1 Introduction

Throughout the semester, we've explored many different algorithms for graph analysis, and graph clustering specifically. However, most of the implementations of these algorithms that we've employed throughout the semester have been sequential implementations in imprecise, higher-level languages. From a hardware perspective, there was performance left on the table, as most modern laptops sport dual or quad-core processors, and desktops with six, eight, and even greater numbers of cores are becoming increasingly common.

The goal of this project is to identify a model for parallel computation that readily suits graph analysis algorithms and examine the performance of different graph algorithms. The performance of different algorithms with and without in-memory graph compression is of specific interest for this project, as the machine being used for benchmarking is running a 16-core 32-thread AMD 2950X with 32 GiB of DDR4 at 2666MHz.

Reimplementing the algorithms discussed below is of secondary interest, and, when possible, the benchmarks are conducted with existing, well-tested implementations of the given algorithms.

2 Algorithmic Challenges

Graph algorithms present a unique challenge in developing work-efficient parallel implementation due to their agonizingly sequential structure. The typical flow for a graph clustering algorithm is to first calculate some observation on a graph, then cut the graph based on that observation, until there exists no more optimal cuts to make.

The necessary sequential ordering of each cut means that there is a strict upper limit on the amount of parallelism achievable. That is to say that, rather than taking a parallel approach to the entire problem, one must try to extract as much parallelism as possible between performing cuts on the graph.

2.1 Local Clustering

An interesting subset of algorithms that I wound up doing most of my testing and experiments with were local graph clustering algorithms. To a computer engineer like myself, *local* algorithms typically means that they're written for a shared-memory computational model, rather than distributed memory. However, in the context of graph analysis, *local* clustering refers to a family of clustering algorithms that, rather than finding clusters for all points in the graph, simply find a cluster around a single point.

Local clusters, namely thanks to their smaller sample set, are more easily found in a highly-parallel fashion, because the only cut performed comes at the end of the algorithm. Their working set is also a small fraction of the total size of the graph, which means that investigating the cluster around specific points of interest can be done much more quickly.

2.2 Data Challenges

One final, more subtle aspect that makes graph clustering a challenging problem to parallelize is the inherent work-imbalance implicit in the data set. Because each node has different characteristics in a graph, algorithms runtimes differ depending on which node they're run on. While this characteristic is of negligible importance for sequential graph algorithms, the work-imbalance across a data set makes decomposing the problem much more challenging.

Because work does not necessarily map evenly over nodes, decomposition becomes more challenging than simple segmenting a set of nodes into equal-sized chunks for each parallel worker to process. The naive, segmenting approach to parallel processing may be functional, in this case, but it provides very poor guarantees for resource utilization and computational span. Though work is often correlated with the edges in a set, clustering is a node-oriented computation, and thus dependent on the work associated with each node.

3 Ligra and Parallel Computation Models

Ligra (`https://github.com/jshun/ligra`) is a parallel graph-processing framework written in C++ which uses CilkPlus. Ligra's model for parallelism is based on mapping computation over subsets of the vertices in a graph, which one can think of as a moving frontier. This 'frontier-based' approach to parallelizing graph clustering, thanks to the natural similarity to one step in a global clustering algorithm to a single map operation. Ligra also offers a filtering model, wherein the frontier is reduced by some metric, which is also well-suited to the model of a graph clustering algorithm.

One very important part of Ligra's performance is the fact that it uses CilkPlus, which is a work-stealing runtime system. In Cilk's parallel model, parellism can be expressed as function calls that can execute in parallel, points where the parallel function calls need to join, and parallel for loops, where each iteration of the loop is processed in parallel. Cilk's terminology for this model is `spawn`, `sync`, and `par_for`.

Because Cilk is work-stealing, though, once a parallel worker runs out of work that it can do on its own, the worker will attempt to randomly steal work from other workers. Because a worker is either working or attempting to steal, Cilk is able to guarantee real-world span that is very close to optimal, even in cases where there is no readily apparent way to decompose the work into equal amounts for each parallel worker.

4 Algorithm Discussion

Betweenness clustering (see Algorithm 1) is a great example of this structural limitation for graph-clustering algorithms.

Algorithm 1: Betweenness-Based Clustering

```
input  : A graph $G$ and a desired number of clusters $k$
output: A set of subgraphs
Function gn_betweenness_clustering (G, k)
    while $number\_connected\_components(G) > k$ do
        if $size(G) = 0$ then
            break;
        end
        $edge\_betweenness \leftarrow edge\_betweenness\_centrality(G)$;
        $remove\_edge(G, max(edge\_betweenness))$;
    end
    return $connected\_components(G)$;
```

Each edge-removal cut is dependent on the full completion of the edge-betweenness computation that preceeds it, which means that the only place to extrace parallelism in the algorithm is in that edge-betweenness calculation.

