Subtractive Vertex Magic Labelings and Parallel Computing

Jordan Turley

January 24, 2019

1 Introduction

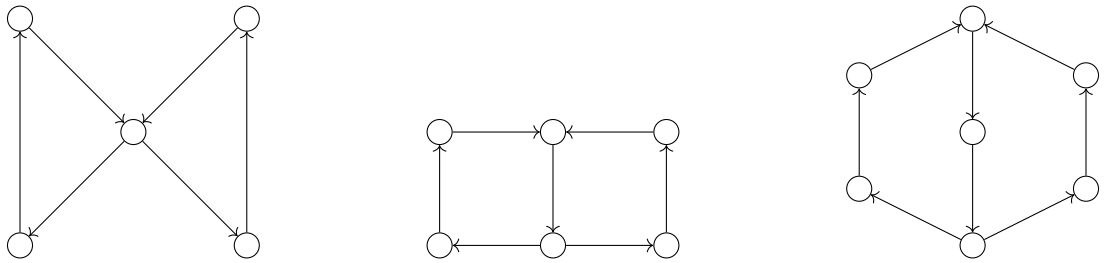
Graph theory is a branch of mathematics which deals with relating data through vertices and edges. There are applications in almost all aspects of life, like modeling roadways, friendships on social networks, and more. Several problems can be translated into graph theory problems. Graph labeling is a subfield of graph theory where integers are assigned to the vertices, edges, or both, such that some constraint is satisfied. Hundreds of labelings exist, such as prime, graceful, and magic. In this paper we focus on subtractive vertex magic labelings of directed graphs [1]. These labelings are easy to check but hard to find by hand. However, labelings are easy to generate and test with a computer. We present single-threaded and multi-threaded programs which generate and test subtractive vertex magic labelings for a given graph in hopes of finding a pattern that we can generalize. We also analyze the efficiency of these programs to see how performance scales as the number of processors increases.

2 Subtractive Vertex Magic Labelings

For a directed graph $G = (V, E)$, a subtractive vertex magic labeling is a bijection $\lambda : V \cup E \to \{1, 2, ...|V| + |E|\}$ such that each vertex has the same magic constant. For a vertex $x \in V$, the magic constant of x is equal to $\lambda(x) + \Sigma\lambda(yx) - \Sigma\lambda(xy)$ for all adjacent vertices y, where yx is an edge directed into x and xy is an edge directed out of x.

In other words, for a graph G with n vertices and m edges, a subtractive vertex magic labeling is an assignment of the integers $\{1, 2, ..., m + n\}$ to the vertices and edges such that the magic constant is the same for every vertex, where the magic constant is equal to the vertex value plus the sum of the edge values directed in minus the sum of the edge values directed out.

We can use any graph we desire, but we decided to focus on a specific type of graph where two identical directed cycles are joined in the middle by a certain number of vertices, denoted $C^2_{m,n}$ where m is the cycle size and n is the number of vertices joining the two cycles, for $n < m$. Some examples are shown in Figure 1.



(a) $C^2_{3,1}$ graph (b) $C^2_{4,2}$ graph (c) $C^2_{5,3}$ graph

Figure 1: Examples of graphs

3 Methods

To evaluate the performance of the program, we used a single-threaded sequential version and a multi-threaded OpenMP version of the program running on a powerful server. We used the $C_{4,1}^2$ graph, show in Figure 2.

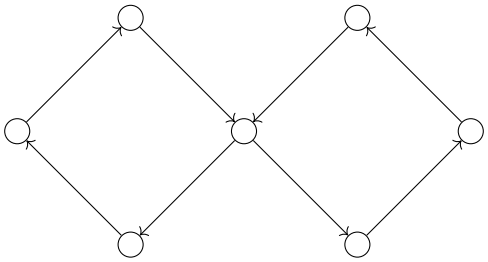


Figure 2: $C_{4,1}^2$ graph

3.1 Software

Given a graph $G = (V, E)$ with $|V| = m$ and $|E| = n$, in order to find subtractive vertex magic labelings, we must generate all permutations of the set $\{1, 2, ..., m+n\}$, as any of the vertices and edges can be labeled with any integer from this set. For each permutation, we assign the first m numbers in the sequence to the vertices and the remaining n numbers to the edges of the graph. Then, we test if every vertex has the same magic constant. If any one vertex has a magic constant that is different from another, we know that the labeling is not valid and we can move on to the next. When we find a valid labeling, we record the index and the magic number to avoid any I/O that could cause unwanted overhead. After we have checked all possible permutations, we regenerate each permutation using the factoradic number [2] of the index and output the permutations to a file.

