. ACM.