CASS-MT Task #7 - Georgia Tech

GraphCT:
A Graph Characterization Toolkit

David A. Bader, David Ediger,
Karl Jiang & Jason Riedy

October 26, 2009
Outline

- Motivation
- What is GraphCT?
  - Package for Massive Social Network Analysis
  - Can handle graphs with billions of vertices & edges
- Key Features
  - Common data structure
  - A “buffet” of functions that can be combined
- Using GraphCT
- Future of GraphCT
- Function Reference
Driving Forces in Social Network Analysis

An explosion of data!

Facebook User Growth Since Creation

300 million active Facebook users worldwide in September 2009
Current Social Network Packages

- UCINet, Pajek, SocNetV, tnet
- Written in C, Java, Python, Ruby, R
- Limitations
  - Runs on workstation
  - Single-threaded
  - Several thousand to several million vertices
  - Low density graphs

- We need a package that will easily accommodate graphs with several billion vertices on large, parallel machines
The Cray XMT

- **Tolerates latency** by massive multithreading
  - Hardware support for 128 threads on each processor
  - Globally hashed address space
  - No data cache
  - Single cycle context switch
  - Multiple outstanding memory requests

- **Support for fine-grained, word-level synchronization**
  - Full/empty bit associated with every memory word

- **Flexibly supports dynamic load balancing**

- **GraphCT currently tested on a 64 processor XMT:** 8192 threads
  - 512 GB of globally shared memory

Image Source: cray.com
What is GraphCT?

- **Graph Characterization Toolkit**
- Efficiently summarizes and analyzes static graph data
- Built for large multithreaded, shared memory machines like the Cray XMT
- Increases productivity by decreasing programming complexity
- Classic metrics & state-of-the-art kernels
- Works on all types of graphs
  - directed or undirected
  - weighted or unweighted

Dynamic spatio-temporal graph
Key Features of GraphCT

- Low-level primitives to high-level analytic kernels
- Common graph data structure
- Develop custom reports by mixing and matching functions
- Create subgraphs for more in-depth analysis
- Kernels are tuned to maximize scaling and performance (up to 64 processors) on the Cray XMT

Load the Graph Data  
Find Connected Components  
Run k-Betweenness Centrality on the largest component
typedef struct {
    int numEdges;
    int numVertices;
    int startVertex[NE]; /* start vertex of edge, sorted, primary key */
    int endVertex[NE]; /* end vertex of edge, sorted, secondary key */
    int intWeight[NE]; /* integer edge weight */
    int edgeStart[NV]; /* per-vertex index into endVertex array */
    int marks[NV]; /* common array for marking or coloring of vertices */
} graph;
Using GraphCT
Usage options

Operations on input graphs can be specified in 3 ways:

- Via the command line
  - Perform a single graph operation
  - Read in graph, execute kernel, write back result

- Via a script [in progress]
  - Batch multiple operations
  - Intermediate results need not be written to file (though they can be)

- Via a developer’s API
  - Perform complex series of operations
  - Manipulate data structures
  - Implement custom functions
The command line interface
1. Command line parameters

Example: `./GraphCT-CLI -i patents.txt -t dimacs -o result.txt -z kcentrality 1`

- **-i**: Input file
- **-t**: Graph type, can currently be either ‘dimacs’ or ‘binary’. ‘binary’ type is binary compressed row format generated by GraphCT
- **-o**: Output file
- **-z**: Kernel type (see following sections):
2. Kernel types (index)

- Specified after \(-z\) flag
  - kcentrality \(k\) \(V_s\)
  - degree
  - conductance
  - modularity
  - components
  - clustering
  - transitivity
  - diameter \(n\)
3. Degree distribution & graph diameter

Diameter can only be ascertained by repeatedly performing breadth first searches different vertices.

- The more breadth first searches, the better approximation to the true diameter
- $diameter < P$
  - Does breadth first searches from $P$ percent of the vertices, where $P$ is an integer

Degree distribution:
- $z$ degree: gives
  - Maximum out-degree
  - Average out-degree
  - Variance
  - Standard deviation
4. Conductance and modularity

-z conductance, -z modularity

- Defined over colorings of input graph
  - Describe how tightly knit communities divided by a cut are
  - Not very meaningful in command line mode
  - In batch mode a coloring can be followed by conductance/modularity calculation

- In batch mode:
  - Finds connected components
  - Modularity uses component coloring as a partition
  - Conductance uses the largest component as the cut
5. Vertex k-Betweenness Centrality

-z kcentrality k Vs

- **Vs**: number of source vertices (of breadth first search)
  - Set equal to NV (number of vertices) for exact computation
- **k**: count shortest path length + k
- Outputs file with k-BC scores ordered by vertex number

- **Note**: Set *k* equal to 0 for betweenness centrality

6. Transitivity/clustering coefficient

-z transitivity

- **Writes output file with local transitivity coefficient of each vertex**
  - Measures number of transitive triads over total number of transitive triples

-z clustering

- **Writes output file with local clustering coefficient of each vertex**
  - Number of triangles formed by neighbors over number of potential triangles
  - Gives sense of how close vertex is to belonging to a clique

