2012

A Novel Multithreaded Algorithm for Extracting Maximal Chordal Subgraphs

Mahantesh Halappanavar
University of Nebraska at Omaha

John Feo
University of Nebraska at Omaha

Kathryn Dempsey Cooper
University of Nebraska at Omaha, kdempsey@unomaha.edu

Hesham Ali
University of Nebraska at Omaha, hali@unomaha.edu

Sanjukta Bhowmick
University of Nebraska at Omaha, sbhowmick@unomaha.edu

A Novel Multithreaded Algorithm For Extracting Maximal Chordal Subgraphs

Mahantesh Halappanavar[1], John Feo[1], Kathryn Dempsey[3], Hesham Ali[2], and Sanjukta Bhowmick[2],
E-mail:{mahantesh.halappanavar@pnnl.gov, john.feo@pnnl.gov, kdempsey@unomaha.edu, hali@unomaha.edu and sbhowmick@unomaha.edu}
[1] Pacific Northwest National Laboratory. [2] University of Nebraska at Omaha. [3] University of Nebraska, Medical Center.

***Abstract*—Chordal graphs are triangulated graphs where any cycle larger than three is bisected by a chord. Many combinatorial optimization problems such as computing the size of the maximum clique and the chromatic number are NP-hard on general graphs but have polynomial time solutions on chordal graphs. In this paper, we present a novel multithreaded algorithm to extract a maximal chordal subgraph from a general graph. We develop an iterative approach where each thread can asynchronously update a subset of edges that are dynamically assigned to it per iteration and implement our algorithm on two different multithreaded architectures – Cray XMT, a massively multithreaded platform, and AMD Magny-Cours, a shared memory multicore platform. In addition to the proof of correctness, we present the performance of our algorithm using a testset of synthetical graphs with up to half-a-billion edges and real world networks from gene correlation studies and demonstrate that our algorithm achieves high scalability for all inputs on both types of architectures.**

I. Introduction

Computation of many important combinatorial properties for example, the chromatic number or the size of the maximum clique is NP-hard on general graphs. However, efficient polynomial time algorithms for these same set of problems can be developed on a special type of graph known as the *chordal graph* [1], [2]. A chord is an edge in a graph that connects two non-adjacent vertices in a cycle. Chordal graphs are graphs where any cycle larger than three is bisected by a chord, i.e. the largest unbroken cycle in a chordal graph is a triangle [3]. An alternative approach to solving NP-hard combinatorial optimization problems would therefore be to solve these problems on the maximum chordal subgraph of the larger graph. It has also been shown that extracting chordal subgraphs can be used as a sampling technique for large-scale biological networks [4], [5].

Finding a *maximum* chordal subgraph is an NP-hard problem, but finding a *maximal* chordal subgraph is not [1]. A maximal chordal subgraph cannot be trivially extended into a maximum chordal subgraph by adding new edges to it. Dearing *et al.* provided an efficient serial algorithm for finding a maximal chordal subgraph [1]. In this paper, we present a novel parallel algorithm for extracting maximal chordal subgraphs from large graphs. We retain the key idea of testing whether an edge is chordal from Dearing *et al.* However, our algorithm is targeted for shared-memory multithreaded platforms. Our implementation is based on fine grained parallelism where each vertex identifies the edges that should be retained in the chordal subgraph as per its connections with a selected group of its neighbors. At each iteration the group of neighbors is refined until all the edges are marked as to whether (or not) they belong to a chordal subgraph.

We have implemented our algorithm on (i) a server with four sockets of 12-core AMD Magny-Cours (amounting to a total of 48 threads) and (ii) a massively multithreaded platform, Cray XMT, with 128 processors Our tests on a suite of synthetical as well as real world networks from gene correlation studies show that our algorithm is scalable; although the running time is influenced by the structure and size of a network. From an application perspective, with the exception of the extraction of spanning trees and implementations of breadth first search [6], [7], there exist few multithreaded algorithms for sampling networks. Extracting chordal graphs adds a new level of complexity, in that the traversal pattern is dependent on previously selected vertices. We use the data flow approach to restrict the pattern in which the vertices are selected. To the best of our knowledge, this is the first multithreaded implementation of extracting chordal subgraphs. Our main contributions in this paper are:

1) Design and implementation of a novel multithreaded algorithm to extract maximal chordal subgraphs.
2) A proof of correctness and evaluation of the runtime complexity this algorithm
3) Experimental evaluation on two multithreaded platforms using synthetic and biological networks.

The paper is organized as follows. In Section II we briefly describe the serial algorithm of Dearing *et al.* and an earlier parallel implementation targeted for distributed-memory systems. In Section III we present the iterative parallel algorithm for extracting maximal chordal graphs. In Section IV we describe our experimental setup – platforms and test suite. We present and analyze the experimental results in Section V.

II. Background and Related Work

A graph $G = (V, E)$ is a pair of a set of vertices V and a set of edges E. An edge $e \in E$ is associated with two vertices u and v which are called its endpoints. A vertex u is a neighbor of v if they are joined by an edge. The degree of a vertex is the number of edges incident on it. A walk, of length l, is an alternating sequence of $v_0, e_1, v_1, e_2, \ldots, e_l, v_l$ vertices and edges, such that for $j = 1, ..., l$; v_{j-1} and v_j

are the endpoints of edge e_j. A cycle is a walk that starts from and ends with the same vertex and does not visit any edge twice. We refer you to [3] for more details in graph theoretic terminology.

A *chord* is an edge connecting two non-adjacent vertices in a cycle. A *chordal subgraph* of a general graph $G = (V, E)$ is defined as $G^{'} = (V, E_C)$, where $E_C \subseteq E$ is a set of edges such that any cycle of length longer than three has a chord. A chordal subgraph is *maximum* when the number of edges in E_C is as large as any other maximum chordal subgraph of G. A chordal subgraph is *maximal* when an addition of a new edge to E_C will destroy the chordal property.

A sequential algorithm for extracting a maximal chordal subgraph is given by Dearing *et al.* [1]. This algorithm is based on a modified version of graph traversal. An initial vertex is marked as selected. This vertex and all its associated edges are marked as part of the chordal subgraph. Subsequent steps in the traversal select an yet unmarked vertex that is part of the chordal graph and has the highest number of edges to the partly formed chordal subgraph. Additional edges of this vertex are added to the subgraph if they maintain the chordal property. The algorithm ends once all vertices have been traversed. The algorithmic complexity of this method is $O(|E|\Delta)$, where Δ is the maximum degree of the graph. Since the selection of vertices depends on prior execution, this algorithm is inherently sequential.

