HPC Scalable Graph Analysis Benchmark

Contributors: David A. Bader (Georgia Tech), John Feo (Cray), John Gilbert (UC Santa Barbara), Jeremy Kepner (MIT/LL), David Koester (Mitre), Eugene Loh (Sun Microsystems), Kamesh Madduri (Georgia Tech), Bill Mann (formerly of MIT/LL), Theresa Meuse (MIT/LL), Eric Robinson (MIT/LL)

Version History:
V1.0: Released 24 February 2009

Version 1.0 of this document was generated primarily based on the HPCS SSCA#2 benchmark version 2.2. It was modified to present a more general graph analysis benchmark to the high performance computing (HPC) community.

2.0 Brief Description of the Scalable Graph Analysis Benchmark

The intent of this benchmark is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a *weighted, directed graph*. In addition to a kernel to construct the graph from the input tuple list, there will be three additional computational kernels to operate on the graph. Each of the kernels will require irregular access to the graph's data structure, and it is possible that no single data layout will be optimal for all four computational kernels.

This benchmark includes a *scalable* data generator that produces edge tuples containing the start vertex, end vertex, and weight for each directed edge. The first kernel constructs the graph in a format usable by all subsequent kernels. No subsequent modifications are permitted to benefit specific kernels. The second kernel extracts edges by weight from the graph representation and forms a list of the selected edges. The third kernel extracts a series of subgraphs formed by following paths of specified length from a start set of initial vertices. The set of initial vertices are determined by kernel 2. The fourth computational kernel computes a centrality metric that identifies vertices of key importance along shortest paths of the graph. All the kernels are timed.

In the descriptions of the various components of the computational kernels below, sequential pseudocode is provided for some components.

2.0.1 References

D.A. Bader, K. Madduri, J.R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems," *CTWatch Quarterly*, 2(4B):41-51, November 2006.

D.A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," *The 12th International Conference on High Performance Computing* (HiPC 2005), Springer-Verlag LNCS **3769**:465-476, 2005.

2.1 Scalable Data Generator

2.1.1 Brief Description

The scalable data generator will construct a list of edge tuples containing vertex identifiers and weights that represent data assigned to the edges of the multigraph. Each edge is directed from the first vertex of its tuple to the second. The edge weights are positive integers chosen from a uniform random distribution. The generated list of tuples must not exhibit any locality that can be exploited by the computational kernels. Thus, the vertex numbers must be randomized and a random ordering of tuples must be presented to subsequent kernels. The data generator may be parallelized, but the vertex names must be globally consistent and care must be taken to minimize effects of data locality at the processor level.

2.1.2 Detailed Text Description

The edge tuples will have the form **`<StartVertex, EndVertex, Weight>`** where **`StartVertex`** is the vertex where the directed edge begins, **`EndVertex`** is the vertex where the edge terminates, and **`Weight`** is a positive integer.

The input values required to describe the graph are as follows:

- **N**: the total number of vertices. An implementation may use any set of **N** distinct integers to number the vertices, but at least 48 bits must be allocated per vertex number. Other parameters may be assumed to fit within the natural word of the machine. **N** is derived from the problem's scaling parameter.
- **M**: the number of directed edges. **M** is derived from **N**.
- **C:** the maximum value of an integer edge weight. Weights are chosen uniformly at random from the distribution [1, **C**]. **C** is also derived from the problem's scaling parameter.

The graph generator is based on the Recursive MATrix (R-MAT) scale-free graph generation algorithm [Chakrabarti, *et al.*, 2004]. For ease of discussion, the description of this R-MAT generator uses an adjacency matrix data structure; however, implementations may use any alternate approach that outputs the equivalent list of edge tuples. This model recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes edges within these partitions with unequal probabilities. Initially, the adjacency matrix is empty, and edges are added one at a time. Each edge chooses one of the four partitions with probabilities *a*, *b*, *c*, and *d*, respectively. At each stage of the recursion, the parameters are varied slightly and renormalized. The pseudo-code for this generator is detailed in the next section.

