Graph 500 Benchmark 1 ("Search")

Contributors: David A. Bader (Georgia Institute of Technology), Jonathan Berry (Sandia National Laboratories), Simon Kahan (Pacific Northwest National Laboratory and University of Washington), Richard Murphy (Micron Technology), E. Jason Riedy (Georgia Institute of Technology), and Jeremiah Willcock (Indiana University).

Version History:

V0.1 - Draft, created 28 July 2010

V0.2 - Draft, created 29 September 2010

V0.3 - Draft, created 30 September 2010

V1.0 - Created 1 October 2010

V1.1 - Created 3 October 2010

V1.2 - Created 15 September 2011

Version 0.1 of this document was part of the Graph 500 community benchmark effort, led by Richard Murphy (Micron Technology). The intent is that there will be at least three variants of implementations, on shared memory and threaded systems, on distributed memory clusters, and on external memory map-reduce clouds. This specification is for the first of potentially several benchmark problems.

References: "Introducing the Graph 500," Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, James A. Ang, Cray User's Group (CUG), May 5, 2010.

"DFS: A Simple to Write Yet Difficult to Execute Benchmark," Richard C. Murphy, Jonathan Berry, William McLendon, Bruce Hendrickson, Douglas Gregor, Andrew Lumsdaine, IEEE International Symposium on Workload Characterizations 2006 (IISWC06), San Jose, CA, 25-27 October 2006.

Table of Contents

1 Brief Description of the Graph 500 Benchmark

Data-intensive supercomputer applications are an increasingly important workload, but are ill-suited for platforms designed for 3D physics simulations. Application performance cannot be improved without a meaningful benchmark. Graphs are a core part of most analytics workloads. Backed by a steering committee of over 30 international HPC experts from academia, industry, and national laboratories, this specification establishes a large-scale benchmark for these applications. It will offer a forum for the community and provide a rallying point for data-intensive supercomputing problems. This is the first serious approach to augment the Top 500 with data-intensive applications.

The intent of this benchmark problem ("Search") is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, undirected graph. In addition to a kernel to construct the graph from the input tuple list, there is one additional computational kernel to operate on the graph.

This benchmark includes a scalable data generator which produces edge tuples containing the start vertex and end vertex for each edge. The first kernel constructs an *undirected* graph in a format usable by all subsequent kernels. No subsequent modifications are permitted to benefit specific kernels. The second kernel performs a breadth-first search of the graph. Both kernels are timed.

There are five problem classes defined by their input size:

toy
: 17GB or around 10^{10} bytes, which we also call level 10,

mini
: 140GB (10^{11} bytes, level 11),

small
: 1TB (10^{12} bytes, level 12),

medium

17TB (10^{13} bytes, level 13),

large

140TB (10^{14} bytes, level 14), and

huge

1.1PB (10^{15} bytes, level 15).

Table classes provides the parameters used by the graph generator specified below.

1.1 References

D.A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, Theresa Meuse, HPCS Scalable Synthetic Compact Applications #2 Graph Analysis (SSCA#2 v2.2 Specification), 5 September 2007.

2 Overall Benchmark

The benchmark performs the following steps:

1. Generate the edge list.
2. Construct a graph from the edge list (**timed**, kernel 1).
3. Randomly sample 64 unique search keys with degree at least one, not counting self-loops.
4. For each search key:
    1. Compute the parent array (**timed**, kernel 2).
    2. Validate that the parent array is a correct BFS search tree for the given search tree.
5. Compute and output performance information.

Only the sections marked as **timed** are included in the performance information. Note that all uses of "random" permit pseudorandom number generation.

3 Generating the Edge List

3.1 Brief Description

The scalable data generator will construct a list of edge tuples containing vertex identifiers. Each edge is undirected with its endpoints given in the tuple as StartVertex and EndVertex.

The intent of the first kernel below is to convert a list with no locality into a more optimized form. The generated list of input 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 kernel 1. 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.

