Parallel Network Motif Finding
Michael Schatz, Elliott Cooper-Balis, Adam Bazinet
CMSC 714 High Performance Computing

Abstract
Network motifs are over-represented patterns within a network, and signify the fundamental building blocks of that network. The process of finding network motifs is closely related to the traditional subgraph isomorphism problem in computer science, which finds instances of a particular subgraph in a graph. This problem has been proven NP-complete, and thus even for relatively small subgraphs and graphs, today’s most efficient approaches require a large amount of computation time. Here we present parallel algorithms for network motif finding, including both query parallelization, where different subgraphs are searched for in parallel, and network parallelization, where the original network is partitioned into overlapping regions, and a single subgraph is searched for in parallel among the different regions. The network partitioning algorithm is a novel application of hierarchical clustering and divides the network into strongly connected components, while minimizing the overlap between them. Our results show query parallel network motif finding speeds up the computation time linearly with the number of workers, and allows us to systematically search for larger motifs than ever before. In addition, our novel network partitioning algorithms are effective at partitioning the network, which enables network parallelization to effectively reduce the time required to count instances of a particular subgraph.

1. Introduction
In 2002, Milo et al. spurred interest in the field of network motif analysis [1] by discovering that the same motifs are present in networks that originate from highly diverse sources. A network motif is a pattern (subgraph) of a larger network that occurs at a much higher frequency than is expected by chance. These patterns often have specific functional roles, and form basic building blocks of all complex networks. Milo et al. searched for over-represented motifs in a variety of biological and technological networks, including gene regulation networks in yeast, neural connections within the nematode worm, food webs in different environments, electronic circuit designs, inter-page hyperlinks on the World Wide Web, and connections between routers on the Internet. The specific network motifs discovered in the different networks varied, and the authors hypothesize the particular networks motifs discovered in each network reveal the fundamental properties of that complex network. For example, the feed-forward loop between 3 nodes: A->B, B->C, A->C (configuration 5 in Figure 1), was commonly found in electronic circuit design, as it requires the signal at A to be persistent to activate C via B, and thus dampens noise at A. Surprisingly, though, it was also overrepresented in gene regulation networks, suggesting gene regulation is tolerant to noise using the same mechanism as in an electronic circuit.

Recent work has searched for motifs in networks called protein-protein interaction (PPI) networks, in which nodes represent proteins in an organism, and (undirected) edges connect proteins that interact within an organism. These networks typically have several thousand nodes and tens of thousands of edges, but the exact numbers vary according to the complexity of the organism. PPI networks are publicly available for several model organisms and were built using high throughput laboratory methods that test for molecular interactions between all pairs of proteins. This work has discovered several statistically and biologically important
motifs [2]. For example, signaling pathways within a cell will be present as long chains of connections, while protein complexes will be represented as a clique.

The method Milo et al. used for discovering overrepresented network motifs was to count the number of instances of all possible 3 and 4 node subgraphs in the target network, and then do the same in randomized versions of the target network. Afterwards, they report the motifs that were statistically overrepresented in the target network when compared to the random networks. For example, there are 13 possible 3-node subgraphs to search for in a directed graph (Figure 1). Milo’s algorithm for counting possible motifs was a brute force search of all 3 and 4 node subgraphs in the target networks. This algorithm is so computationally expensive that only very small motifs (n<=4) could be analyzed in a reasonable amount of time, but large motifs are also of great interest.

Figure 1. All possible 3 node directed graphs

In a follow-up paper published in 2007, Grochow and Kellis [3] present a dramatically faster algorithm for network motif discovery. This is the fastest known serial algorithm for network motif discovery. Unlike the Milo algorithm, which counts instances of all possible 3 or 4 node graphs simultaneously, the Grochow and Kellis algorithm takes as input a single query graph and counts the number of instances of it in the network with a very efficient method. To reproduce Milo’s analysis, one would simply run the Grochow and Kellis algorithm on all possible query motifs up to a given size in the target network and in several random networks.