A distributed algorithm for extracting *nearly chordal* subgraphs is described in [5], [4]. The graph is first partitioned and distributed across processors. If both the endpoints of an edge lie within the same processor then it is associated with that processor. Otherwise, it is marked as a border edge, indicating that its end points are stored in the memory of two different processors. The maximal chordal subgraphs from partitions that lie completely within a single processor are computed concurrently using the serial algorithm of Dearing *et al*. The edges that are part of these chordal graphs are termed as chordal edges. Then the border edges across two processors are added to the subgraph if they form a triangle with a chordal edge. The border edges can be sent through message passing across processors. Scalability of this communication is proportional to (b^2/Δ), where b is the average number of border edges and Δ is the maximum degree. A faster version of this algorithm has been developed where communication is not necessary [8]. The cost of communication is reflected in that some of the border edges might be duplicated in E_C and these duplicates need to be removed. In both the versions addition of border edges can sometimes cause the inclusion of cycles larger than three.

Since only the border edges can create cycles, an approach to eliminating larger cycles is to copy the subgraph induced by the border edges to a single processor and delete appropriate edges. However, this process in turn can create other cycles, and the cycle elimination process has to be repeated. Therefore, complete elimination of large cycles is challenging for this implementation and in the worst case the algorithm becomes sequential. Given that many networks are hard to partition the distributed algorithm is not suitable for a multithreaded implementation.

Current multithreaded network (graph) analysis research include methods for finding connected components [9], clustering coefficients [10], community detection [11] and distance-1 graph coloring [12].

III. A Parallel Algorithm

We now describe our multithreaded algorithm, illustrated in Algorithm 1, for extracting a maximal chordal subgraph. Every vertex is associated with a unique identification (id) number from 1 to n, where n is the number of vertices in the graph. Each vertex identifies its lowest parent (LP), which is its neighbor with the smallest id and is also smaller than itself. If no lowest parent exists then its lowest parent is set to 0. Each vertex is associated with a set of chordal neighbors – those neighbors that will remain as its neighbors in a maximal chordal subgraph. Initially this list is empty. The initialization is shown in Lines 4 to 10 in Algorithm 1.

A chordal subgraph is gradually built per iteration (of the **while** loop on Line 11), by adding to the set of chordal neighbors of each vertex. We start with an initial queue of vertices that are LP to at least one other vertex in the graph. At every iteration, we check if a vertex, v in the queue is an LP of one of its neighbors, w. If v is an LP of w, then we check if the set of chordal neighbors of w is a subset of the chordal neighbors of v (Line 15). This indicates that, in the current version of the chordal subgraph, all neighbors of vertex w are also connected to the vertex v. Additionally, v and w are also connected. Combining these two observations it follows that v, w and all the chordal neighbors of w form triangular relationships. Therefore, if the subset condition is satisfied then the v is included as a chordal neighbor of the vertex w (Line 16), and the LP of w is set to its next lowest parent (Line 20).

Note that, over the sequence of iterations, every vertex v, except the vertex with the highest id, will be the LP of all its neighbors numbered higher than itself. Once all the neighbors of v have been processed, v will not be added to the queue again. Iterations of the **while** loop terminate when the queue becomes empty. A maximal chordal subgraph is formed by connecting every vertex with its chordal neighbors (E_C). Figure 1 provides a sample execution of Algorithm 1 on a small graph.

Since our parallel implementation is fine-grained, the execution time per iteration is bounded by the time for each vertex to identify its next chordal edges Lines 15 to 17. This process is achieved by comparing the set of neighbors of two connected vertices v and w, and is a function of the maximum number of chordal neighbors of these vertices. If the vertices in the neighbor sets are arranged in increasing order, as in our implementation, then the complexity is linear to the size of the smallest set of chordal neighbors. The

Algorithm 1 Maximal Chordal Subgraph Algorithm. **Input:** A graph G. **Output:** A *maximal* chordal edge set E_C. **Data structures:** A vector LP of size $|V|$ to store the lowest parent of each vertex. A vector C of size $|E|$ to track the chordal set of each vertex.

```
1:  procedure MAX-CHORDAL(G(V, E))
2:      Q_1 ← ∅
3:      Q_2 ← ∅
4:      for all v ∈ V in parallel do
5:          w ← lowest parent of v
6:          if w ≠ ∅ then
7:              LP[v] ← w
8:              if w ∉ Q_1 then
9:                  Q_1 ← Q_1 ∪ {w}
10:         C[v] ← ∅
11:     while Q_1 ≠ ∅ do
12:         for all v ∈ Q_1 in parallel do
13:             for all w ∈ adj[v] do
14:                 if LP[w] = v then
15:                     if C[w] ⊆ C[v] then
16:                         C[w] ← C[w] ∪ {v}
17:                         E_C ← E_C ∪ {e_{v,w}}
18:                         x ← the next lowest parent of w
19:                         if x ≠ ∅ then
20:                             LP[w] ← x
21:                             if x ∉ Q_2 then
22:                                 Q_2 ← Q_2 ∪ {x}
23:         Q_1 ← Q_2
24:         Q_2 ← ∅
25:     return E_C
```

upper bound per iteration would therefore be $O(\Delta)$, with Δ being the maximum degree of the graph. The iterations continue until the vertices have compared their edges with all their LPs. If we assume that the vertex with the highest degree is assigned the highest id, then the maximum number of iterations would be $O(\Delta)$. Therefore an upper bound to Algorithm 1 is $O(\Delta^2)$. Note that this is a conservative upper bound, because it is unlikely that both the number of chordal neighbors, as well as the number of iterations reach their maximum limit. The memory usage is $O(|V|)$ for storing the vertices and their LPs plus the number of chordal neighbors which would be proportional to $O(E_C)$, the number of chordal edges.

Also note that, each vertex determines the chordality of an unique set of edges–the ones that connect it to the associated LP. No edge is checked more than once, and since each vertex processes a different set, the only synchronization required is to store the set of chordal neighbors as an atomic process. However, for densely connected subgraphs such as cliques of size k, our algorithm would have to compare $(k-1)$ LPs, and therefore require $k-1$ steps. Thus, densely connected components might lead to loss of parallel efficiency. We are exploring alternative approaches to address this issue.

Proof of Correctness: We provide the proof of correctness of Algorithm 1 in two parts: (i) a proof that the extracted subgraph is indeed chordal, and (ii) a proof that the extracted subgraph is maximal.