3.2 Detailed Text Description

The edge tuples will have the form <StartVertex, EndVertex> where StartVertex is one endpoint vertex label and EndVertex is the other endpoint vertex label. The space of labels is the set of integers beginning with **zero** up to but not including the number of vertices N (defined below). The kernels are not provided the size N explicitly but must discover it.

The input values required to describe the graph are:

SCALE
: The logarithm base two of the number of vertices.

edgefactor
: The ratio of the graph's edge count to its vertex count (i.e., half the average degree of a vertex in the graph).

These inputs determine the graph's size:

N
: the total number of vertices, 2^{SCALE}. 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 edges. M = edgefactor * N.

The graph generator is a Kronecker generator similar to 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. These probabilities, the initiator parameters, are provided in Table initiator.

Initiator parameters for the Kronecker graph generator

A = 0.57	B = 0.19
C = 0.19	D = 1-(A+B+C) = 0.05

The next section details a high-level implementation for this generator. High-performance, parallel implementations are included in the reference implementation.

The graph generator creates a small number of multiple edges between two vertices as well as self-loops. Multiple edges, self-loops, and isolated vertices may be ignored in the subsequent kernels but must be included in the edge list provided to the first kernel. 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 the 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, either in local memory or globally across processors.

The scalable data generator should be run before starting kernel 1, storing its results to either RAM or disk. If stored to disk, the data may be retrieved before starting kernel 1. The data generator and retrieval operations need not be timed.

3.3 Sample High-Level Implementation of the Kronecker Generator

The GNU Octave routine in Algorithm generator is an attractive implementation in that it is embarrassingly parallel and does not require the explicit formation of the adjacency matrix.

```
function ij = kronecker_generator (SCALE, edgefactor)
%% Generate an edgelist according to the Graph500
%% parameters.  In this sample, the edge list is
%% returned in an array with two rows, where StartVertex
%% is first row and EndVertex is the second.  The vertex
%% labels start at zero.
%%
%% Example, creating a sparse matrix for viewing:
%%   ij = kronecker_generator (10, 16);
%%   G = sparse (ij(1,:)+1, ij(2,:)+1, ones (1, size (ij, 2)));
%%   spy (G);
%% The spy plot should appear fairly dense. Any locality
%% is removed by the final permutations.

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

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

  %% Set initiator probabilities.
  [A, B, C] = deal (0.57, 0.19, 0.19);

  %% Create index arrays.
  ij = ones (2, M);
  %% Loop over each order of bit.
  ab = A + B;
  c_norm = C/(1 - (A + B));
  a_norm = A/(A + B);

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

```
  ij = ij + 2^(ib-1) * [ii_bit; jj_bit];
end

%% Permute vertex labels
p = randperm (N);
ij = p(ij);

%% Permute the edge list
p = randperm (M);
ij = ij(:, p);

%% Adjust to zero-based labels.
ij = ij - 1;
```

3.4 Parameter Settings

The input parameter settings for each class are given in Table classes.

Problem class definitions and required storage for the edge list assuming 64-bit integers.

Problem class	Scale	Edge factor	Approx. storage size in TB
Toy (level 10)	26	16	0.0172
Mini (level 11)	29	16	0.1374
Small (level 12)	32	16	1.0995
Medium (level 13)	36	16	17.5922
Large (level 14)	39	16	140.7375
Huge (level 15)	42	16	1125.8999

3.5 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.

4 Kernel 1 – Graph Construction

4.1 Description

The first kernel may transform the edge list to any data structures (held in internal or external memory) that are used for the remaining kernels. For instance, kernel 1 may construct a (sparse) graph from a list of tuples; each tuple contains endpoint vertex identifiers for an edge, and a weight that represents data assigned to the edge.

The graph may be represented in any manner, but it may not be modified by or between subsequent kernels. Space may be reserved in the data structure for marking or locking. Only one copy of a kernel will be run at a time; that kernel has exclusive access to any such marking or locking space and is permitted to modify that space (only).

There are various internal memory representations for sparse graphs, including (but not limited to) sparse matrices and (multi-level) linked lists. For the purposes of this application, the kernel is provided only the edge list and the edge list's size. Further information such as the number of vertices must be computed within this kernel. Algorithm kernel1 provides a high-level sample implementation of kernel 1.

The process of constructing the graph data structure (in internal or external memory) from the set of tuples must be timed.