Their algorithm directly addresses the classic subgraph isomorphism problem, which is known to be NP-complete. Even so, their algorithm achieves an exponential speedup over the brute force approach using algorithmic techniques they call subgraph enumeration and symmetry breaking. These optimizations help to avoid rediscovering automorphisms of a previously discovered subgraph, such as the rotation of labels in a clique. Despite the exponential speedup over the brute force approach, the running time for the algorithm is still exponential in the number of nodes in the query graph, and searching all possible query graphs up to 7 nodes requires hours of computation per network. Consequently, there had been no attempt to exhaustively study larger motifs. However, since their algorithm counts instances of a single query motif in the larger network, the authors state their algorithm is amenable to parallelization by having different processing units search for instances of different motifs.

We explored their remark and implemented both query parallelization, where a single processing unit processes a single query subgraph in the full target network as was proposed by the authors, and network parallelization, where the target network is partitioned into overlapping regions which are searched in parallel for a single query subgraph. Query parallelization is embarrassingly parallel, since the different queries are completely independent and require no inter-processor communication. In contrast, network parallelization requires coordination to partition and distribute the network intelligently, and to merge the results without over-counting instances of a subgraph in the overlapping regions. This is especially challenging for the biological networks, which are non-planar and have low effective diameter.

2. Methods
Grochow and Kellis made their source code available under the condition that we did not distribute their code without their permission. The subgraph isomorphism code is approximately 10,000 lines of Java, not including the graph and ancillary libraries. Therefore, we decided to implement our solution in Java given the time available for the project.

We selected Parallel Java [4] for inter-processor communication for our project, since it appears to be the most actively developed message passing library for Java. It supports both OpenMP-style fork-join parallelism for SMP machines, and MPI-style SPMD parallelism for clusters of machines.
Parallel Java is packaged as a single JAR file to be included in the CLASSPATH of the parallel application, and uses ssh to launch processes from a frontend processor. Inter-node communication is handled through a Comm class that provides MPI-style primitives such as send/receive, and size/ rank. Primitive data types are transferred natively, and Java objects can be transferred using serialization techniques.

2.1 Query Parallelization
Query parallelization is useful for performing a high throughput search for network motifs by counting instances of every possible subgraph up to a given size. Every processing unit gets a full copy of the target network, and searches for 1 or more queries. The queries are independent, but the search time per query varies considerably depending on the number of nodes and edges in the query subgraph. Therefore, the total runtime depends on how the queries are scheduled.

To address this, we implemented 2 scheduling policies: naïve and first-fit. Under the naïve policy, the full list of S possible subgraphs is precomputed and each of the P processing units is assigned a block of size S/P subgraphs to evaluate. This scheduler has very low overhead, and the master node simply sends a range of queries to each processing unit. However, this scheduler could have large load imbalance if a given worker gets a block of unusually complicated subgraphs. As an alternative, we also implemented a first-fit scheduler, where the master distributes queries one at a time to workers. This ensures each work is always busy, (except possibly for the last P-1 queries), but has higher overhead since workers must request work after every subgraph.

2.2 Network Parallelization
In contrast to query parallelization, network parallelization is primarily useful for speeding up the search of a single subgraph. It may even be necessary to use network parallelization to make the problem feasible, such as when the target network is too large to fit into main memory on a single processing unit. Even when the entire graph would fit into memory, network parallelization may have significant cache performance benefits from increased locality in the search. Network parallelization, however, also has additional overhead. It has to partition the network into different regions, distribute the partitions and merge the results from each worker. The partitioning can potentially be reused for several query subgraphs, so the time to partition can be amortized over the full execution time. However, the regions of the network are not independent, and a particular instance of a subgraph may be present in the overlap between multiple regions. Therefore each instance of the subgraph must be examined to avoid over-counting the total number of instances.