It should be noted that this procedure can very easily be applied to other graph labelings. To generate potential labelings you simply need to generate all permutations and apply them to the vertices, edges, or both and test the labeling. It would be very easy to test for prime, graceful, or other types of magic labelings with the code that has already been written. This may be of value in the future as there are other labelings that we are interested in.

3.2 Hardware

To evaluate the performance of the program, we used an x64 server which contains four AMD Opteron 6378 processors. Each processor has 16 2.4 GHz CPU cores for a total of 64 CPU cores. The server also has 64 GB of RAM, but the program used a very small amount of memory, so this was not of concern.

4 Performance

We see the runtime, speedup, and efficiency in Figures 3, 4, and 5. For eight or fewer threads, we achieve linear or even slightly super-linear speedup. However, as we approach 64 threads, the runtime does decrease significantly, but the speedup is not quite linear. There are several reasons that this could occur. The distribution of permutations that result in valid labelings is seemingly random. There may be some pattern but we have not discovered it yet. As a result, some threads may have to do more work than others. When the work is broken into 64 pieces, one thread could be doing significantly more than another, which could result in a slightly longer runtime and non-linear speedup. There is also more overhead in general when going up to 64 threads. Even though there are no critical sections in the code, this could still present an issue. There could also be a number of other reasons for this non-linear speedup that we have not yet discovered.

When looking at the runtime we see why a parallel version of this program is necessary. For a relatively simple graph with only seven vertices and eight edges, it takes almost a day to test all potential labelings. There are two other graphs that have the same number of potential labelings, so to find all valid labelings would take half of a week. In the grand scheme of things this is not that long, but for an undergraduate research course, time is somewhat limited and it is of great value to get results as quickly as possible. With the multi-threaded version and the 64 core server, results could be found for all of the graphs that we tested in a single class period rather than over the course of a week.

The next biggest graph has 17! potential labelings, so theoretically a sequential algorithm would take more than 226 days, which is not feasible for the type of research we are conducting. A parallel algorithm with 64 threads would theoretically take only five days. With the help of a more powerful computer or a supercomputer, this time could be further decreased.

Even if an individual did not have access to a system with many cores like we did, the multi-threaded version could still be useful. Most modern home computers or laptops have at least a quad-core processor. With four threads, results for this particular graph could be found in less than five hours, and results for all three graphs with the same number of potential labelings could be found in under 24 hours.

