Towards a Reproducible, Verifiable Parallel Graph Analysis Benchmark

Rob F. Van der Wijngaart
Intel Corporation
Software and Services Group

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

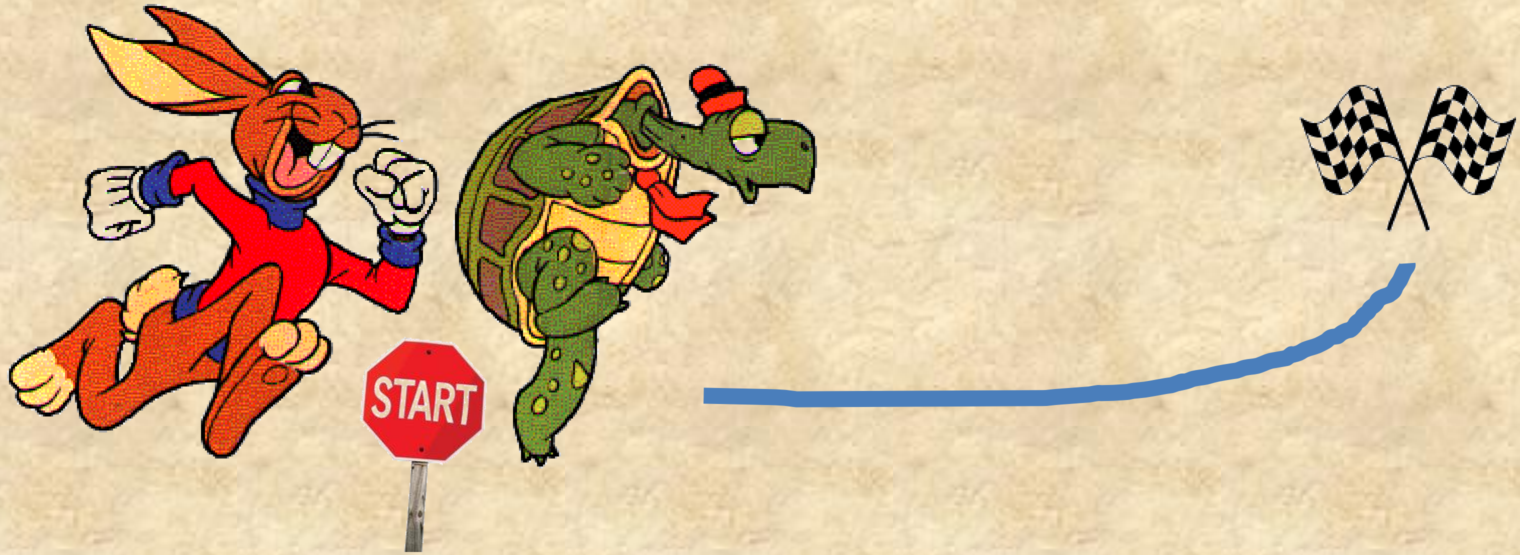
Notice revision #20110307

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
Configuration: Xeon E7-8870 , 4 CPUs, 40 cores, 64 GB DDR3, KMP_AFFINITY=scatter, 4K pages. Compiler: icc 11.1
For more information go to http://www.intel.com/performance

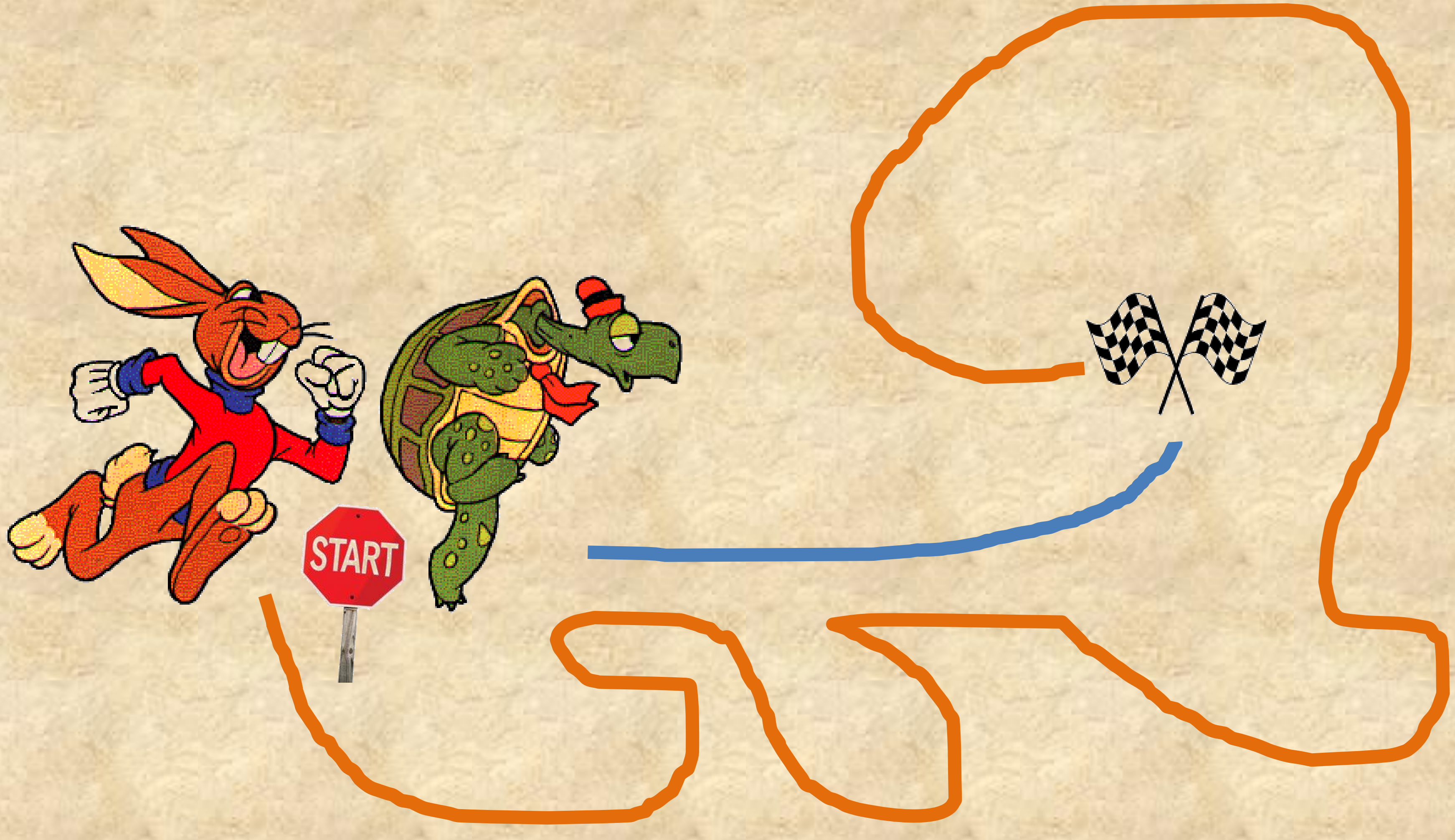
What is a benchmark?