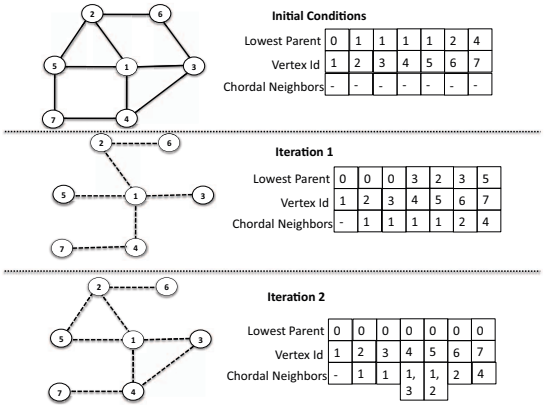


Figure 1. Example of how maximal chordal subgraph can be extracted from a general graph.

Theorem 1: The subgraph extracted by Algorithm 1, $G' = (V, E_C)$, where $E_C \subseteq E$, is chordal.

Proof: We will prove this theorem by induction. Before the execution of the first iteration, the chordal neighbor set of each vertex is empty. Thus, the subgraph consists only of disjointed vertices and is trivially chordal. Let a new edge (v_j, v_i) be added to E_C. We will show that the addition of this edge maintains the chordality conditions.

The edge (v_j, v_i) can be added to E_C only if: (i) v_i and v_j are connected (Line 13) and (ii) C_i is a subset of C_j (Line 15). This implies that every neighbor v_k of v_i will also be a neighbor of v_j according to the chordal subgraph induced by E_C. Note that if $v \in C_w$, then $(v, w) \in E_C$ (Line 17). Therefore (v_k, v_i) and (v_k, v_j) are already in E_C. Since v_j and v_i are connected, the cycle formed by v_i, v_j and v_k would be at most a triangle. Therefore, adding $(v_j v_i)$ to E_C, if permitted, will maintain chordality.

Thus, at each iteration only edges that maintain chordality are added to E_C. It follows that when the algorithm terminates, the graph induced by edges in E_C will be chordal.

Theorem 2: If the subgraph extracted by Algorithm 1, $G^{'} = (V, E_C)$, where $E_C \subseteq E$, is a connected graph, then $G^{'}$ is also a maximal chordal subgraph of $G = (V, E)$.

Proof: We assume that E_C produces a connected subgraph. To prove maximality we will show that if an edge in $E \setminus E_C$ (any non-chordal edge as identified by Algorithm 1) is added to E_C, then the chordal property will no longer hold. We will prove this theorem by contradiction.

Let us assume that we can add an edge $(v_j, v_i) \in \{E \setminus E_C\}$ to E_C after the final iteration of Algorithm 1, and the subgraph so formed still maintains the chordal property. We assume without loss of generality that v_j has a smaller identification number than v_i. Therefore, there existed some iteration p, where v_j was set to be the LP of v_i. Since edge $(v_j,\ v_i)$ is not part of E_C obtained from Algorithm 1, it

means that C_i was not a subset of C_j when the edge (v_j, v_i) was processed. As a consequence, there existed at least one vertex v_k that was in C_i but not in C_j. By the design of Algorithm 1 we derive that all chordal neighbors of a vertex have lower identification numbers than itself (Line 16).

Also note that only LPs of v_i are added to C_i (Line 14). Therefore, v_k was a former LP of v_i and since v_j is the current LP, it follows that the identification number of v_k is less than that of v_j.

If v_k was a neighbor of v_j, then v_k would have already been considered as part of C_j and would have been rejected. Therefore, (v_k, v_j) was not in E_C (Line 17). However, since E_C is connected, there exists at least one path from v_k to v_j in the subgraph induced by E_C. This path would include at least one more vertex v_t. Therefore, adding the edge (v_j, v_i) will create a cycle larger than a triangle $(v_j, v_i, v_k, v_t, v_j)$ that will destroy the chordal property. Thus, our earlier assumption was wrong and the chordal subgraph obtained is indeed maximal.

If E_C has disjoint components, then based on Theorem 2 each component itself is maximal. We can assign an identification number to each component and combine pairs of successively numbered components (lower to higher) by adding *any one* edge from the original graph G whose endpoints lie across the components. Since we are only combining successively numbered components in a low to high order, i.e. (1 and 2) (2 and 3) (3 and 4), but not (4 and 1) or (2 and 4); and with only one edge per component pair, there will be no cycles and the resultant subgraph will still be chordal. Additionally, if the original graph G is itself connected then numbering the vertices in the order they appear in a breadth first search will ensure that at the end of Algorithm 1, E_C will produce a connected subgraph.

IV. Experimental Setup

In this section we provide details of the hardware platforms and the testsets.

A. Hardware Platforms

The first platform is a **Cray XMT** system comprising of 128 Threadstorm (MTA-2) processors that are interconnected via a high bandwidth 3D Torus network (Cray SeaStar2). Each MTA-2 processor consists of 128 streams (hardware threads) and a very-long instruction pipeline with 21 stages. The processor uses a policy of interleaved scheduling – at each cycle, an instruction is chosen from a different thread that is ready for execution. The virtual global address space on XMT is built from physically distributed memory modules of 8 GBytes of DDR-1 memory on each processor. Thus, the total system memory 1 TBytes. A unique feature of XMT is the *hardware hashing* mechanism to make memory accesses uniform. This mechanism maps the data randomly to memory modules in block sizes of 64 Bytes. The average latency of a memory access is 600 cycles with a worst-case latency of about 1000 cycles. A 128-processor system has a sustained bandwidth of 86.4 GB/s. Further details on XMT can be found in [13].

The second platform is AMD Opteron (**Magny-Cours**) based system comprising of 48 cores (4 sockets with 12 core processors) with 256 GB of globally addressable memory. Each 12-core processor is a multi-chip module consisting of two 6-core dies with independent memory controllers. Each processor has three levels of caches: 64 KB of L1 (data), 512 KB of L2, and 12 MB of L3. While L1 and L2 are private to each core, L3 is shared between the six cores of a die. Each socket has 64 GB of memory that is globally addressable by all the four sockets. The sockets are interconnected via AMD HyperTransport-3 technology. Further details on Opteron can be found in [14].

