Maintaining Connectivity in Parallel Graph Partitioning

Christopher I. Jones
jonesc10@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY

Ian Bogle
boglei@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY

George M. Slota
slotag@rpi.edu
Rensselaer Polytechnic Institute
Troy, NY

1 INTRODUCTION

Graph partitioning is the process of creating vertex-disjoint sets on a graph that fit some optimization objective and set of balance constraints. Generally, the sum of vertex weights are balanced across all parts and the size of the edge cut is minimized. Running the graph partitioning process in parallel is highly desired so that we can quickly process larger graphs, on the scale of billions of vertices or higher. Our partitioner PuLP [1] uses a label-propagation approach with vertex-centric computations in parallel to scalable solve the multi-weight partitioning problem. In this work, we introduce an additional constraint to the partitioning problem, which is maintaining connectivity of the parts in a partition. In other words, we aim to reduce the number of connected components in the final solution, where in the ideal case each part is fully connected. This effort investigates various methods for retaining part connectivity during partitioning with PuLP. Results will eventually be integrated into the PuLP and XtraPuLP libraries [2].

2 METHODS

The work in this project is implemented into the existing PuLP [1] algorithm. PuLP has two main iterations that it repeatedly performs on every vertex in the graph. A *balance* iteration focuses on balancing the weights of each part and a *refine* iteration improves the cut without hurting the current balance. For a given balance vertex, PuLP considers the *gain* associated with this vertex joining a part P:

$$gain(P) = n_P \cdot w_P \tag{1}$$

Where n_P is the number of neighbors of this vertex that are in part P, and w_P is how underweight P is (i.e. how much more vertex weight is needed for P to achieve its target constraint). A vertex will join the neighboring part that has the highest *gain* value. We investigated various different methods throughout the PuLP algorithm for maintaining connectivity to the original algorithm. In the following discussion, assume for simplicity that the input graphs are fully connected and components are defined by their current part boundaries.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 Merging Small Components

A basic method we employed was simply merging all smaller components until the number of components equals the number of parts. This operation can be carried out between processing iterations of the algorithm or as a final step. To perform this method, we consider the largest component as defining a given part and merge neighboring small parts into it. We use a breadth-first search variant to find the available neighbors of small parts, and perform a weighted merge based on collected metrics such as how many neighboring edges, vertices, and components each part currently has.

2.2 Preventing Dis-connectivity

Instead of re-assigning large numbers of vertices at once, which can upset part balance, we also consider restricting which vertices are allowed to move during the PuLP algorithm. This is to prevent dis-connectivity of parts from occurring in the first places. We consider three variants: 1.) We only consider allowing leaves defined by a BFS in a given part to move their assignments; 2.) We allow only vertices with C or fewer successors in a BFS to move; and 3.) We explicitly compute part biconnectivity and prevent articulation vertices from moving. The intuition behind these methods is that by only re-assigning the leaves of a tree or non-articulation points, we don't disconnect more than a single vertex from a part in a single re-assignment. By preventing a vertex in a rooted tree with C lower-level successors, we prevent at most $C + 1$ vertices from being disconnected in a single assignment. However, these methods tended to be rather heavyweight computationally or far too restrictive to allow our weighted constraints to be satisfied.

2.3 Additional Metrics

The goal of this method is to see if we can introduce another metric into the objective equations, such as *gain* (Equation 1), that will improve connectivity. Previous methods involve doing operations outside of these equations; we want to modify such equations with new metrics in order to keep the algorithm vertex-centric. One potential metric that we introduce is *average number of children* (ANC), which is again based on the same BFS trees generated in the *Leaves Only* method (but is easier to track). We denote ANC as a_P:

$$a_P = \sum_{\substack{u \in N(v) \\ u \in P}} \frac{children(u)}{d(u)} \tag{2}$$

Where $N(v)$ is the neighborhood of v, $children(u)$ is the number of children u has in the component's BFS tree, and $d(u)$ is the degree of u. For a given vertex v, and for each neighbor u of v, the ANC metric finds the ratio of the number of children of u to the degree of u. Then these ratios are summed up per part, P, to give a_P. The intuition is that a vertex v would be better off joining a part that has a higher ANC value, as that means the portion of the part that is neighboring v is relatively well connected. We introduce the ANC metric in two main ways into the *gain* equation:

$$gain(P) = (n_P + a_P) \cdot w_P \tag{3}$$

$$gain(P) = n_P \cdot w_P / a_P \tag{4}$$

Note that Equation (4) goes against intuition, by producing a higher gain value when the ANC value is small.