7. Component statistics

-z components

Statistics about connected components in graph
- Number of components
- Largest component size
- Average component size
- Variance
- Standard deviation

Writes output file with vertex to component mapping
Writing a script file [in progress]
1. Example script

read dimacs patents.txt => binary_pat.bin
print diameter 10
save graph
extract component 1 => component1.bin
print degrees
kcentrality 1 256 => k1scores.txt
kcentrality 2 256 => k2scores.txt
restore graph
extract component 2
print degrees
2. Script fundamentals

- Work on single ‘active graph’
- Can save and restore graphs at any point, like memory feature on pocket calculator
- Operations can:
  - Output data to the screen (e.g. degree information)
  - Output data to file (e.g. kcentrality data)
  - Modify the active graph (extract subgraph, component)
3. Example breakdown

read dimacs patents.txt => binary_pat.bin

- Two operations: reads in ‘patents.txt’ as a dimacs graph file, and writes the resulting graph back out as a binary file called ‘binary_pat.dat’
  - Binary graph is usually smaller and quicker to load
  - => filename always takes the output of a particular command and writes it to the file ‘filename’
  - Current graph formats are ‘dimacs’ and ‘binary’

print diameter 10

- print command is used to print information to the screen
  - Shows the estimated diameter based on BFS runs from 10% of vertices
3. Example breakdown (cont.)

save graph
- Retain the current active graph for use later

extract component 1 => component1.bin
- extract command is used to use a coloring to extract a subgraph from the active graph
  - component 1 colors the largest connected component
- Writes resulting graph to a binary file

print degrees
- Any kernel from the previous section may be used
- If output is a graph or per-vertex data, it cannot be printed
3. Example breakdown (cont.)

kcentrality 1 256 => k1scores.txt

- Calculates k=1 betweenness centrality based on breadth first searches from 256 source vertices
  - Result stored in ‘k1scores.txt’, one line per vertex
  - kcentrality result cannot be printed to screen since it is per-vertex data

restore graph

- Restore active graph saved earlier
- Can restore same graph multiple times
3. Example breakdown (cont.)

extract component 2

- Extract the second largest component of the graph
Graph parsers
DIMACS graph parser

c comments
c here
p max n m
e v1 v2 w

- **DIMACS file:**
  - `c` = comment
  - `p` = problem line: `n` = number of vertices, `m` = number of edges
  - `e` = edge: indicates an edge from `v1` to `v2` of weight `w`

- **Use standalone parser or read directly into GraphCT**
  - Standalone parser outputs binary format graph file
    - Good if graph will be used multiple times to reduce I/O time
From data to analysis

- GraphCT produces a simple listing of the metrics most desired by the analyst
- At a glance, the size, structure, and features of the graph can be described
- Output can be custom tailored to show more or less data
- Full results are written to files on disk for per-vertex kernels
  - k-Betweenness Centrality
  - Local clustering coefficients
  - BFS distance
- Excellent for external plotting & visualization software
The Future of GraphCT

- Additional high-level tools
  - Divisive betweenness-based community detection
  - Greedy agglomerative clustering (CNM)
  - Hybrid techniques
  - Additional subgraph generators

- Helper functions
  - Data pre-processing
  - Support for common graph formats

- Extension to support dynamic graph data
  - STINGER example
Experimental Kernels
Random walk subgraph extraction

void findSubGraphs(graph *G, int nSG, int subGraphPathLength)

- Choose a number of random starting vertices nSG
- Perform a BFS of length subGraphPathLength from each source vertex
- Extract the subgraph:
  subG = genSubGraph(G, NULL, 1);
Developer’s Notes:
A Programming Example
1. Initialization & graph generation

// I want a graph with ~270 million vertices
getUserParameters(28);

// Generate the graph tuples using RMAT
SDGdata = (graphSDG*) malloc(sizeof(graphSDG));
genScalData(SDGdata, 0.57, 0.19, 0.19, 0.05);

// Build the graph data structure
G = (graph *) malloc(sizeof(graph));
computeGraph(G, SDGdata);
// Display statistics on the vertex out-degree
calculateDegreeDistributions(G);

// Find the graph diameter exactly
calculateGraphDiameter(G, NV);
// This will require 270M breadth first searches!

// Estimate the graph diameter
calculateGraphDiameter(G, 1024);
// This only does 1024 breadth first searches
3. Mark & summarize connected components

```c
// run connected components & store the result in the graph
numComp = connectedComponents(G);

// display component size statistics based on colors
calculateComponentDistributions(G, numComp, &max, &maxV);
```
4. Find 10 highest 2-betweenness vertices

BC = (double *) malloc(NV * sizeof(double));

// k=2, 256 source vertices
kcentrality(G, BC, 256, 2);

printf("Maximum BC Vertices\n");
for (j = 0; j < 10; j++) {
    maxI  = 0;
    maxBC = BC[0];
    for (i = 1; i < NV; i++)
        if (BC[i] > maxBC) {maxBC = BC[i]; maxI = i;}
    printf("#%2d: %8d - %9.6lf\n", j+1, maxI, maxBC);
    BC[maxI] = 0.0;
}
Function Reference
Initialize default environment

```c
void getUserParameters(int scale)
```