Merriam-Webster: A standardized problem or test that serves as a basis for evaluation or comparison (as of computer system performance)

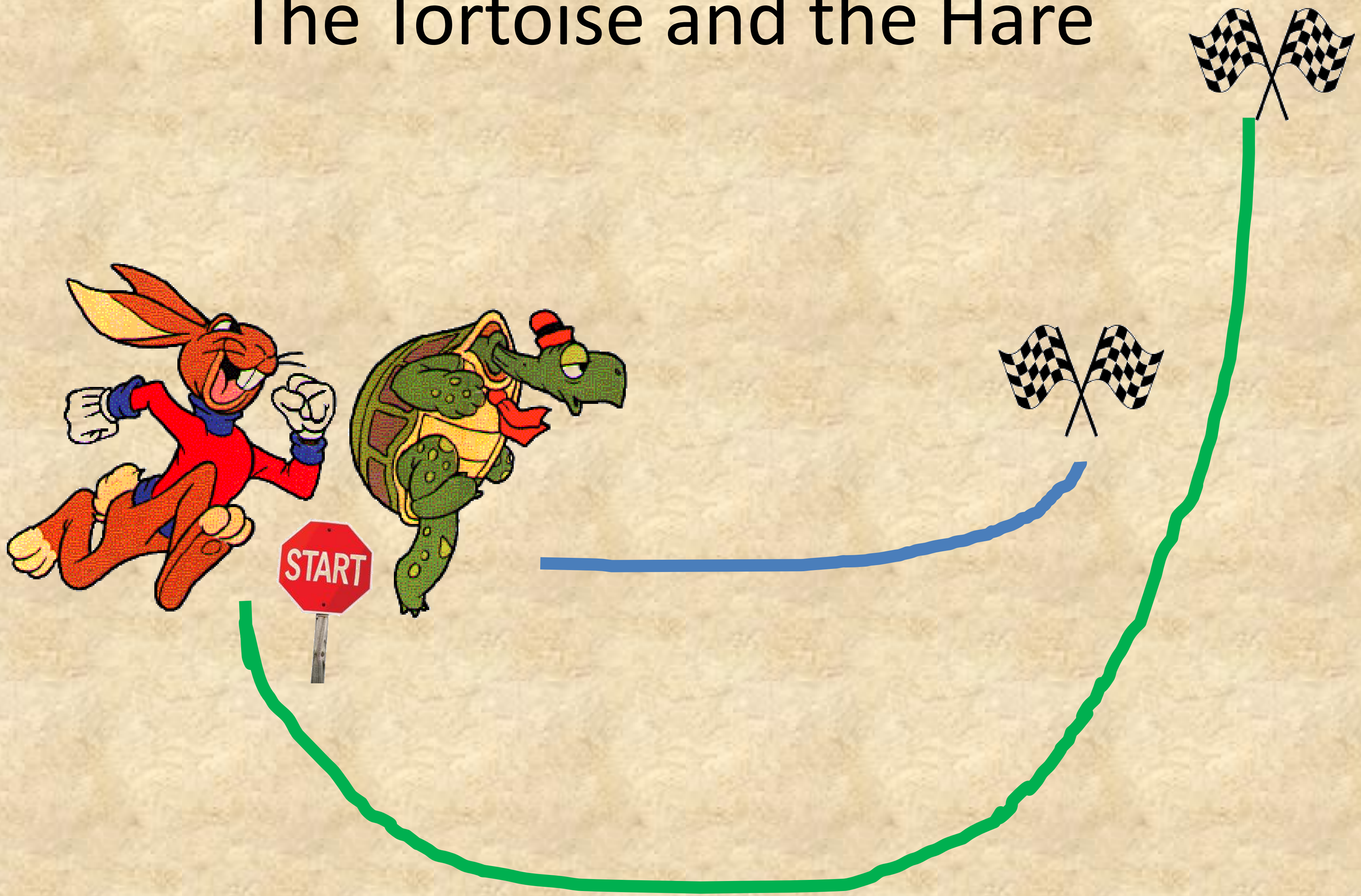
Aesop's Benchmark
The Tortoise and the Hare