It is possible that the R-MAT algorithm may create a small number of multiple edges between two vertices, and even self loops. Multiple edges, self-loops, and isolated vertices, may be ignored in the subsequent kernels. The algorithm also generates the data tuples with high degrees of locality. Thus, as a final step, vertex numbers must be randomly permuted, and then edge tuples randomly shuffled.

It is permissible to run the data generator in parallel. In this case, it is necessary to ensure that the vertices are named globally, and that the generated data does not possess any locality in the local and/or global memory system.

The scalable data generator may be run in conjunction with kernels 1 through 4, or the data generator may be run separately with the data stored to disk. If stored to disk, the data may be retrieved before starting kernel 1. The data generator operations need not be timed.

In addition, there are a couple input parameters required by the graph kernels, which are outside of the scope of the Scalable Data Generator:

- **SubGraphPathLength** – the maximum path length (in edges, from a specified source vertex) for sub-graphs generated by the subgraph extraction kernel 3.
- **K4approx** - is an integer used in an approximate algorithm for kernel 4.

2.1.3 Selected pseudo-code to generate the R-MAT power-law graph

This is an attractive implementation in that it is embarrassingly parallel and does not require the explicit formation the adjacency matrix.

```
% Set number of vertices.
N    = 2^SCALE;

% Set number of edges.
M    = 8*N;

% Set R-MAT probabilities.
% Parameters can't be symmetric in order for it to be
% a power law distribution.
a = 0.55; b = 0.1; c = 0.1; d = 0.25;

% Create index arrays.
ii = ones(M,1);
jj = ones(M,1);
% Loop over each order of bit.
ab = a + b;
c_norm = c/(c + d);
a_norm = a/(a + b);

for ib = 1:SCALE
  % Compare with probabilities and set bits of indices.
  ii_bit = rand(M,1) > ab;
  jj_bit = rand(M,1) > ( c_norm.*ii_bit + a_norm.*not(ii_bit) );
  ii = ii + (2^(ib-1)).*ii_bit;
  jj = jj + (2^(ib-1)).*jj_bit;
end

% Create adjacency matrix for viewing purposes.
AA = sparse(ii,jj,ones(M,1));
```

2.1.4 Suggested Parameters Settings

The primary input parameters are expressed in terms of a single integer value SCALE, which may be used to increase the problem size (for example, SCALE = 31, 32, 33, …). The default parameters for R-MAT and subsequent kernels are also listed below:

Benchmark Parameter	v2 setting
SCALE	an integer value
SubGraphPathLength	3
K4approx	an integer value between 1 and SCALE

Note that for future reference, we will assume the input graph has $N = 2^{\text{SCALE}}$ vertices, $M = 8N$ edges and a maximum integer weight $C = 2^{\text{SCALE}}$. K4approx is an integer used in an approximate algorithm for Kernel 4. When K4approx = SCALE, we say the implementation is **exact**. For other settings of K4approx, we say the implementation is **approximate**.

2.1.5 References

D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A recursive model for graph mining*, SIAM Data Mining 2004.

Section 17.6, Algorithms in C (third edition). Part 5 Graph Algorithms, Robert Sedgewick (Programs 17.7 and 17.8)

P. Sanders, *Random Permutations on Distributed, External and Hierarchical Memory*, Information Processing Letters 67 (1988) pp 305-309.

2.2 Kernel 1 – Graph Construction

2.2.1 Description

The first kernel must construct a (sparse) graph from a list of tuples; each tuple contains start and end vertex identifiers for a directed edge, and a weight that represents data assigned to the edge.

The graph can be represented in any manner, but it cannot be modified by or between subsequent kernels. Space may be reserved in the data structure for marking or locking, under the assumption that only one copy of a kernel will be run at a time.

There are various representations for sparse directed graphs, including (but not limited to) sparse matrices and (multi-level) linked lists. Representations for sparse directed graphs may be more complicated than for sparse simple graphs. For the purposes of this application, code developers are not given the maximum size of the graph (the number of vertices) and are expected to determine that value during graph construction.