Algorithm 2: Recusive Modularity Maximization

```
input  : A graph $G$ and a maximum number of clusters $k$
output: A set of subgraphs
Function mm_recurse (G, k, $q_0$, q)
    $B \leftarrow modularity\_matrix(G)$;
    $u \leftarrow power\_iteration(B)$;
    if $k \leq 1$ or $q_0 > q$ then
        return G;
    end
    $S_{left} \leftarrow \{v | v \in G \cup u[v] \geq 0\}$;
    $S_{right} \leftarrow \{v | v \in G \cup u[v] < 0\}$;
    $k \leftarrow k - 1$;
    if $k >= 1$ then
        return $\{S_{left}, S_{right}\}$;
    end
    $left\_graphs \leftarrow$ mm_recurse$(S_{left}, k, q_0, modularity(G, S_{left}))$;
    $k \leftarrow k - len(left\_graphs)$;
    if $k == 0$ then
        return $\{left\_graphs, S_{right}\}$;
    else
        $right\_graphs \leftarrow$ mm_recurse$(S_{right}, k, q_0, modularity(G, S_{right}))$;
        return $\{left\_graphs, right\_graphs\}$;
    end
Function mm_clustering (G, k)
    return mm_recurse $(G, k, 0, 0)$;
```

Modularity maximization (see Algorithm 2) is a more complex algorithim that offers much more opportunities for parallel speedup in each cut step, because each cut step removes significantly more edges from the graph at once. Each recursive step of modularity maximization can be computed independently, which is beneficial to the spacial locality of each parallel unit of work.

PageRank-Nibble is a local clustering algorithm which is presented by J. Shun et. al. in *Parallel Local Graph Clustering.* The pseudocode is reproduced in Algorithm 3. PageRank-Nibble is, notably, only a function to compute a vector of node-weight pairs. Shun et. al. propose a parallelized sweep-cup algorithm to make the final pass over the points and decide on the final cluster.

Algorithm 3: PageRank-Nibble

```
input : A graph $G$, a seed vertex $v$, an error margin $\epsilon$, and a teleport
        probability $\alpha$
output: A set of subgraphs
$p \leftarrow \{\}$;
$r \leftarrow \{\}$;
$r' \leftarrow \{\}$;
Function UpdateNeighbor (s, d)
    $r'[d] \leftarrow r'[d] + (1-\alpha)r[s]/(2d(s))$;
Function UpdateSelf (v)
    $p[v] \leftarrow p[v] + \alpha r[v]$;
    $r'[v] \leftarrow (1-\alpha)r[v]/2$;
Function PR-Nibble (G, k)
    $r \leftarrow (x, 1)$;
    $Frontier \leftarrow \{x\}$;
    while $Frontier > 0$ do
        $vertexMap(Frontier, \texttt{UpdateSelf})$;
        $edgeMap(G, Frontier, \texttt{UpdateSelf})$;
        $r \leftarrow r'$;
        $Frontier \leftarrow \{v | v \in G \cap r[v] \geq d(v)\epsilon\}$;
    end
    return p;
```

5 Discussion

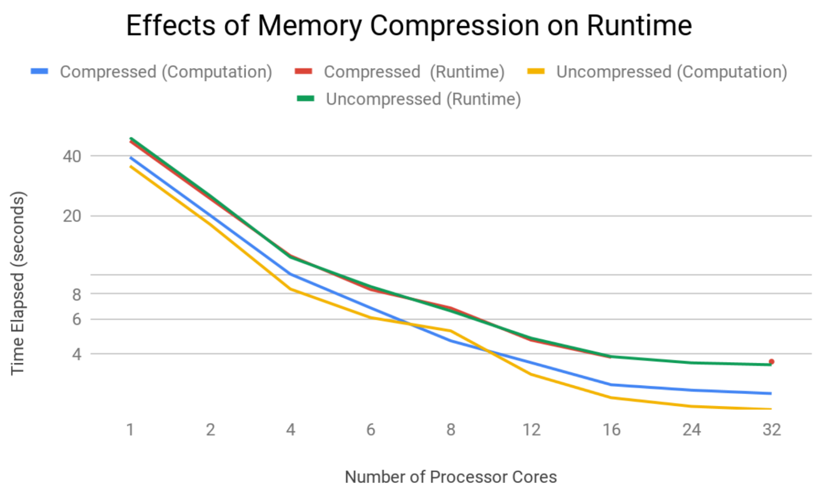
Unfortunately, despite the wonderful interface of Ligra, implementing graph clustering algorithms in C/C++ is a bit of a nightmare. Both C and C++ lack efficient semantics for working with and manipulating sets of data points (similar to those found in higher-level languages, like Python). I didn't anticipate how difficult correctly implementing either modularity-maximization clustering or PageRank-Nibble would be. You can find my partial (very non-functional) attempts at implementing them in `src/{MM_Clustering.C,PR-Nibble.C}`.

The data taken below is from J. Shun's implementation of PageRank-Nibble, which can be found in Ligra's `localAlg` applications on GitHub. I burned so much time trying to re-implement these algorithms that I had very little time to analyze the data that I collected. In addition to the graph comparing the performance of compressed and uncompressed representations of the graph, I also have data on the performance of PageRank-Nibble versus the traditional Nibble algorithm, but I didn't have the time to analyze it before submitting the report.

The data was run on the orkut network from SNAP (`https://snap.stanford.edu/data/com-Orkut.html`).

I was, honestly, fairly surprised to see how little of an impact in-memory compression made for the runtime of the clustering algorithm. In the past, I had heard that graph analysis algorithms are so memory-bandwidth intensive that in-memory compression could actually accelerate the algorithm by alleviating some of the strain on the memory interface. However, in this case, the compressed graphs are always slightly worse than uncompressed graphs in both total runtime and computational runtime.

Just as expected, PageRank-Nibble demonstrates excellent speedup when scaling from one to 32 processor cores.



6 Sources

`https://people.eecs.berkeley.edu/~jrs/papers/partnotes.pdf`
`http://people.csail.mit.edu/jshun/local.pdf`
`https://github.com/jshun/ligra`
`https://www.cilkplus.org/`