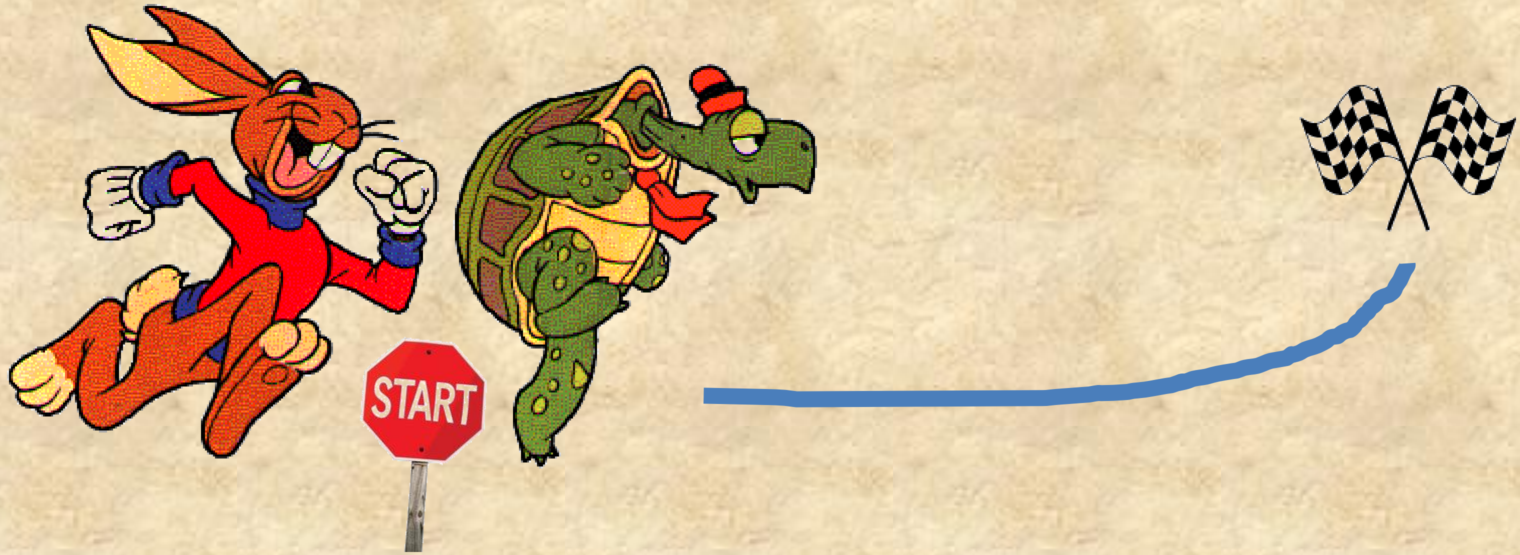
Aesop's Benchmark
The Tortoise and the Hare



Aesop's Benchmark
The Tortoise and the Hare



Aesop's Benchmark
The Tortoise and the Hare



Moral of the story:
competitors must complete the same task to be compared fairly

Computer Benchmark Learnings

- Many mistakes are unintentional:
  - Algorithmic “improvements”
  - Shortcuts: only test some results
  - Unwarranted compiler overrides

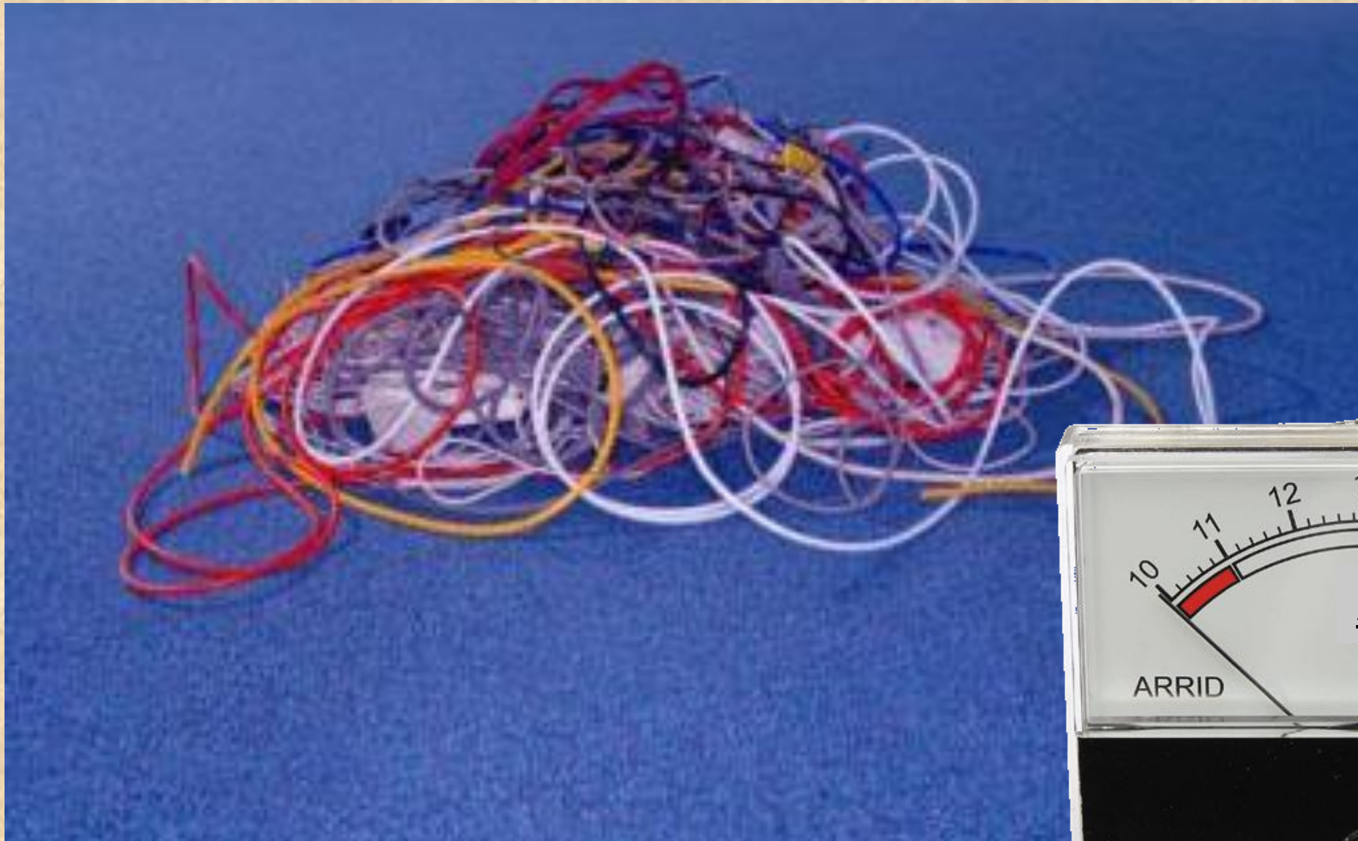
- Some mistakes are (semi-)intentional
- Most mistakes are eliminated with a good verification test—debugging aid!