2.2.1 Network Partitioning
The success of network parallelization depends in large part on how well the search network is divided. There are several algorithms for partitioning a graph into multiple regions called a multi-cut of the graph [5,6,7]. These algorithms typically attempt to balance the number of nodes in each partition, or minimize the number of edges that are cut. However, none of the known algorithms are suitable for subgraph isomorphism, since we also require partitions to overlap by the radius of the query to guarantee that every instance of the query will be found. Otherwise the subgraph isomorphism algorithm would have to be modified extensively to consider and merge partial matches, which is well beyond the scope of our project.

We implemented a variation of hierarchical clustering [8] for partitioning the target network for a query with radius r. The radius of a graph is the minimum distance to reach every node in the graph from the “center node” of the graph. For example, for a linear chain of 7 nodes, the maximal distance between any two nodes, the diameter of the graph, is 6, from the leftmost to the rightmost, but the radius is only 3, from the center to either the leftmost or rightmost. For a circular ring of 7 nodes, both the diameter and radius are 3. For a clique of any size, both the diameter and radius are 1.

The partitioning algorithm works as follows. First, the connected components of the target network are extracted and processed separately, since a query subgraph cannot span multiple disconnected components. Then, for each node n of a connected component of size N > MIN_COMPONENT_SIZE, the algorithm first creates a partition centered at n that includes all nodes that are within r hops from n. This creates N overlapping subgraphs that are
highly redundant, but are guaranteed to fully contain every query of radius $r$ (Figure 2). This is easily demonstrated by considering a connected component $W$, and a subgraph $Q$ with radius $r$, i.e. from some node $n_c$ in $Q$, every other node in $Q$ is reachable within $r$ hops. Since the partitioning algorithm creates a partition centered at every node containing every node within $r$ hops, we are guaranteed to consider every center node $n_c$ of $Q$ with all nodes within $r$ hops and thus count every instance of $Q$.

After the first phase, if there are more partitions than desired, the partitions are merged together. The number of partitions can be controlled directly, or based on a characteristic of the graph, such as the desired size per partition. The algorithm finds the pair of partitions with the highest overlap, defined by the fraction of nodes in common compared to the sum of their sizes (Figure 4). Merging two partitions may make other partitions redundant, so the remaining partitions are checked and redundant partitions are discarded. This merging process is repeated until just the target number of partitions remains. This process of merging partitions with the most overlap is analogous to a traditional hierarchical clustering algorithm, where the most similar data points are merged together into the same cluster. This procedure is thus effective at keeping highly connected regions of the network in the same partition and separating regions that are relatively disjoint, i.e., those which have a low degree of overlap.
merging into larger partitions, but at the cost of having potentially more overlap in the aggregate.

2.2.2 Query Distribution
At initialization, all processing units – master and worker – read both the target graph and query graph files using calls from the supplied graphtools library. With both the master and workers containing the graphs in memory, the issues of sending large pieces of data via message passing are circumvented. Since the partitioning is based on the radius of the query graph, the master node pre-computes all possible partitions based on the various radii of all queries to be performed. The correct partition is fetched and used when searching for a particular query. In this way we avoid re-computing the partitioning for each query graph.

The input to the program is a file containing all query graphs, and each query in the file is sequentially searched for in the target graph. Basic communication between master and workers consists of the master sending compact graph representations to the workers, and the workers sending back the total instances found in their respective chunk of the target graph. For generality, handshaking logic is done using messages containing these same data types: at the beginning and end of a query, the master sends a null graph object to alert the workers, and similarly the workers report their status by sending various negative values –1 for “ready for query” and -2 for “completed final query.”

The target graph is made up of two types of components: those that are large enough to be partitioned into smaller sub-components and those that are not. When searching for a particular query, the master creates a compact graph object of each component and sends it to a worker that is awaiting data. Regardless of whether or not this component has been partitioned into smaller sub-components, the entire component is sent. If the component has been partitioned, a particular sub-component is flagged so that the worker knows that is the one to search. When a worker has completed searching a given component, it reports back an integer value of the number of instances found, using the ownership rule explained below to avoid over-counting. Upon receiving the count, the master updates its running tally for the query subgraph and sends a new component to the now idle worker.