Algorithm 1, like many graph algorithms, is characterized by irregular memory access patterns that results in poor utilization of resources. The two platforms chosen in this study have contrasting architectural features that provide for a thorough evaluation of this algorithm. The principal tool for tolerating latency on the XMT is the use of massive multithreading. The processor is capable of context switching in a single clock cycle, and can therefore tolerate latencies arising from memory stalls and thread synchronizations. In contrast, an Opteron processor relies mostly on memory caches to tolerate latency, and consequently the performance suffers for irregular applications. A faster clock speed and smaller instruction pipeline makes single-thread performance on Opteron relatively faster. But, lack of concurrency results in a large penalty on the XMT. In our experiments we observe situations when the two platforms outperform each other under different circumstances. We present results on Opteron to demonstrate the suitability of our algorithm on modern multicore processors, and on XMT to demonstrate potential suitability for emerging massively multithreaded (manycore) architectures.

B. Test Suites

We perform our experiments on synthetically generated networks as well as datasets obtained from microarray analysis of the hypothalamus of mice. Table 1 gives the structural description of the networks used in our experiments.

Synthetic Networks: Our synthetic networks (graphs) were generated using the R-MAT algorithm [15]. The input parameters for R-MAT include the size of graph in terms of the number of vertices and edges, and a set of four probabilities that sum to one. We specified the number (SCALE) of vertices as powers of two and set the number of edges to eight times the number of vertices. Depending on the probabilities specified, R-MAT can generate graphs with a wide range of characteristics.

For our experiments we generated the following three types of networks; RMAT-ER (with probabilities of $\{0.25, 0.25,\ 0.25, 0.25\}$) which belongs to the class of Erdős-Rényi random graphs and exhibits normal degree distribution; RMAT-G $\{0.45, 0.15,\ 0.15,\ 0.25\}$) and RMAT-

B $\{0.55, 0.15, 0.15, 0.15\}$) which exhibit a large variation in degree distribution similar to graphs popularly known as scale free small world networks. RMAT-G and RMAT-B also have contain local subcommunities. The degree distribution of RMAT-B is much wider than that of RMAT-G

Biological Networks: The biological data represents gene correlation networks downloaded from NCBI's GEO database [16]. We used two datasets: (i) GSE5140 that contains results of age related changes in the hypothalamus tissue of creatine supplemented mice and untreated mice, and (ii) GSE17072 that contains results of cancer-related mutations from normal breast tissue to non-familial breast cancerous tissue. The network was built by comparing the Pearson correlation coefficients ($\rho \leq 0.0005$) of all gene-pairs in each dataset; genes with high correlations ($0.95 \leq \rho \leq 1.00$) were connected to form the network. The biological networks, also exhibit a power law degree distribution and form communities, but their size is much smaller than the synthetically generated ones due to the limitations in experimental data. However the ratio of edges to vertices of the biological networks is much higher.

Comparison Between Biological and Synthetic Networks: We observe that there exist difference in the structural characteristics of the synthetic versus biological networks. Notably, the average clustering coefficient distributions for the synthetic networks reveal that most nodes have from 2 to 100 neighbors and a very low average clustering coefficient - ranging from 0.00 to 0.20 coefficient distribution reveals neighbors in the range of 2 to 150, and a much wider range of clustering coefficients; with high average clustering coefficients having a smaller number of neighbors. Nodes with the highest degree tend to have the smallest average clustering coefficients. In fact as the degree of a node rises, its clustering coefficient tends to decrease. This reflects the theory that biological networks tend to be *assortative* [17], i.e. two hubs are unlikely to be connected. It is not beneficial for hub nodes in biological networks to be connected to each other, or even for their neighbors to be well connected, because such connections introduce vulnerability in a network for intelligent attacks. In contrast no consistent pattern of clustering coefficient can be identified in the synthetic networks. This observation reflects the fact that networks obtained from different domains exhibit different characteristics.

The contrasting characteristics of the two kinds of inputs will help explain the differences in the performance of our algorithm, as presented in Section V.

V. Results And Discussion

We now present the experimental results. We use a compressed storage format to store the graphs in memory, where the neighbors of each vertex are stored contiguously. We run two different versions of the algorithm, an *unoptimized version* where the neighbors for each vertex is in an unordered format, and an *optimized version* where the neighbor lists are ordered, which allows a vertex to find its current lowest parent quickly. The run times presented for optimized versions do not include the sorting time. In our implementation, we exploit the fact that the chordal edge set of a vertex automatically gets built in an orderly manner because it considers its parents in an order. Therefore, testing set intersections is efficient, linear in terms of the size of the smallest set. We provide performance results on the two platforms separately, and then consider the relative performance for two instances. We further evaluate the performance of our algorithm by considering the distinct characteristics of these two kinds of inputs, synthetical and biological.

Performance on XMT: The first set of results are based on executing Algorithm 1 on the Cray XMT platform. The results are presented in Figure 4 for synthetical graphs, and Figure 5 for biological graphs. The plots for XMT are on the left side of these figures. The figures present both strong scaling (the same problem on different number of processors) and weak scaling (different sizes of problems on the same number of processors) results. We observe good scalability of our algorithm on the XMT for both classes of input. Weak scaling experiments are done for the synthetic graph by considering different scales (powers of two): 24, 25, and 26. At each scale the graph approximately doubles in size in terms of the number of vertices and edges. We plot compute time in seconds on a log (base two) scale along the vertical axis, and the number of processors along the horizontal axis. Note that the number of processors is doubled at each data point, and therefore, the scaling plots are log-log plots.

In Figure 4, we plot performance on XMT on synthetic graphs: RMAT-ER, RMAT-G and RMAT-B. Among these graphs, which are of roughly the same size, RMAT-B graphs require the most time to complete, because they have higher maximum degree, the largest variation in vertex degrees, as well as the most densely connected components. These characteristics aggravate the performance of the unoptimized algorithm. However, the two algorithm variants show similar performance on RMAT-ER and RMAT-G. Note that the optimized version (Opt) is nearly twice as fast as the unoptimized (Unopt) version for RMAT-B. We also observe that we lose efficiency at full machine utilization (128 processors) for many cases. We request for about 100 thread-streams on each processor. Thus, the total number of threads could reach up to $12,800$ on the entire system.