```
function G = kernel_1 (ij)
%% Compute a sparse adjacency matrix representation
%% of the graph with edges from ij.

  %% Remove self-edges.
  ij(:, ij(1,:) == ij(2,:)) = [];
  %% Adjust away from zero labels.
  ij = ij + 1;
  %% Find the maximum label for sizing.
  N = max (max (ij));
  %% Create the matrix, ensuring it is square.
  G = sparse (ij(1,:), ij(2,:), ones (1, size (ij, 2)), N, N);
  %% Symmetrize to model an undirected graph.
  G = spones (G + G.');
```

4.2 References

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

5 Sampling 64 Search Keys

The search keys must be randomly sampled from the vertices in the graph. To avoid trivial searches, sample only from vertices that are connected to some other vertex. Their degrees, not counting self-loops, must be at least one. If there are fewer than 64 such vertices, run fewer than 64 searches. This should never occur with the graph sizes in this benchmark, but there is a non-

zero probability of producing a trivial or nearly trivial graph. The number of search keys used is included in the output, but this step is untimed.

6 Kernel 2 – Breadth-First Search

6.1 Description

A Breadth-First Search (BFS) of a graph starts with a single source vertex, then, in phases, finds and labels its neighbors, then the neighbors of its neighbors, etc. This is a fundamental method on which many graph algorithms are based. A formal description of BFS can be found in Cormen, Leiserson, and Rivest. Below, we specify the input and output for a BFS benchmark, and we impose some constraints on the computation. However, we do not constrain the choice of BFS algorithm itself, as long as it produces a correct BFS tree as output.

This benchmark's memory access pattern (internal or external) is data-dependent with small average prefetch depth. As in a simple concurrent linked-list traversal benchmark, performance reflects an architecture's throughput when executing concurrent threads, each of low memory concurrency and high memory reference density. Unlike such a benchmark, this one also measures resilience to hot-spotting when many of the memory references are to the same location; efficiency when every thread's execution path depends on the asynchronous side-effects of others; and the ability to dynamically load balance unpredictably sized work units. Measuring synchronization performance is not a primary goal here.

You may not search from multiple search keys concurrently.

ALGORITHM NOTE We allow a benign race condition when vertices at BFS level k are discovering vertices at level k+1. Specifically, we do not require synchronization to ensure that the first visitor must become the parent while locking out subsequent visitors. As long as the discovered BFS tree is correct at the end, the algorithm is considered to be correct.

6.2 Kernel 2 Output

For each search key, the routine must return an array containing valid breadth-first search parent information (per vertex). The parent of the $\text{search}_{\text{key}}$ is itself, and the parent of any vertex not included in the tree is -1. Algorithm kernel2 provides a sample (and inefficient) high-level implementation of kernel two.

```
function parent = kernel_2 (G, root)
%% Compute a sparse adjacency matrix representation
%% of the graph with edges from ij.

  N = size (G, 1);
  %% Adjust from zero labels.
  root = root + 1;
  parent = zeros (N, 1);
  parent (root) = root;
```

```
  vlist = zeros (N, 1);
  vlist(1) = root;
  lastk = 1;
  for k = 1:N,
    v = vlist(k);
    if v == 0, break; end
    [I,J,V] = find (G(:, v));
    nxt = I(parent(I) == 0);
    parent(nxt) = v;
    vlist(lastk + (1:length (nxt))) = nxt;
    lastk = lastk + length (nxt);
  end

  %% Adjust to zero labels.
  parent = parent - 1;
```

7 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 and BFS algorithm 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 specification.

The executable specification verifies its results by comparing them with results computed directly from the tuple list.

Kernel 2 validation: after each search, run (but do not time) a function that ensures that the discovered breadth-first tree is correct by ensuring that:

1. the BFS tree is a tree and does not contain cycles,
2. each tree edge connects vertices whose BFS levels differ by exactly one,
3. every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree,
4. the BFS tree spans an entire connected component's vertices, and
5. a node and its parent are joined by an edge of the original graph.

Algorithm validate shows a sample validation routine.