2.2.3 Counting Queries Correctly
Since the partitions of a component overlap, it may be possible for the same instance of a query to be found multiple times in the overlapping region, and thus the number of instances would be over-counted. One solution is to store each instance in a hash table and count only the unique instances. This method has high communication and memory overhead since a query may be present hundreds of thousands of times in a network. Instead we use an ownership rule to determine the partition that should count an instance of a query. As explained above, each worker is sent the query subgraph, and an array containing all of the partitions with one partition flagged to search. For each instance of the query found in its partition, the worker checks the array to see if another partition with lower index could also contain the query. If there is another partition, the instance is not counted; otherwise, that partition owns the instance, and adds it to the count. This distributes the problem of over-counting results to the individual workers and has minimal overhead.

3. Results
Our main focus is to search for motifs in biological networks, specifically the yeast protein-protein interaction network examined in the Grochow and Kellis paper. This network records 2493 interactions between 1379 proteins. The network has 162 connected components, although it is dominated by a few large components, the largest having 778 nodes. The diameter of large connected component is 25, but the effective diameter (the average distance between any 2 nodes) is only 9.4, indicating the network is relatively dense and thus hard to partition. Prior work on this network had systematically explored motifs with 7 or fewer nodes, but we used the nauty [9,10] graph tools to enumerate all possible graphs up to 8 nodes (Figure 5), each with radius between 1 and 4.

Computation was performed using the 64 node CBCB wren cluster. Each node has a dual core 3.2 GHz Xeon processor with 4 GB of RAM, several TB of disk space shared over NFS, and allows direct ssh access. Thus, it was a suitable resource for Parallel Java and for our programs, although we
did have to contend with minimal interference from other people’s Condor jobs.

3.1 Query Parallelization
Generally speaking, we expected first-fit Query Parallelization to do better than naïve Query Parallelization on any significantly large query set. Indeed, this is what Figure 6 shows: first-fit outperforms naïve QP in the “sweet spot”, that is, where message passing overhead is offset by continuous computation. This particular run was for a set of all 6-node queries, and did not run for a particularly long time. No doubt the sweet spot would be larger and more pronounced for even larger query sets, which leads us to believe that first-fit QP is appropriate in the majority of cases.

Next, we conducted exhaustive timings of first-fit QP on all N-node query sets where N went from 2 to 8. For each query set, we varied the number of processors from 1 to 64, as appropriate. The results are presented in Figure 7. Speedup was as expected, generally linear with a small amount of message passing overhead. Using our parallelization technique, we were able to compute counts for all 8-node queries in a reasonable amount of time, thus making exhaustive analysis of this next order of magnitude possible.

3.2 Network Partitioning
The network partitioning algorithms are designed to partition the network at regions with relatively few edges, its bottlenecks, and leave regions with small overlap in separate partitions. The success of the
algorithms depends on the number and placement of bottlenecks in the network. Very regular networks, such as a regular grid of 20x20 nodes, with each node connected to its 8 neighbors, may have no natural bottlenecks. Nevertheless, both the hierarchical partitioning algorithm (HIERARCHICAL), and the variant that merges the partition with minimum size (MIN_SIZE), can partition the grid into 2 roughly equal sized partitions in less than 2 seconds (Figure 8).

Figure 8. Network partitioning of a 20x20 grid for a query of radius 1. The black nodes in the middle are shared by both the top and bottom partitions.

In contrast, biological networks have an irregular “scale-free” structure, such that the degree distribution follows a power law, with most nodes having low degree, but a few nodes having very high degree. In these networks, it may be possible to find true bottlenecks that can be used to partition the network and with small overlap. However, if the bottlenecks are not evenly spaced, the partition sizes may be unbalanced.

We explored the overlap between partitions by computing the percent of total number of instances of a query found in the large component of the yeast PPI network that were duplicates for the different partitioning algorithms. Figure 9 shows the results for linear chains of nodes between 3 and 8 nodes long, each having radius = len/2. For the 2 partitioning algorithms, we set the partitioning algorithms to create partitions of approximately 50 or 100 nodes each, corresponding to 15 or 7 partitions of the 778 node large connected component. As expected, the hierarchical partitioning had very few duplicated instances, since it attempts to minimize the overlap between partitions. In contrast, the minimum size variant often had a large amount of overlap, and as much as 225% of the instances were redundant. For cliques with between 3 and 8 nodes, each with radius 1, there were very few duplicated instances for the different algorithms, except for MIN_SIZE_50, which had between 15% and 54% duplicates (results not shown).