Computer Benchmark Learnings

What makes a good verification test?

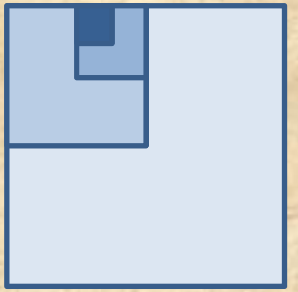
- Frugal; test should not be too onerous
  - Time(test) <(<) Time(problem)
  - Memory/disk(test) <(<) Memory/disk(problem)
- No false negatives; test should not be too tight
  - Legitimate algorithmic variation should pass
  - Don’t demand bitwise identical floating point computations

- No false positives; test should not be too lax
  - Mr. Bijk’s zero current test
  - HPCC RandomAccess table test
  - NAS Parallel Benchmarks Multi-Grid convergence test
  - S3D reaction rate test
  - SSCA2: Problem verified different from problem measured




10
11
12
13
14
15
16
A
ARRID
MA1016

Scalable Synthetic Compact Application 2 (SSCA2)



- Construct small-world directed graph
  - Use R-MAT for edge generation
  - Reject self edges (trivial) and duplicate edges (hardish)
  - Permute vertex indices to remove first-quadrant bias
  - Assign random, uniformly distributed weights to edges

- Compute Betweenness Centrality (BC) for all vertices
  - Use set of vertices as sources for Breadth First Search (BFS) in shortest path computations
  - Skip graph edges whose weight is divisible by 8
  - Measure Traversed Edges Per Second (TEPS)

What is betweenness centrality?

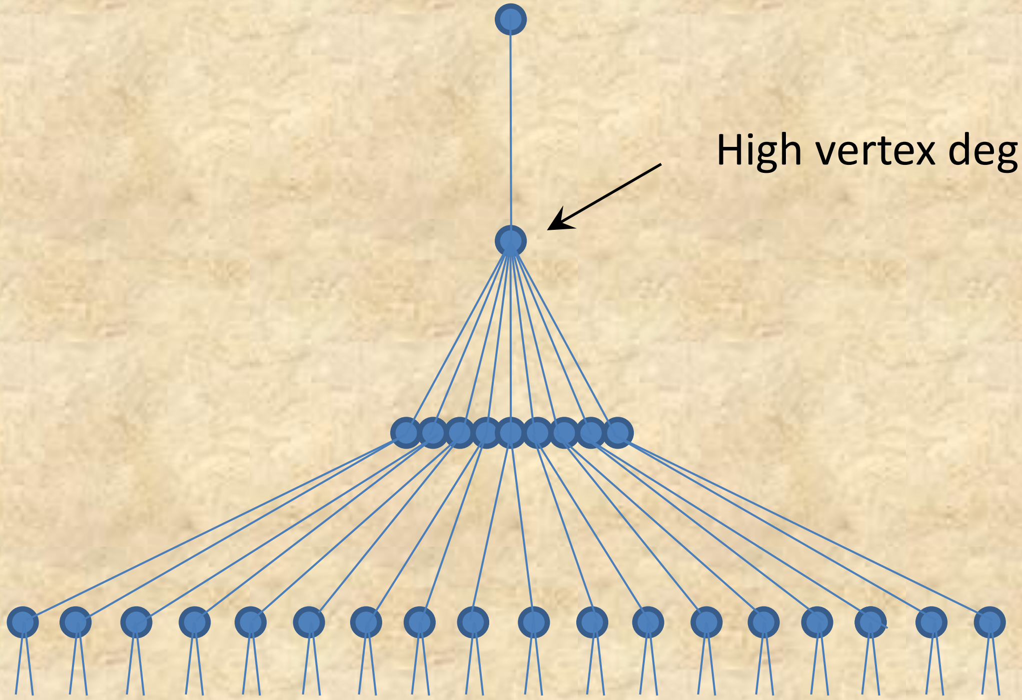
Definitions:

- $s, v, t \in V$ (set of vertices of the graph)
- σ_{st}: # all shortest paths from s (source) to t
- $\sigma_{st}(v)$: # shortest paths from s (source) to t, passing through v
- dependency $\delta_{st}(v) = \sigma_{st}(v) / \sigma_{st}$
- Betweenness centrality: $BC(v) = \sum_{s \neq v \neq t} \delta_{st}(v)$

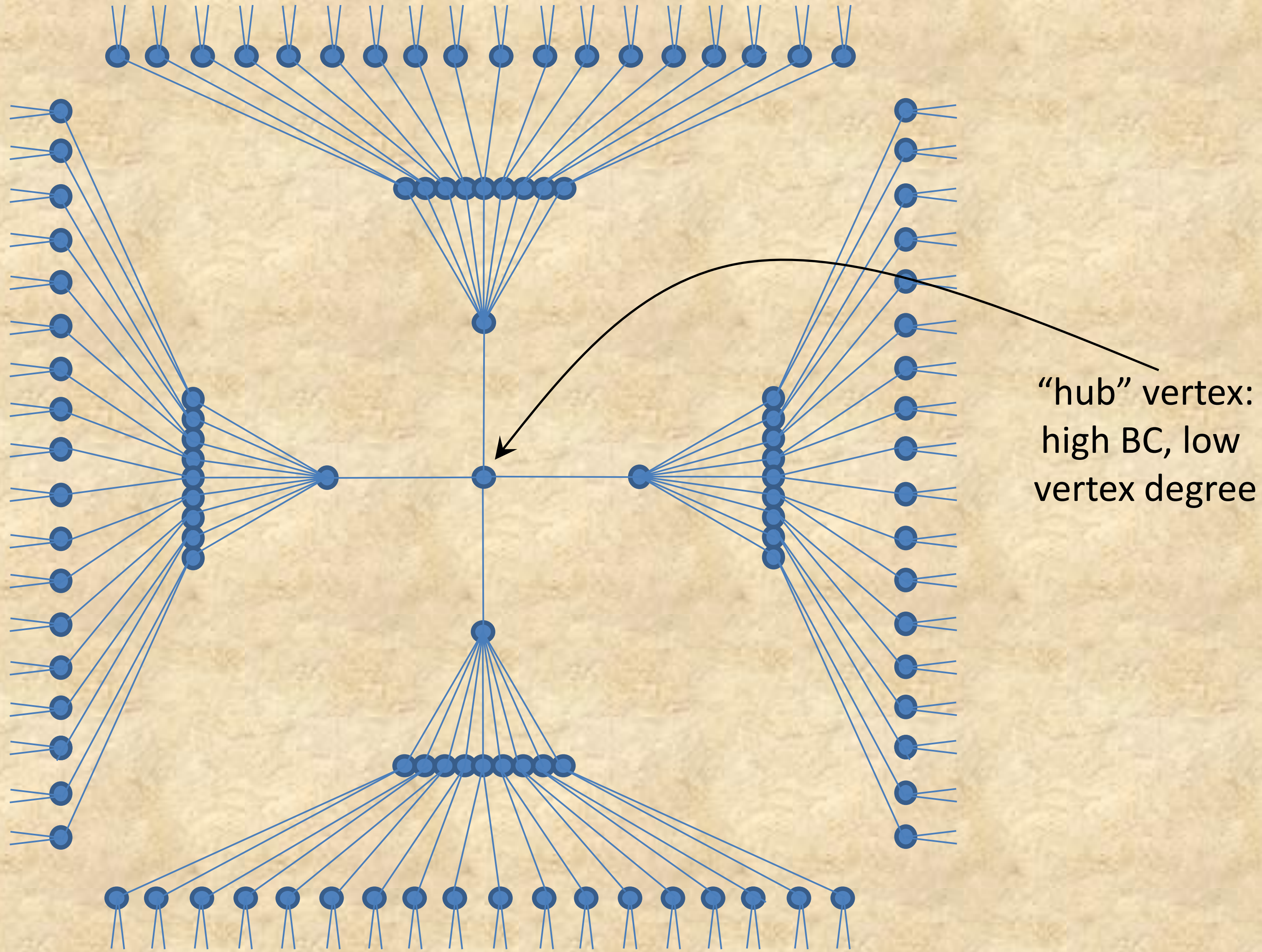
Efficient recent implementations use BFS. Typically two distinct steps:

1. Do BFS from subset of source vertices s; assign to each other vertex v the distance and shortest paths count from s
2. Reverse traversal, accumulate $\delta_{st}(v)$ (step 1) in BC, using simple recursive relation