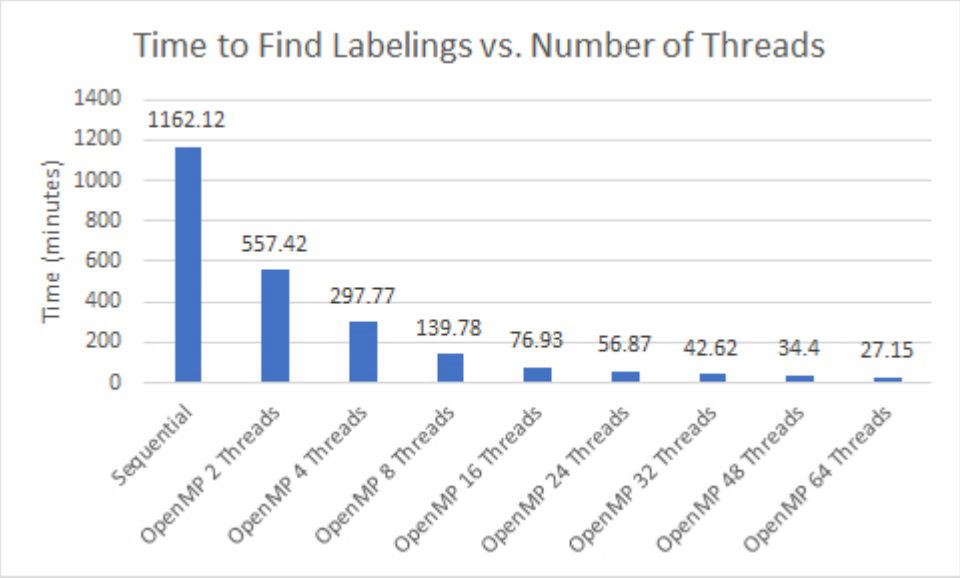


Figure 3: Runtime

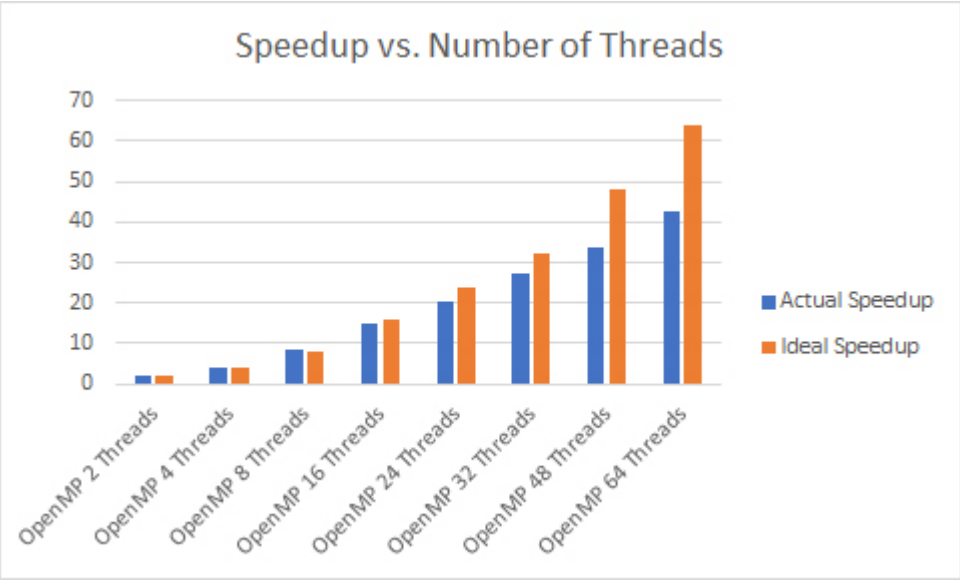


Figure 4: Speedup

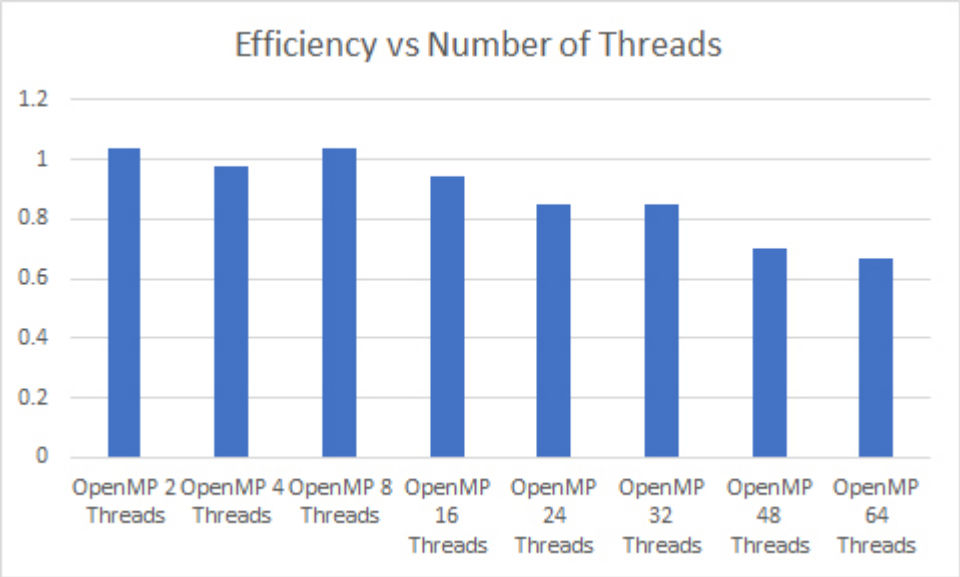


Figure 5: Efficiency

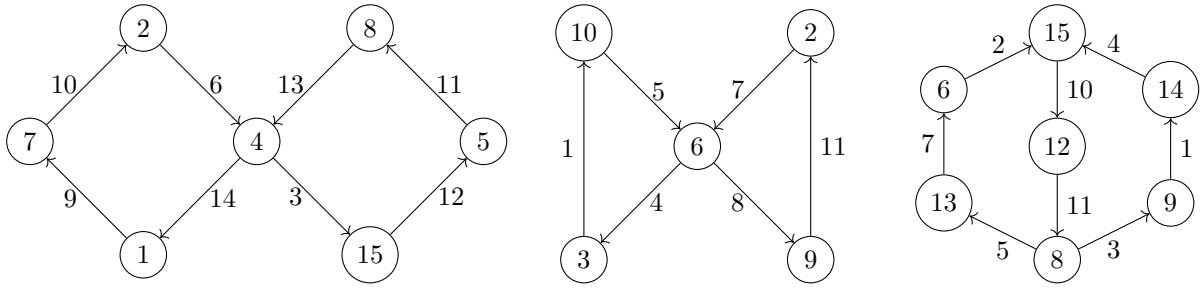
5 Results

With this problem, we were more concerned with the resulting labelings found for these graphs than we were with the runtime and performance. We were able to get results for graphs we were interested in fairly quickly which was in some ways more important than the performance analysis of the algorithm, since once we have the results, there is no need to run the program again for the same graph. In Figure 6, we see the graphs that were tested, the total number of potential labelings, the number of valid labelings out of these, and the runtime. The results for all of these graphs could be found in less than two hours. In Figure 7, we see a few examples of valid labelings that were found by the program. It is valuable to know that these labelings are valid; however, the graphs that we were able to test were very limited. There are an infinite number of graphs of this style. We hope to use these results to find some sort of pattern as the graphs get

larger that we can generalize. Once we move up to a graph that has, for example, 30! or more permutations to test, it is infeasible for even a supercomputer to find labelings for a graphs.

Graph	Total Permutations	Valid Labelings	Runtime (seconds)
$C^2_{3,2}$	9!	16	< 1
$C^2_{3,1}$	11!	192	< 1
$C^2_{4,3}$	11!	132	< 1
$C^2_{4,2}$	13!	332	7
$C^2_{4,1}$	15!	1720	1601
$C^2_{5,4}$	13!	232	7
$C^2_{5,3}$	15!	1166	1588
$C^2_{6,5}$	15!	950	1648

Figure 6: Tested Graphs and Results



(a) $C^2_{4,1}$ with k = 6 (b) $C^2_{3,1}$ with k = 6 (c) $C^2_{5,3}$ with k = 11

Figure 7: Examples of Subtractive Vertex Magic Labelings Found

6 Conclusions and Future Work

The multi-threaded program proved to be significantly faster and as a result more useful than the single-threaded version. The program has been used to find subtractive vertex magic labelings for several graphs and will be used for more in the future. We hope to be able to move to a supercomputer to be able to run the program for larger graphs in a reasonable amount of time.

We hope to use these results to find a pattern between the labelings that can be generalized for any graph of this type rather than only these small specific cases. We also plan to use this for more families of graphs to find subtractive vertex magic labelings which we will also try to generalize.