- Sets a number of application parameters
- `scale`: determines size of graph generation
  - \( \log_2 \) Number of Vertices
Load external graph data

```c
int graphio_b(graph *G, char *filename)
```

- Load from a binary data file containing compressed data structure using 4-byte integers
- Format:
  - Number of Edges (4 bytes)
  - Number of Vertices (4 bytes)
  - Empty padding (4 bytes)
  - `edgeStart` array `(NV * 4 bytes)`
  - `endVertex` array `(NE * 4 bytes)`
  - `intWeight` array `(NE * 4 bytes)`
Scalable data generator

```c
void genScalData(graphSDG*, double a, double b, double c, double d)
```

- **Input:**
  - RMAT parameters A, B, C, & D
  - Must call `getUserParameters()` prior to calling this function

- **Output:**
  - graphSDG data structure (raw tuples)

- **Note:** this function should precede a call to `computeGraph()` to transform tuples into a graph data structure

---

D. Chakrabarti, Y. Zhan, and C. Faloutsos. “R-MAT: A recursive model for graph mining”. In *Proc. 4th SIAM Intl. Conf. on Data Mining* (SDM), Orlando, FL, April 2004. SIAM.
Graph construction

```c
void computeGraph(graph *G, graphSDG *SDGdata)
```

**Input:**
- graphSDG data structure

**Output:**
- graph data structure
Directed graph -> undirected

```c
graph * makeUndirected(graph *G)
```

▶ Input:
  - graph data structure

▶ Output:
  - Returns an undirected graph containing bidirectional edges for each edge in the original graph. Duplicate edges are removed automatically.
Generate a subgraph

```c
graph * genSubGraph(graph *G, int NV, int color)
```

**Input:**
- graph data structure (marks[] must be set)
- NV should always be set to NULL
- color of vertices to extract

**Output:**
- Returns a graph containing only those vertices in the original graph marked with the specified color
K-core graph reduction

```c
graph * kcore(graph *G, int K)
```

- **Input:**
  - graph data structure
  - minimum out-degree K

- **Output:**
  - Returns a graph containing only those vertices in the original graph with an out-degree of at least K
Vertex k-Betweenness Centrality

double kcentrality(graph *G, double BC[], int Vs, int K)

- **Vs**: number of source vertices
  - Set equal to G->NV for an exact computation
- **K**: count shortest path length + K
- **BC[]**: stores per-vertex result of computation

**Note**: Set K equal to 0 for betweenness centrality

Degree distribution statistics

void calculateDegreeDistributions(graph*)

Input:
- graph data structure

Output:
- Maximum out-degree
- Average out-degree
- Variance
- Standard deviation
Component statistics

```c
void calculateComponentDistributions (graph *G,
   int numColors, int *max, int *maxV)
```

**Input:**
- graph data structure
- numColors: largest integer value of the coloring

**Output:**
- max: size of the largest component
- maxV: an integer ID within the largest component
double computeModularityValue(graph *G,
   int membership[], int numColors)

Input:
- graph data structure
- membership[]: the vertex coloring (partitioning)
- numColors: the number of colors used above

Output:
- Modularity score is returned
Conductance score

double computeConductanceValue(graph *G, int membership[])

- **Input:**
  - graph data structure
  - membership[]: a binary partitioning

- **Output:**
  - Conductance score is returned
int connectedComponents(graph *G)

- **Input:**
  - graph data structure

- **Output:**
  - G->marks[] : array containing each vertex’s coloring where each component has a unique color
  - Returns the number of connected components
Breadth first search

```c
int * calculateBFS(graph *G, int startV, int mode)
```

**Input:**
- graph data structure
- startV: vertex ID to start the search from
- mode:
  - mode = 0: return an array of the further vertices where the first element is the number of vertices
  - mode = 1: return an array of the distances from each vertex to the source vertex

**Output:**
- Returns an array according to the mode described above

Graph diameter

```c
int calculateGraphDiameter(graph *G, int Vs)
```

- **Input:**
  - graph data structure
  - Vs: number of breadth-first searches to run

- **Output:**
  - Returns the diameter (if Vs = NV) or the length of the longest path found

- **Note:** *this can be used to find the exact diameter or an approximation if only a subset of source vertices is used*
Global transitivity coefficient

double calculateTransitivityGlobal(graph *G)

Input:
- graph data structure

Output:
- Returns the global transitivity coefficient (for both directed and undirected graphs)

Tore Opsahl and Pietro Panzarasa. “Clustering in weighted networks,”
Local transitivity coefficient

double * calculateTransitivityLocal(graph *G)

Input:
- graph data structure

Output:
- Returns the local transitivity coefficient for each vertex in an array

Local clustering coefficient

double * calculateClusteringLocal(graph *G)

- **Input:**
  - graph data structure

- **Output:**
  - Returns the local clustering coefficient for each vertex in an array