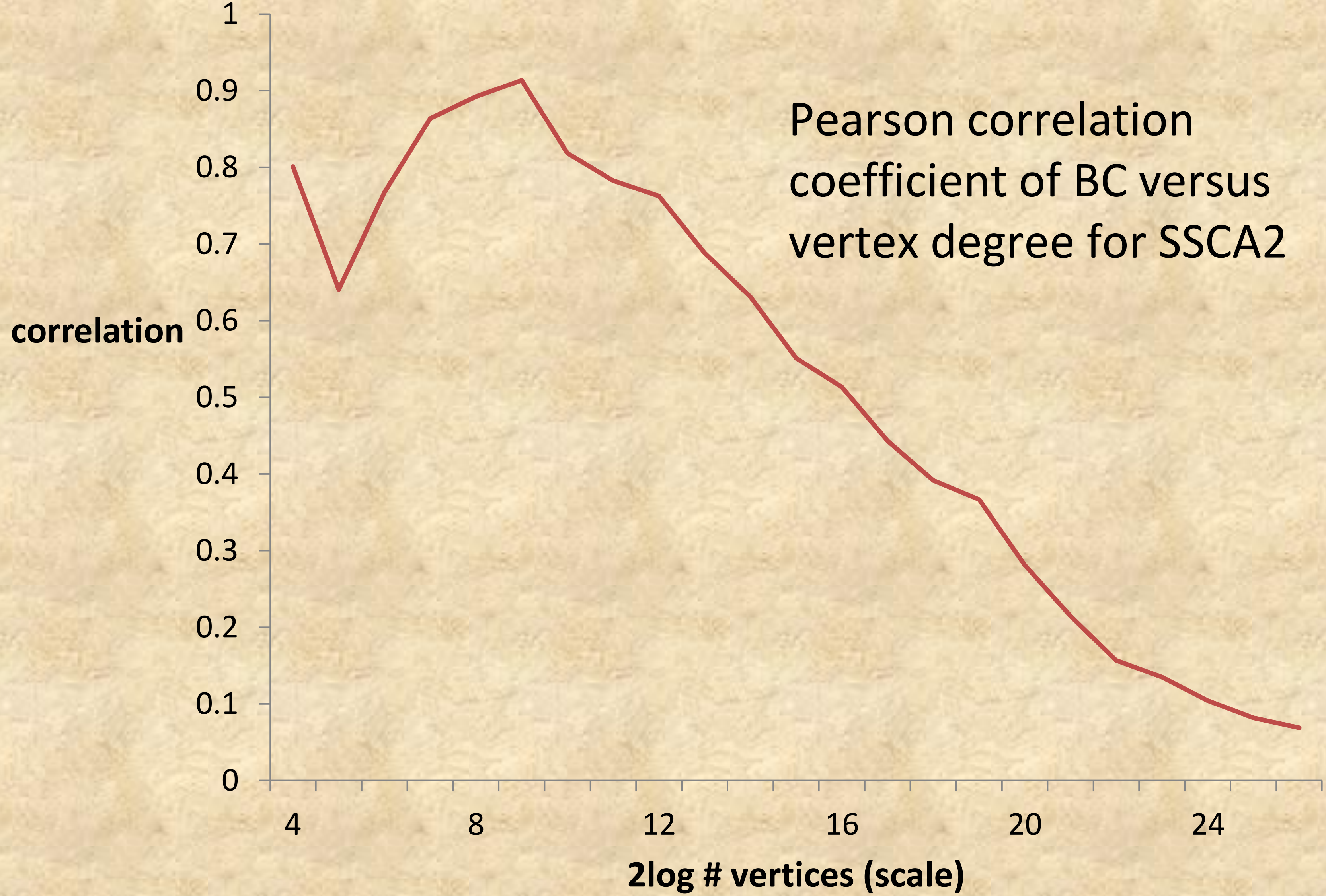
BC ≈ vertex degree?



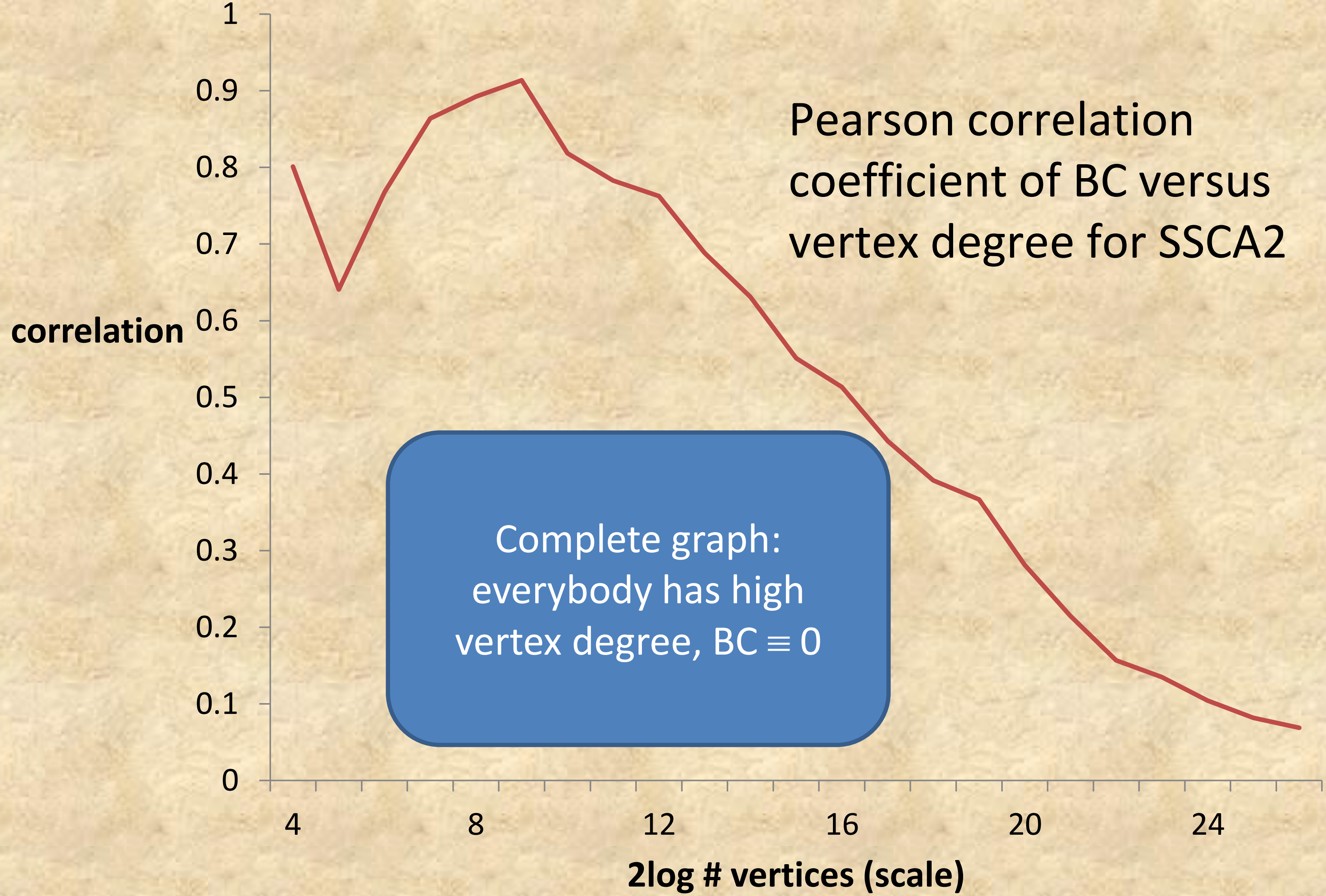
BC ≈ vertex degree?