The process of constructing the graph data structure from the set of tuples will be timed.

As an aid to verification, statistics may be collected on the graph. Calculating the statistics may occur during the generation process or may be deferred to an untimed verification step.

2.2.2 References
Section 17.6 Algorithms in C third edition Part 5 Graph Algorithms, Robert Sedgewick (Program 17.9)

2.3 Kernel 2 – Classify Large Sets

2.3.1 Description
Examine all edge weights to determine those vertex pairs with the largest integer weight. The output of this kernel will be an edge list, S, that will be saved for use in the following kernel. The process of generating the two lists/sets will be timed.

2.4 Kernel 3 – Graph Extraction

2.4.1 Description
For each of the edges in the set S, produce a subgraph which consists of the vertices and edges of all the paths of length *SubGraphPathLength* starting with that edge. A possible computational kernel for graph extraction is Breadth-First Search. The process of graph extraction will be timed.

2.4.2 References
Section 18.7 Algorithms in C third edition Part 5 Graph Algorithms, Robert Sedgewick (Programs 18.8 and 18.9)

2.5 Kernel 4 – Graph Analysis Algorithm

2.5.1 Brief Description
The intent of this kernel is to identify of the set of vertices in the graph with the highest betweenness centrality score. Betweenness Centrality is a shortest paths enumeration-based centrality metric, introduced by Freeman (1977). This is done using a betweenness centrality algorithm that computes this metric for every vertex in the graph. Let σ_{st} denote the number of shortest paths between vertices s and t, and $\sigma_{st}(v)$ the number of those paths passing through v. Betweenness Centrality of a vertex v is defined as $BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$.

The output of this kernel is a betweenness centrality score for each vertex in the graph and the set of vertices with the highest betweenness centrality score.

For kernel 4, we use the edge weights to filter a subset of edges E used in this kernel's input graph $G = (V, E)$. We select only edges which have at least one bit set in the three least significant bit positions of the binary representation of the edge weight. In other

words, edges with a weight evenly divisible by 8 are not considered in the betweenness centrality. Hence, $|E| \approx 7/8\, M = 7N$. Note that it is permissible for an implementation either to filter the edges first and then run an unweighted betweenness centrality algorithm, or to modify an unweighted betweenness centrality algorithm that inspects edge weights during execution. Kernel 4 is timed.

Because of the high computation cost of kernel 4, an **exact** implementation considers all vertices as starting points in the betweenness centrality metric, while an **approximate** implementation uses a subset of starting vertices (V_S). We use the input parameter K4approx, an integer set from 1 to SCALE, to vary the work performed by kernel 4. When K4approx equals SCALE, the implementation is **exact**. Otherwise, $|V_S| = 2^{\text{K4approx}}$ vertices are selected randomly from V.

Note that it is assumed the underlying graph forms a single connected component so that a graph traversal from any vertex $v \in V$ yields most or all of vertices in the graph. The R-MAT parameters are selected such that this is the case for most vertices in the graph. This allows the predicted operations required to very closely match the actual operations performed.

2.5.2 Recommended algorithm

A straight-forward way of computing betweenness centrality for each vertex would be as follows:

1. Compute the length and number of shortest paths between all pairs (s,t).
2. For each vertex v, calculate the summation of all possible pair-wise dependencies $\delta_{st}(v) = \dfrac{\sigma_{st}(v)}{\sigma_{st}}$.

Recently, Brandes (2001) proposed an algorithm that computes the exact betweenness centrality score for all vertices in the graph in $O(NM + N^2 \log N)$ for weighted graphs, and $O(NM)$ for unweighted graphs. The algorithm is detailed below:

Define the *dependency* of a source vertex $s \in V$ on a vertex $v \in V$ as $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$.

The betweenness centrality of a vertex v can then be expressed as $BC(v) = \sum_{s \neq v \in V} \delta_s(v)$.