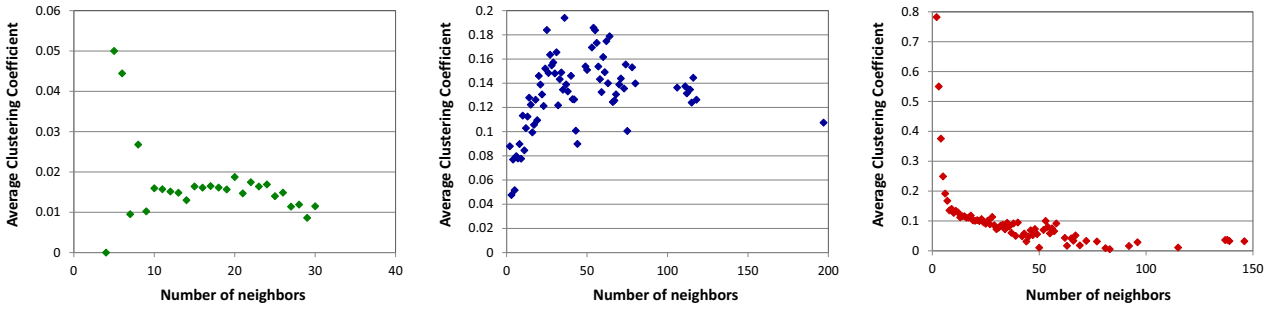
In Figure 5, we show performance of four gene correlation networks, GSE5140-CRT, GSE5140-UNT, GSE17072-CTL, and GSE17072-NON. These graphs are considerably small for XMT, and therefore, we only show performance for up to 16 processors. We observe that the optimized version of the algorithm is much faster than the unoptimized version on these inputs.

Performance on Opteron: Our second set of results are on execution of synthetic and biological graphs on the AMD

Group	Vertices	Edges	Avg Degree	Max Degree	Variance	Edges by Vertices
RMAT-ER(24)	16,777,216	134,217,654	8	42	16	7.99
RMAT-ER(25)	33,554,432	268,435,385	8	41	16	7.99
RMAT-ER(26)	67,108,864	536,870,837	8	48	16	7.99
RMAT-G(24)	16,777,216	134,181,095	8	1,278	416	7.99
RMAT-G(25)	33,554,432	268,385,483	8	1,489	442	7.99
RMAT-G(26)	67,108,864	536,803,101	8	1,800	469	7.99
RMAT-B(24)	16,777,216	133,658,229	8	38,143	8,086	7.99
RMAT-B(25)	33,554,432	267,592,474	8	54,974	9,539	7.99
RMAT-B(26)	67,108,864	535,599,280	8	77,844	11,214	7.99
GSE5140(CRT)	45,023	714,628	16	690	630	15.87
GSE5140(UNT)	45,020	644,651	14	315	421	14.31
GSE17072(CTL)	48,803	949,094	19	365	1702	19.44
GSE17072(NON)	48,803	1,109,553	23	463	2537	22.73

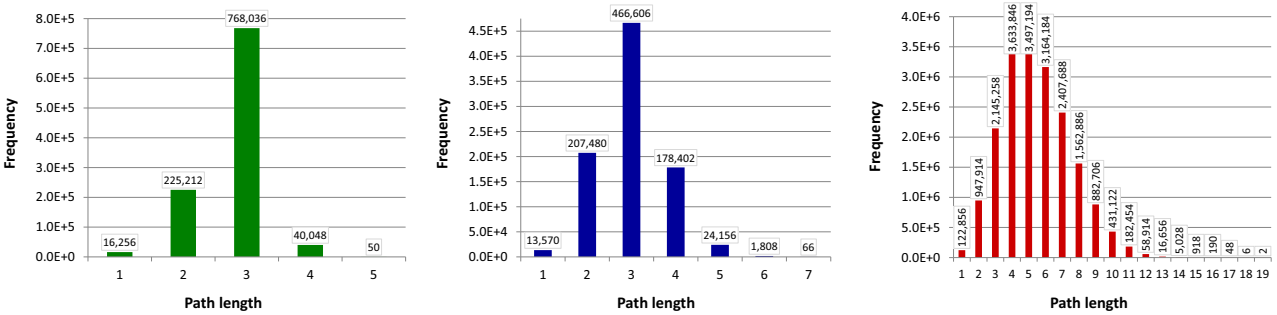
Table I

PROPERTIES OF THE TEST SUITE OF GRAPHS. THE NUMBERS IN PARENTHESIS FOR THE RMAT GRAPHS DENOTES THE SCALE, WHICH DETERMINES THE NUMBER OF VERTICES(2^{SCALE}) OF THE GRAPH.THE GENE CORRELATION NETWORKS GSE5140 ARE GROUPED INTO CREATINE TREATED (CRT) AND UNTREATED (UNT) MODELS, AND THE GSE1702 ARE GROUPED INTO NORMAL (OR CONTROL) TISSUES (CTL) AND NON-FAMILIAL CANCEROUS TISSUES (NON).