Figure 9. The fraction of duplicated instances for query chains of different lengths.

Figure 10 shows the standard deviation in partition size for queries with radius between 1 and 4. As expected, the minimum size partitioning creates partitions that are relatively uniform in size, and have consistently low standard deviation in size. In contrast, hierarchical partitioning creates partitions with a very large standard deviation in size. This is because it has a tendency to create partitions of the relatively small but densely connected regions on the periphery of the yeast PPI network and leave the bulk of the network in a few larger partitions.

3.3 Network Parallelization

Network parallelization is primarily useful for speeding up the search for a particular query subgraph. We explored the performance of network parallelization by choosing a random sample of eight 7-node queries and recording the computation time while varying the number of workers and partitioning algorithms. The partitioning algorithm
was set to create on average 25, 50 or 100 node partitions, corresponding to 30, 15, or 7 partitions for the large connected component, respectively. If the connected component had fewer nodes than the requested size, it was not partitioned but sent to a worker as a whole.

Figure 10. Standard deviation in partition sizes. Hierarchical partitions vary widely in size, but minimum size variation creates relatively uniform partitions.

Figure 11 shows the results for a serial monolithic search on a 1 processor, and a parallel execution with 4, 8, 16 and 32 workers. The hierarchical partitioning algorithm had relatively uniform runtime regardless of the number of worker threads because the partitioning was only marginally effective in this network (HIERARCHICAL_25 and HIERARCHICAL_50 are not shown, but have similar results). As explained in the previous section the majority of the nodes were placed in a single partition, and this partition dominated the overall runtime. The overall runtime was between 10% to 15% slower than the serial search, suggesting a very high message passing overhead with Parallel Java, since the hierarchical partitioning algorithm has minimal overlap between partitions.

The minimum size partitioning algorithm showed good speedup between 4, 8, 16, and 32 workers. The most effective partitioning was MIN_SIZE_25, which creates approximately the same number of partitions as workers from the large component, and thus had the best load balance. It had nearly linear speedup between 4 and 32 workers. This partitioning also has the highest level of overlap between the partitions, so it takes a high number of workers before the overall runtime is faster than the serial search on the monolithic graph.

4. Conclusions
Network motifs reveal the building blocks of complex networks and can show surprising similarity between networks from diverse sources. Searching for network motifs requires a large amount of independent computation and is thus a good candidate for parallel computing. Our results show that we can dramatically reduce the runtime needed for finding motifs, by searching for occurrences of different query subgraphs in parallel using a technique called query parallelization. This technique has almost perfect linear speedup in the overall computation.

Furthermore, we have shown that we can speed up the computation for a single query using a technique called network parallelization. At the core of this technique is a novel algorithm for decomposing an irregular network into overlapping regions with minimal overlap, inspired by hierarchical clustering. We also presented a second algorithm for partitioning a network that creates partitions of uniform size by merging the smallest partition with the partition it overlaps the most. We demonstrated that network parallelization can have near linear speed up for searching for individual queries when used in conjunction with the minimize size partitioning algorithm.
In the future, we plan to explore larger networks to give the partitioning algorithms more opportunity to partition the network, and to have more work for the individual workers and thus offset the message passing overhead. The yeast PPI graph only has interactions for 1379 proteins, but some organisms have many more proteins. Unfortunately, only a small number of organisms have PPI networks available at this time, and the networks are of generally lower quality. The low effective diameter and relatively few bottlenecks of the yeast PPI network makes it intrinsically difficult to partition, but other networks may have more structure to exploit. We also plan to explore variants of the partitioning algorithms that can both minimize overlap and create uniform sized partitions.

5. References