Brandes noted that it is possible to augment Dijkstra's single-source shortest paths (SSSP) algorithm (for weight graphs) and breadth-first search (BFS) for unweighted graphs to compute the dependencies. Observe that a vertex $v \in V$ is on the shortest path between two vertices $s,t \in V$, iff $d(s,t) = d(s,v) + d(v,t)$. Define a set of *predecessors* of a vertex v on shortest paths from s as $pred(s,v)$. Now each time an edge $\langle u,v \rangle$ is

scanned for which $d(s,v) = d(s,u) + d(u,v)$, that vertex is added to the predecessor set $pred(s,v)$. Then, the following relation would hold: $\sigma_{sv} = \sum_{u \in pred(s,v)} \sigma_{su}$.

Setting the initial condition of $pred(s,v) = s$ for all neighbors v of s, we can proceed to compute the number of shortest paths between s and all other vertices. The computation of $pred(s,v)$ can be easily integrated into Dijkstra's SSSP algorithm for weighted graphs, or BFS Search for unweighted graphs. Brandes also observed that the dependency $\delta_s(v)$ satisfies the following recursive relation:

$$\delta_s(v) = \sum_{w:v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(1 + \delta_s(w)\right).$$

The algorithm proceeds as follows. First, compute n shortest paths trees, one for each $s \in V$. During these computations, also maintain the predecessor sets $pred(s,v)$. The dependencies can be computed by traversing the vertices in non-increasing order of their distance from s (i.e., starting at the leaves of the shortest paths tree, and working backwards towards s). To finally compute the centrality value of vertex v, we merely have to sum all dependencies values computed during the n different SSSP computations. The $O(N^2)$ space requirement can be reduced to $O(N+M)$ by maintaining a *running centrality score*.

2.5.3 Selected Pseudocode

We detail a parallel algorithm for computing betweenness centrality, based on Brandes's algorithm (Bader, 2006). Observe that parallelism can be exploited at two levels:

- The BFS/SSSP computations from each vertex can be done concurrently, provided the centrality running sums are updated atomically.
- Fine-grained parallelism in the BFS/SSSP can be exploited. For instance, when the adjacencies of a vertex are visited, the edge relaxation can be done concurrently.

Input: $G(V,E)$

Output: Array $BC[1\ldots n]$, where $BC[v]$ gives the centrality metric for vertex v

1 **for** *all* $v \in V$ ***in parallel*** **do**
2 $BC[v] \leftarrow 0$;
3 **let** $V_S \subseteq V$ ***and*** $|V_S| = 2^{K4approx}$ ***. /* exact vs. approximate */***
4 **for** *all* $s \in V_S$ ***in parallel*** **do**
5 $S \leftarrow$ empty stack;
6 $P[w] \leftarrow$ empty list, $w \in V$;
7 $\sigma[t] \leftarrow 0, t \in V$; $\sigma[s] \leftarrow 1$;

```
8       d[t] ← −1, t ∈ V;   d[s] ← 0;
9       queue Q ← s;
10      while Q ≠ ∅ do
11          dequeue v ← Q;
12          push v → S;
13          for each neighbor w of v in parallel do
14              if d[w] < 0 then
15                  enqueue w → Q;
16                  d[w] ← d[v] + 1;
17              if d[w] = d[v] + 1 then
18                  σ[w] ← σ[w] + σ[v];
19                  append v → P[w];
20      δ[v] ← 0, v ∈ V;
21      while S ≠ ∅ do
22          pop w ← S;
23          for v ∈ P[w] do
24              δ[v] ← δ[v] + (σ[v] / σ[w]) (1 + δ[w]);
25          if w ≠ s then
26              BC[w] ← BC[w] + δ[w];
```