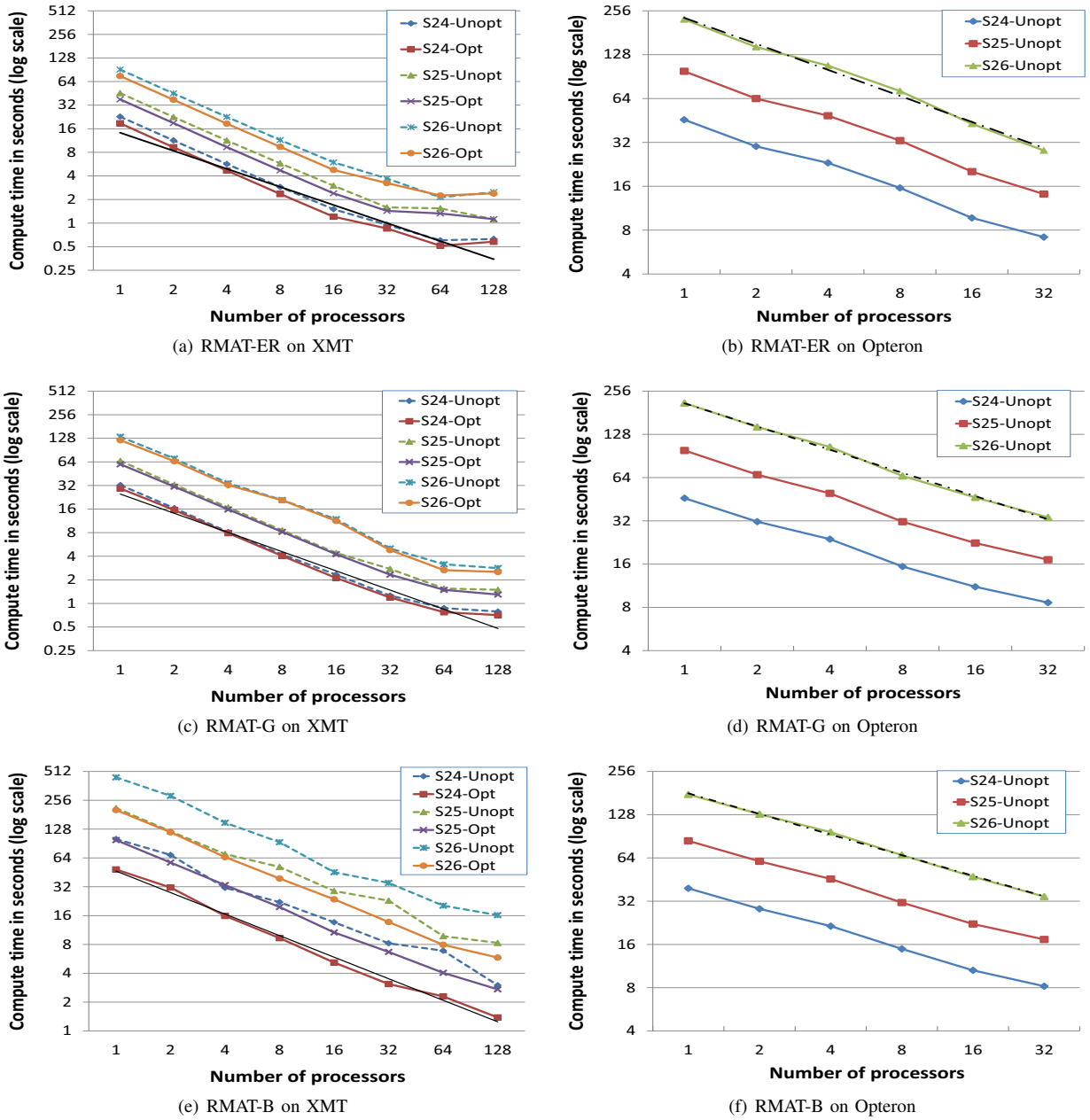
(a) RMAT-ER-10 (b) RMAT-B-10 (c) GSE5140(UNT)

Figure 2. **Average Clustering Coefficient Vs. Number of Neighbors.** RMAT-ER and RMAT-B with SCALE=10 (1024 vertices), and GSE5140(UNT). In the biological networks nodes with high average clustering coefficients have fewer neighbors, indicating the *assortative* nature of the networks.



(a) RMAT-ER-10 (b) RMAT-B-10 (c) GSE5140(UNT)

Figure 3. **Distribution of Shortest Paths.** RMAT-ER and RMAT-B with SCALE=10 (1024 vertices), and GSE5140(UNT). The biological networks have a much wider distribution of shortest paths than the synthetically generated ones.



(a) RMAT-ER on XMT

(b) RMAT-ER on Opteron

(c) RMAT-G on XMT

(d) RMAT-G on Opteron

(e) RMAT-B on XMT

(f) RMAT-B on Opteron

Figure 4. **Performance of Synthetic Graphs on Cray-XMT and Opteron**: The Y-axis gives execution time in seconds (log scale), and the X-axis gives number of processors on XMT and number of cores on Opteron. We request for 100 threads per processor on the XMT; only one thread is executed per core on the Opteron. A solid black trend line is provided for plots on XMT, and black dashed lines for Opteron. Plots for optimized version are marked `Opt`, and those for unoptimized are marked `Unopt`.

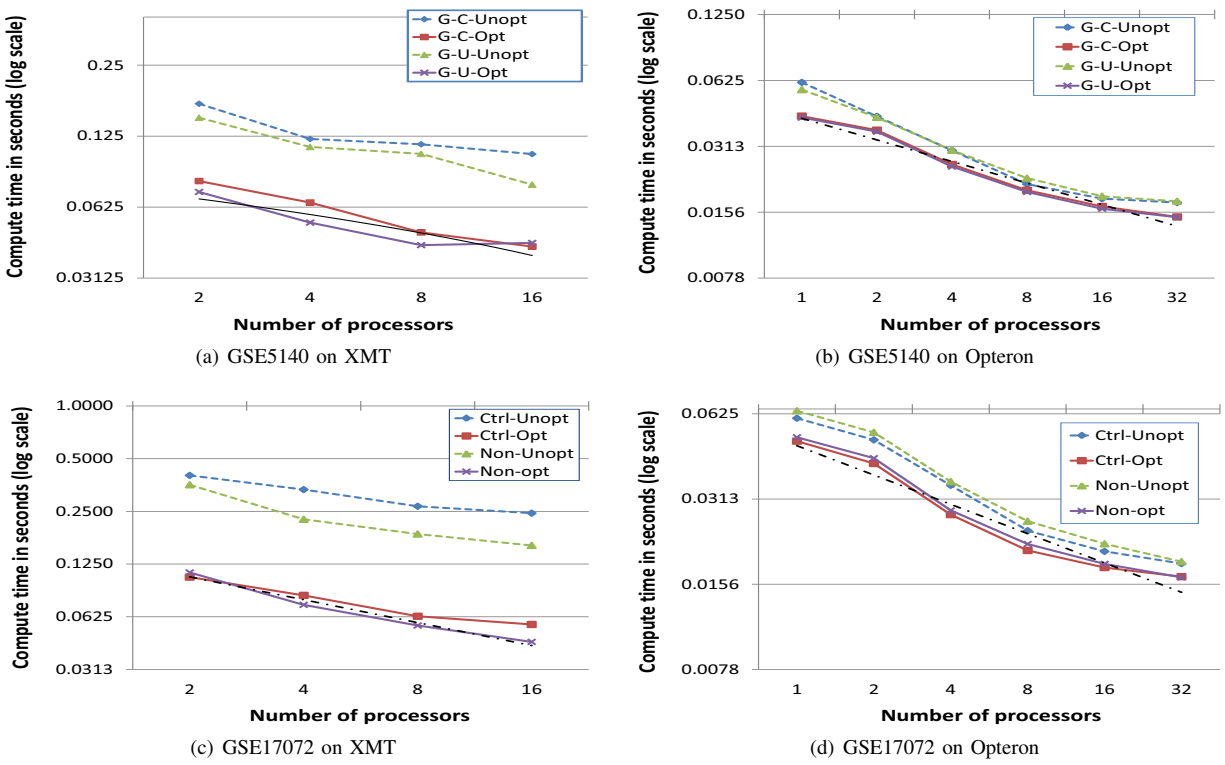


Figure 5. **Performance of Biological Networks on Cray-XMT and Opteron**: The Y-axis gives execution time in seconds (log scale), and the X-axis gives number of processors on XMT and number of cores on Opteron. We request for 100 threads per processor on the XMT; only one thread is executed per core on the Opteron. A solid black trend line is also provided.