After implementing the multi-threaded version in OpenMP, we also implemented a version in MPI. The results it produced for which labelings are valid were identical to the results produced by the sequential and OpenMP, so we were confident that the results were correct. When running on a laptop, the performance was slightly better than the OpenMP version. We planned to use two identical 64 core servers to decrease the runtime even further for the larger graphs. However, we observed some strange behavior. OpenMP with only 64 threads consistently outperformed MPI with 128 processes. We wrote test programs for OpenMP and MPI which simply generated the permutations and did nothing else and tested with 64 threads or processes. All OpenMP processes finished within seconds of each other, but some MPI processes finished minutes before others even though each process was doing the exact same amount of work. We are not sure if there was a problem in the code or with the configuration of the systems. In the future we would like to

utilize MPI as it should theoretically decrease runtime even more and would allow us to move to something like a supercomputer for larger graphs.

Because the type of graphs we are focusing on are symmetric, the labelings come in pairs. If you have one labeling, it is very easy to find another by simply flipping the labeling, as shown in Figure 8. So, the permutations that are found to be valid come in pairs. It would be very valuable if we were able to devise a way to eliminate duplicate labelings. Keeping track of which permutations have and haven't been checked and checking new permutations could cause more overhead than it is worth, but if it could be done in an efficient way, it could theoretically cut the runtime of our program in half.

As mentioned before, the code can be slightly modified to test for other labelings. It would not be difficult to test for prime, graceful, or any other type of graph labeling. It may be of interest to write versions of the algorithm which can find each of these types of labelings as we are also interested in these types of labelings and finding patterns between them.

Figure 8: The two labelings are reverses of each other and are essentially equivalent

References

[1] C. A. Barone. *Magic labelings of directed graphs.* PhD thesis, University of Victoria, 2008.

[2] Irene. Factorial base numbers and permutations. https://generalabstractnonsense.com/2012/07/Factorial-base-numbers-and-permutations/.