BC ≈ vertex degree?



BC ≈ vertex degree?



Pearson correlation coefficient of BC versus vertex degree for SSCA2

Scalable Synthetic Compact Application 2 (SSCA2)

- Don’t verify, because parallel implementation of
  1. R-MAT gives different results for different numbers of threads (no races)
  2. Permutation of vertex indices gives different results for different numbers of threads, and has races
  3. Selection of source vertices gives different results for different numbers of threads, and has races
  4. Filtering of edge duplicates/weights assignment has races

- Don’t verify, because it can’t be done for graph sizes that have never been generated before

1. R-MAT gives different results for different numbers of threads

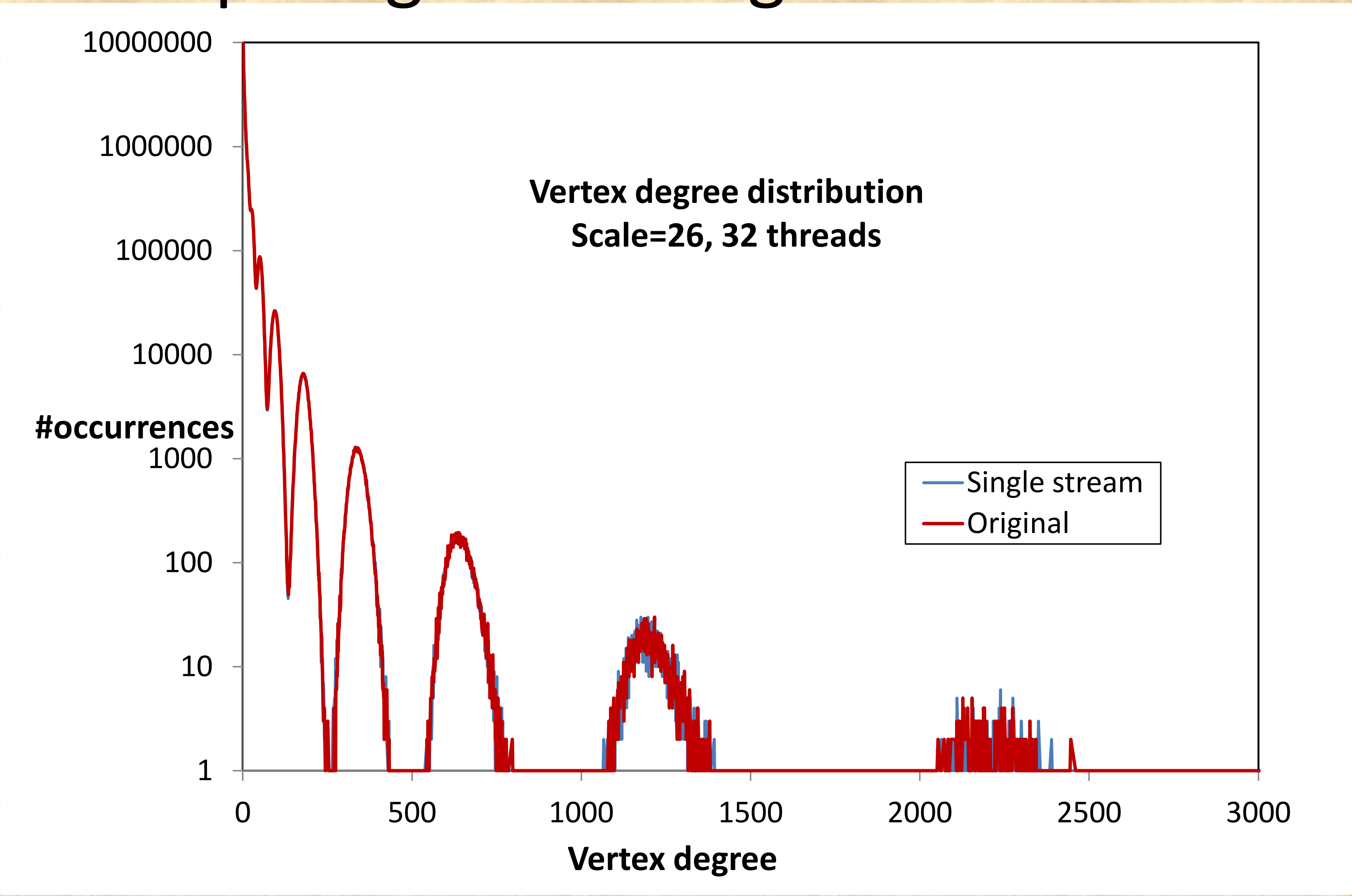
- Each thread uses independent stream of (Pseudo) Random Numbers (PRNs) to create edges
- Fix: use single stream of PRNs, and distribute sub-streams evenly among threads
- Requires: jump ahead in single stream in *O*(logN) time for jump N
  - Easy for multiplicative Linear Congruential Generators: $x_k = (a*x_{k-1})\%m$, $x_0 = s$, but usually short period (e.g. NAS Parallel Benchmarks: 2^{46})
  - Harder for mixed LCGs with longer period (2^{64}) : $x_k = (a*x_{k-1} + c)\ \%m$, but can be done