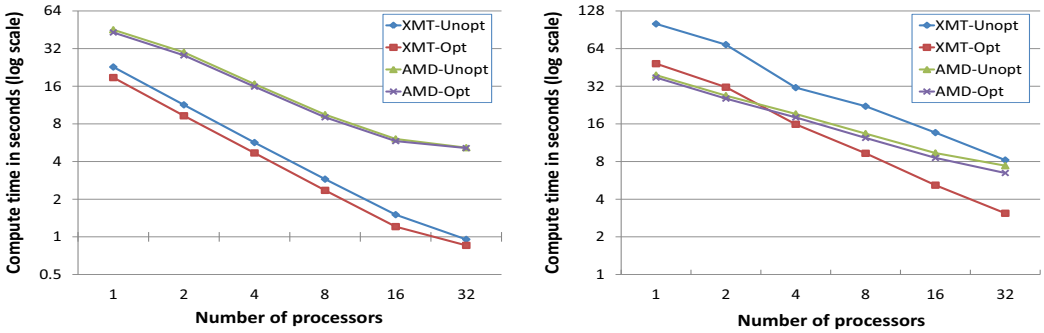
Opteron platform. These results are presented in Figure 4 for synthetical graphs, and Figure 5 for biological graphs. The plots for Opteron are on the right side of these figures. Our goal is to demonstrate suitability of Algorithm 1 for standard multicore platforms that are increasingly becoming available to researchers. The synthetic graphs are generated separately on the two platforms. However, the graphs have very minor differences in size and quality.

Algorithm 1 scales well on the Opteron platform. The differences between optimized and unoptimized algorithms was insignificant. We believe that the differences in the memory system of the two architectures – caches in particular – are the reason for this similarity. Setting aside the minor differences in inputs, execution on synthetic graphs took considerably more time on the Opteron for RMAT-ER and RMAT-G, but not RMAT-B. While Opteron was relatively insensitive to the differences in the three inputs, XMT was more sensitive. However, for biology networks Opteron was the faster of the two.

Relative Performance: In order to perform a uniform comparison between the two platforms, we also experimented with a set of synthetic graphs for Scale 24 that were generated on the XMT and used on both the platforms (Figure 6). As can be seen from this figure, RMAT-ER runs faster, and scales better, on the XMT. In contrast, execution of RMAT-B on Opteron is faster for smaller number of processors. As the number of processors increase, the optimized version of the code on XMT gives the smallest running time. However, performance on Opteron is still stronger than the unoptimized version on XMT. We also observe that there is very little difference in the performance of optimized and unoptimized versions on the Opteron.

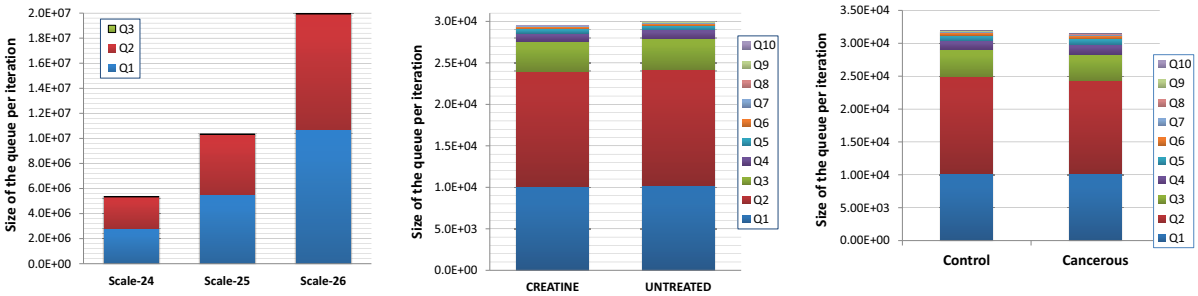
Our results indicate that Opteron is a better platform for networks that have densely connected components – RMAT-B and the biological networks. Densely connected components take more iterations to resolve, and might lead to loss of parallel efficiency. Therefore, a combination of faster clock speeds and a hierarchical cache system makes Opteron a better system. On the other hand, inputs like RMAT-ER and RMAT-G have fewer densely connected components and resolve quickly with large amounts of concurrency at every iteration. As a result, they perform much better on the XMT, which is better equipped to handle memory latencies via parallelism. Table II summarizes the speedup of Algorithm 1 for different networks. The speedup is much higher on XMT (as much as 47 for RMAT-G(with SCALE=26)) as compared to AMD (6 for RMAT-ER(SCALE=25)). However, the biological networks, because of their small size show much lower values.

Queue Sizes and Iteration Counts: Figure 7 gives the



(a) RMAT-ER (b) RMAT-B

Figure 6. **Relative Performance on XMT and Opteron**: The Y-axis gives the execution time in logarithmic scale and the X-axis gives the number of processors on XMT and cores on Opteron. While we request for about 100 threads per processor on the XMT, only one thread is executed per core on the Opteron. RMAT-ER and RMAT-B with scale 24 were generated on the XMT and used on both the platforms.



(a) RMAT-B (b) GSE5140 (c) GSE17072

Figure 7. **Queue Sizes and Number of Iterations**: Number of iterations in the **while** loop and sizes of queue (Q_1) at each iteration are plotted on the Y-axis (refer Algorithm 1). Different test instances are plotted on the X-axis. The results shown here are from the execution on XMT. Numbers on Opteron are similar.