3 RESULTS

Gathering experimental results was done with five test graphs with up to about 5 million edges each: a scientific mesh, a road network, an ASIC circuitry network, a web-crawl, and a social network. To simplify the solutions of our testing, we only compute on the giant component of each of these graphs.

3.1 Merging Small Components

We considered our two approaches. First, we merged all small components at the end of each iteration; however, this generally upset our balance constraints. Second, we simply ran the merging as a final step of the original algorithm with results shown in Table I. We note that a single merge step generally retained balance constraints for most networks and could therefore be used as a final refinement step with other methods.

Table 1: Merging as a Final Step

	Vertex Overweight	Edge Cut
Scientific Mesh	1.22	2,350
Social Network	1.31	241K
Web-Crawl	1.18	94.5K
Road Network	1.06	3,710
ASIC Circuit	1.04	16.6K

3.2 Preventing Dis-connectivity

Generally, restricting only leaves or articulation vertices to change parts prevented us from achieving part balance. We show in Table 2 a test with multiple values of C, which is the threshold at which any vertex with more than C successors is not allowed to move parts. Note that the normal PuLP algorithm is equivalent to ($C = \infty$). In many tests, when only allowing the leaves to move ($C = 0$), the algorithm would stall or oscillate between moving just a few vertices back and forth.

3.3 Additional Metrics

We tested the addition of the ANC metric, a_P, into the gain equation, as defined by Equations (3) and (4). To update the a_P metric, BFS trees are recalculated at the start of each outer iteration. For all test graphs, the use of Equation (3) gave a small improvement in the number of components, without disrupting vertex balancing or the edge cut. Using Equation (4), we saw in most cases the number of components increased significantly.

Table 2: Leaves Only Method on the Web-Crawl Graph

C	Vertex Overweight	Edge Cut	Small Comp. Count
0	1.5-2.5	>1,000K	0
1	1.1-1.2	800K	4,000
2	$\leq$ 1.1	650K	5,000
∞	<1.1	150K	5,500

Table 3: Introduction of ANC Metric on a Web-Crawl Graph

	Vertex	Edge	Small Components	
	Overweight	Cut	Count	Size
Normal	1.039	125.4K	6,100	22.17
Equation (3)	1.040	125.5K	5,800	22.53
Equation (4)	1.057	434.4K	14,400	13.94
Add Merging as a Final Step:				
Normal	1.182	94.5K	0	N/A
Equation (3)	1.174	95.3K	0	N/A
Equation (4)	1.081	356K	0	N/A

In some cases, as displayed in Table IV, Equation (4) produces many components that have a small average size. When we add in the small component merging method as a final step, we saw relatively good results in the web-crawl graph. For the other test graphs, adding the merging step did not produce similar or more definitive results.

4 APPLICATION: REDISTRICTING

We demonstrate our methods by translating the political redistricting problem to graph partitioning. We define *census blocks* as vertices, *districts* as parts, and shared borders between blocks as edges. Previous work on this problem includes PEAR [3], a parallel evolutionary algorithm. Our goal is to optimize *compactness* (boundary over perimeter) while ensuring *competitiveness* (equalizing demographics) and *connectedness*. We apply PuLP to North Carolina, using population demographics as vertex weights and border lengths as edge weights. We show our results in Figure 3. We achieve our balance constraints with low error and low time cost while producing connected, compact, and competitive districts.

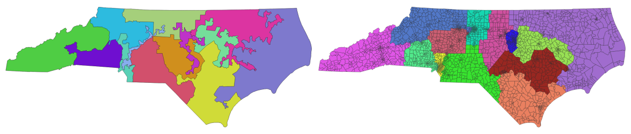

Figure 1: The state of North Carolina's congressional districts from 2013-2016 (left) and a redistricting of North Carolina using PuLP with population vertex weights, border length edge weights, and merging as a final step (right).

ACKNOWLEDGEMENTS

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Rensselaer Polytechnic Institute and Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

REFERENCES

[1] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam, PuLP: Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks, in the Proceedings of the 2nd IEEE Conference on Big Data (BigData 2014).
[2] https://github.com/HPCGraphAnalysis/PuLP/
[3] Y. Y. Liu, W. K. T. Cho, S. Wang, PEAR: a massively parallel evolutionary computation approach for political redistricting optimization and analysis, in Swarm and Evolutionary Computation, 30, 78-92, 2016.