2.5.4 Performance Metric (***TEPS***)

In order to compare the performance of various kernel 4 implementations across a variety of architectures, programming models, and productivity languages and frameworks, as well as normalizing across both **exact** and **approximate** implementations, we adopt a new performance metric described in this section. In the spirit of well-known computing rates *floating-point operations per second* (flops) measured by the Linpack benchmark and *global updates per second* (GUPs) measured by the RandomAccess benchmark, we define a new rate called *traversed edges per second* (**TEPS**). We measure **TEPS** through the benchmarking of kernel 4 as follows. Let $\text{time}_{K4}(N)$ be the measured execution time for kernel 4. For both the exact and approximate implementations of betweenness centrality, the number of graph edges traversed is directly proportional to $|V_S| * |E| \approx |V_S| * 7N$, where $|V_S| = N$ for the exact implementation and $|V_s| = 2^{\text{K4approx}}$ for the approximate implementation. We define the normalized performance rate (number of edge traversals per second) as

$$TEPS(N) = \begin{cases} 7N * N \,/\, \text{time}_{K4}(N), & \text{for exact implementation;} \\ 7N * 2^{\text{K4approx}} \,/\, \text{time}_{K4}(N), & \text{for approximate implementation.} \end{cases}$$

2.5.5 References

D.A. Bader and K. Madduri, Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks, *Proc. The 35th International Conference on Parallel Processing (ICPP)*, Columbus, OH, August 2006.

U. Brandes, A faster algorithm for betweenness centrality. *J. Mathematical Sociology*, 25(2):163–177, 2001.

L.C. Freeman, A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

2.6 Validation

It is not intended that the results of full-scale runs of this benchmark can be validated by exact comparison to a standard reference result. At full scale, the data set is enormous; and its exact details depend on the pseudo-random number generator used. Therefore, the validation of an implementation of the benchmark uses soft checking of the results.

We emphasize that the intent of this benchmark is to exercise these algorithms on the largest data sets that will fit on machines being evaluated. However, for debugging purposes it may be desirable to run on small data sets, and it may be desirable to verify parallel results against serial results, or even against results from the executable spec.

The executable specification verifies its results for kernels 1-4 by comparing them with results computed directly from the tuple list and, for modest problem sizes, by plotting various graphs which demonstrate that the results make sense.

It is easy to verify linear-time kernels 1-3. To validate kernel 4, we suggest substituting the R-MAT generator with a 2-dimensional torus generator, and then evaluating betweenness centrality for this instance. We have included a reference implementation of the 2D torus generator in the executable specification. By symmetry, every vertex in the 2D torus has the same betweenness centrality score, which is analytically computed to be:

$$BC(v) = \begin{cases} \frac{1}{2} \cdot 2^{\frac{3 \cdot SCALE}{2}} - 2^{SCALE} + 1, & \text{when SCALE is even;} \\ \frac{3}{4} \cdot 2^{\frac{3 \cdot SCALE - 1}{2}} - 2^{SCALE} + 1, & \text{when SCALE is odd.} \end{cases}$$

For validation, we can compare the betweenness scores in kernel 4 with the above result for a 2D torus.

Other straight-forward validity checks for kernel 4 include:

- All non-isolated vertices $v \in V$ will have a positive betweenness centrality score $BC[v] \leq N(N-1)$.
- Let $d(i, j)$ be the length of any shortest path from vertex i to j. Then $\sum_{\substack{\forall s,t \in V \\ s \neq t}} d(s,t) \geq \sum_{v \in V} BC[v]$. This relation may be used in validation by accumulating the shortest path lengths (one per pair of vertices) during kernel 4's execution. This sum should be greater than the sum of the N betweenness centrality scores.
- In the R-MAT instances with parameter settings, for the vertices with the highest betweenness centrality scores, there is a direct correlation to their out-degree. Hence, during the data generation phase, the vertices with the largest out-degree may be marked, and compared with the results from kernel 4.

2.7 Evaluation Criteria

In approximate order of importance, the goals of this benchmark are:

- Fair adherence to the intent of the benchmark specification
- Maximum problem size for a given machine
- Minimum execution time for a given problem size

Less important goals:

- Minimum code size (not including validation code)
- Minimal development time
- Maximal maintainability
- Maximal extensibility