Group	XMT (UnOpt)	XMT (Opt)	AMD (UnOpt)
RMAT-ER(24)	32.34	31.70	6.38
RMAT-ER(25)	40.29	29.57	**6.96**
RMAT-ER(26)	32.25	28.0	7.9
RMAT-G(24)	41.08	41.29	5.33
RMAT-G(25)	44.17	45.97	5.75
RMAT-G(26)	**47.35**	**47.97**	6.24
RMAT-B(24)	33.61	35.08	4.80
RMAT-B(25)	21.37	36.16	4.86
RMAT-B(26)	16.70	34.09	5.13
GSE5140(CRT)	1.40	1.22	3.54
GSE5140(UNT)	1.43	1.14	2.85
GSE17072(CTL)	1.52	1.25	3.41
GSE17072(NON)	2.05	1.65	3.12

Table II
Speedup for different networks on XMT and Opteron. SPEEDUP ON XMT IS PROVIDED FOR 128 PROCESSORS, AND ON OPTERON IT IS FOR 32 PROCESSORS, RELATIVE TO SINGLE PROCESSOR PERFORMANCE ON EACH PLATFORM RESPECTIVELY.

number of iterations required to process all the edges, and the number of vertices designated as lowest parent (LP) per iteration. Note that we consider one iteration of the **while** loop as an iteration and the size of Q_1 as LP. The number of elements in the queue represent the amount of parallel work available at each iteration. For all three synthetic graphs, a chordal subgraph was obtained in roughly three iterations. Most of the LPs were processed in the first and second iterations (slightly more in the second iteration), and the third iteration had 2 to 3 LPs. The number of edges identified as chordal were proportional to the size of the graph – larger the size, bigger was the chordal edge set.

In contrast, the biological networks required about 10 iterations to complete although they are much smaller in size and have smaller maximum degrees than RMAT-G and RMAT-B. Similar to synthetic graphs, most of the LPs were processed in the second iteration. These results indicate that not only the degree distribution, but also the *assortative nature* of the graph, as discussed in Section IV, is an important factor in determining the execution time.

We observe that only a small portion of the graphs in our test suite are chordal. The RMAT-ER graphs have about 11%, RMAT-G graphs have about 10% and the RMAT-B graphs have about 6% chordal edges. Although we are finding a *maximal* chordal subgraph, and not the *maximum* chordal subgraph, these values remain nearly constant across all the three scales. The biological graphs also have a small percentage of chordal edges – 8%(for untreated), 4%(for

creatine), 7% (for control) and 6% (for cancerous). These values suggest that although RMAT-B and biological graphs have densely connected components, the distance between these components, which form the non-chordal portions of a network, form a larger portion. This observation is also corroborated by the distribution of shortest paths as shown in Figure 3. RMAT-B has a slightly wider distribution of shortest paths than RMAT-ER indicating well separated densely connected components. The biological networks present the widest distribution of shortest paths.

VI. Conclusions and Future Plans

We presented a novel multithreaded algorithm for extracting maximal chordal subgraphs and demonstrated scalable performance on two multithreaded platforms. Using a set of inputs from synthetically generated as well as from biological experiments, we demonstrated how some of the graph properties influence performance on multithreaded platforms. In the near future, we plan to conduct experiments with a broader set of inputs, develop implementations on other multithreaded platforms such as general purpose graphics processing units (GPUs). We will also explore datasets from several classes of applications and how they might benefit from the graph sampling technique based on maximal chordal subgraphs.

Acknowledgments

This work was made possible by the College of Information Science and Technology, University of Nebraska at Omaha and Grant Number P20 RR16469 from the National Center for Research Resources(NCRR), a component of the National Institutes ofHealth (NIH). Its contents are the sole responsibility of the authors and do not represent the official views of NCRR or NIH. This work was also funded in part by the Center for Adaptive Super Computing Software - MultiThreaded Architectures (CASS-MT) at the U.S. Department of Energy's Pacific Northwest National Laboratory, which is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

References

[1] P. Dearing, D. Shier, and D. Warner, "Maximal chordal subgraphs," *Discrete Applied Mathematics*, vol. 20, no. 3, pp. 181 – 190, 1988.

[2] D. J. Rose and R. E. Tarjan, "Algorithmic aspects of vertex elimination," in *Proceedings of 7th Annual ACM Symposium on Theory of Computing*, ser. STOC '75. New York, NY, USA: ACM, 1975, pp. 245 – 254.

[3] J. L. Gross and J. Yellen, *Handbook of Graph Theory (Discrete Mathematics and Its Applications)*, 1st ed. CRC Press, Dec. 2003.

[4] K. Duraisamy, K. Dempsey, H. Ali, and S. Bhowmick, "A noise reducing sampling approach for uncovering critical properties in large scale biological networks," in *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS)*, july 2011, pp. 721 –728.

[5] K. Dempsey, K. Duraisamy, H. H. Ali, and S. Bhowmick, "A parallel graph sampling algorithm for analyzing gene correlation networks," *Procedia CS*, vol. 4, pp. 136 – 145, 2011.

[6] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. IEEE Computer Society, 2006, pp. 523–530. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2006.34

[7] D. A. Bader and G. Cong, "Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs," *J. Parallel Distrib. Comput.*, vol. 66, no. 11, pp. 1366–1378, 2006.

[8] K. Dempsey, K. Duraisamy, S. Bhowmick, and H. H. Ali, "The development of parallel adaptive sampling algorithms for analyzing biological networks," 2011, submitted to a workshop.

[9] D. Ediger, J. Riedy, D. A. Bader, and H. Meyerhenke, "Tracking structure of streaming social networks," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, ser. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1691 – 1699.

[10] D. Ediger, K. Jiang, J. Riedy, and D. Bader, "Massive streaming data analytics: A case study with clustering coefficients," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, april 2010, pp. 1 – 8.

[11] J. Riedy, M. H., D. Ediger, and D. Bader, "Parallel community detection for massive graphs," in *9th International Conference on Parallel Computing and Applied Mathematics (PPAM)*, September 2011, pp. 11 – 14.

[12] U. Catalyurek, J. Feo, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Multithreaded algorithms for graph coloring," 2011, submitted to a journal. Invited presentation at SIAM Conference on Computational Science and Engineering (CSE11).

[13] J. Feo, D. Harper, S. Kahan, and P. Konecny, "ELDORADO," in *CF '05: Proceedings of the 2nd Conference on Computing Frontiers*. New York, NY, USA: ACM, 2005, pp. 28 – 34.

[14] "AMD Opteron 6100 series processor," available at http://www.amd.com/us/products/embedded/processors/opteron/Pages/opteron-6100-series.aspx.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *In SIAM Conference on Data Mining*, 2004.

[16] A. Bender, J. Beckers, I. Schneider, and S. M. Holter, "Creatine improves health and survival of mice." *Neurobiol Aging*, vol. 9, pp. 1404–11, 2008.

[17] M. E. J. Newman, "Assortative mixing in networks," *Phys. Rev. Lett.*, vol. 89, pp. 208 701–1 – 208 701–4, Oct 2002.