```
function out = validate (parent, ij, search_key)
  out = 1;
  parent = parent + 1;
```

```
search_key = search_key + 1;

if parent (search_key) != search_key,
  out = 0;
  return;
end

ij = ij + 1;
N = max (max (ij));
slice = find (parent > 0);

level = zeros (size (parent));
level (slice) = 1;
P = parent (slice);
mask = P != search_key;
k = 0;
while any (mask),
  level(slice(mask)) = level(slice(mask)) + 1;
  P = parent (P);
  mask = P != search_key;
  k = k + 1;
  if k > N,
    %% There must be a cycle in the tree.
    out = -3;
    return;
  end
end

lij = level (ij);
neither_in = lij(1,:) == 0 & lij(2,:) == 0;
both_in = lij(1,:) > 0 & lij(2,:) > 0;
if any (not (neither_in | both_in)),
  out = -4;
  return
end
respects_tree_level = abs (lij(1,:) - lij(2,:)) <= 1;
if any (not (neither_in | respects_tree_level)),
  out = -5;
  return
end
```

8 Computing and Outputting Performance Information

8.1 Timing

Start the time for a search immediately prior to visiting the search root. Stop the time for that search when the output has been written to memory. Do not time any I/O outside of the search

routine. If your algorithm relies on problem-specific data structures (by our definition, these are informed by vertex degree), you must include the setup time for such structures in *each search*. The spirit of the benchmark is to gauge the performance of a single search. We run many searches in order to compute means and variances, not to amortize data structure setup time.

8.2 Performance Metric (TEPS)

In order to compare the performance of Graph 500 "Search" implementations across a variety of architectures, programming models, and productivity languages and frameworks, 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 HPCC RandomAccess benchmark, we define a new rate called traversed edges per second (TEPS). We measure TEPS through the benchmarking of kernel 2 as follows. Let $time_{K2}(n)$ be the measured execution time for kernel 2. Let m be the number of input edge tuples within the component traversed by the search, counting any multiple edges and self-loops. We define the normalized performance rate (number of edge traversals per second) as:

$TEPS(n) = m / time_{K2}(n)$

8.3 Output

The output must contain the following information:

SCALE
: Graph generation parameter

edgefactor
: Graph generation parameter

NBFS
: Number of BFS searches run, 64 for non-trivial graphs

construction_time
: The single kernel 1 time

min_time, firstquartile_time, median_time, thirdquartile_time, max_time
: Quartiles for the kernel 2 times

mean_time, stddev_time
: Mean and standard deviation of the kernel 2 times

min_nedge, firstquartile_nedge, median_nedge, thirdquartile_nedge, max_nedge
: Quartiles for the number of input edges visited by kernel 2, see TEPS section above.

mean_nedge, stddev_nedge
: Mean and standard deviation of the number of input edges visited by kernel 2, see TEPS section above.

min_TEPS, firstquartile_TEPS, median_TEPS, thirdquartile_TEPS, max_TEPS
: Quartiles for the kernel 2 TEPS

harmonic_mean_TEPS, harmonic_stddev_TEPS
: Mean and standard deviation of the kernel 2 TEPS. **Note**: Because TEPS is a rate, the rates are compared using **harmonic** means.

Additional fields are permitted. Algorithm output provides a high-level sample.

```
function output (SCALE, edgefactor, NBFS, kernel_1_time, kernel_2_time,
kernel_2_nedge)
  printf ("SCALE: %d\n", SCALE);
  printf ("edgefactor: %d\n", edgefactor);
  printf ("NBFS: %d\n", NBFS);
  printf ("construction_time: %20.17e\n", kernel_1_time);

  S = statistics (kernel_2_time);
  printf ("min_time: %20.17e\n", S(1));
  printf ("firstquartile_time: %20.17e\n", S(2));
  printf ("median_time: %20.17e\n", S(3));
  printf ("thirdquartile_time: %20.17e\n", S(4));
  printf ("max_time: %20.17e\n", S(5));
  printf ("mean_time: %20.17e\n", S(6));
  printf ("stddev_time: %20.17e\n", S(7));

  S = statistics (kernel_2_nedge);
  printf ("min_nedge: %20.17e\n", S(1));
  printf ("firstquartile_nedge: %20.17e\n", S(2));
  printf ("median_nedge: %20.17e\n", S(3));
  printf ("thirdquartile_nedge: %20.17e\n", S(4));
  printf ("max_nedge: %20.17e\n", S(5));
  printf ("mean_nedge: %20.17e\n", S(6));
  printf ("stddev_nedge: %20.17e\n", S(7));

  TEPS = kernel_2_nedge ./ kernel_2_time;
  N = length (TEPS);
  S = statistics (TEPS);
  S(6) = mean (TEPS, 'h');
  %% Harmonic standard deviation from:
  %% Nilan Norris, The Standard Errors of the Geometric and Harmonic
  %% Means and Their Application to Index Numbers, 1940.
  %% http://www.jstor.org/stable/2235723
  tmp = zeros (N, 1);
  tmp(TEPS > 0) = 1./TEPS(TEPS > 0);
  tmp = tmp - 1/S(6);
  S(7) = (sqrt (sum (tmp.^2)) / (N-1)) * S(6)^2;

  printf ("min_TEPS: %20.17e\n", S(1));
  printf ("firstquartile_TEPS: %20.17e\n", S(2));
  printf ("median_TEPS: %20.17e\n", S(3));
  printf ("thirdquartile_TEPS: %20.17e\n", S(4));
  printf ("max_TEPS: %20.17e\n", S(5));
  printf ("harmonic_mean_TEPS: %20.17e\n", S(6));
  printf ("harmonic_stddev_TEPS: %20.17e\n", S(7));
```

8.4 References

Nilan Norris, The Standard Errors of the Geometric and Harmonic Means and Their Application to Index Numbers, The Annals of Mathematical Statistics, vol. 11, num. 4, 1940. http://www.jstor.org/stable/2235723

9 Sample Driver

A high-level sample driver for the above routines is given in Algorithm driver.

```
SCALE = 10;
edgefactor = 16;
NBFS = 64;

rand ("seed", 103);

ij = kronecker_generator (SCALE, edgefactor);

tic;
G = kernel_1 (ij);
kernel_1_time = toc;

N = size (G, 1);
coldeg = full (spstats (G));
search_key = randperm (N);
search_key(coldeg(search_key) == 0) = [];
if length (search_key) > NBFS,
  search_key = search_key(1:NBFS);
else
  NBFS = length (search_key);
end
search_key = search_key - 1;

kernel_2_time = Inf * ones (NBFS, 1);
kernel_2_nedge = zeros (NBFS, 1);

indeg = histc (ij(:), 1:N); % For computing the number of edges

for k = 1:NBFS,
  tic;
  parent = kernel_2 (G, search_key(k));
  kernel_2_time(k) = toc;
  err = validate (parent, ij, search_key (k));
  if err <= 0,
    error (sprintf ("BFS %d from search key %d failed to validate: %d",
                    k, search_key(k), err));
  end
  kernel_2_nedge(k) = sum (indeg(parent >= 0))/2; % Volume/2
end
```

```
output (SCALE, edgefactor, NBFS, kernel_1_time, kernel_2_time,
kernel_2_nedge);
```

10 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

Author: Graph 500 Steering Committee
Date: 2010-10-03 16:15:14 EDT
HTML generated by org-mode 7.01h in emacs 22