$$x_k = (a^k s + c(\sum_{i=0}^{k-1} a^i)) \pmod{m} = (a^k s + c(a^k - 1)/(a-1)) \pmod{m}$$

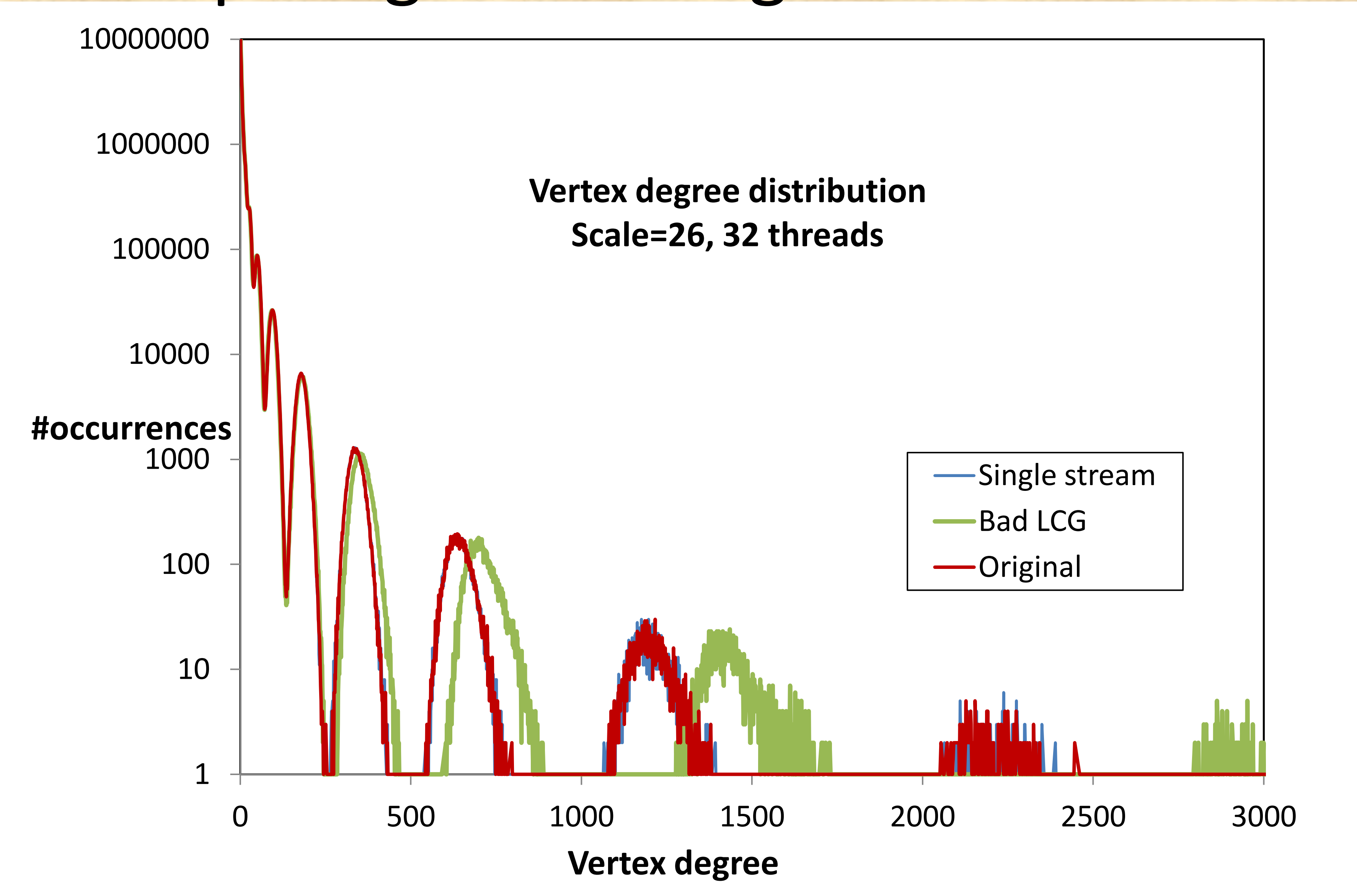
- RMAT needs 5*n*log(n) PRNs to create n edges; LCG with period 2^{64} can generate 2^{55} edges before duplication starts.
- Current max for Graph500¶: $n=2^{42}$ (Intrepid, IBM Blue Gene/P; 32,768 nodes; 131,072 cores). Storage: 70TB.

¶source: www.graph500.org, June 2011

Comparing vertex degree distributions



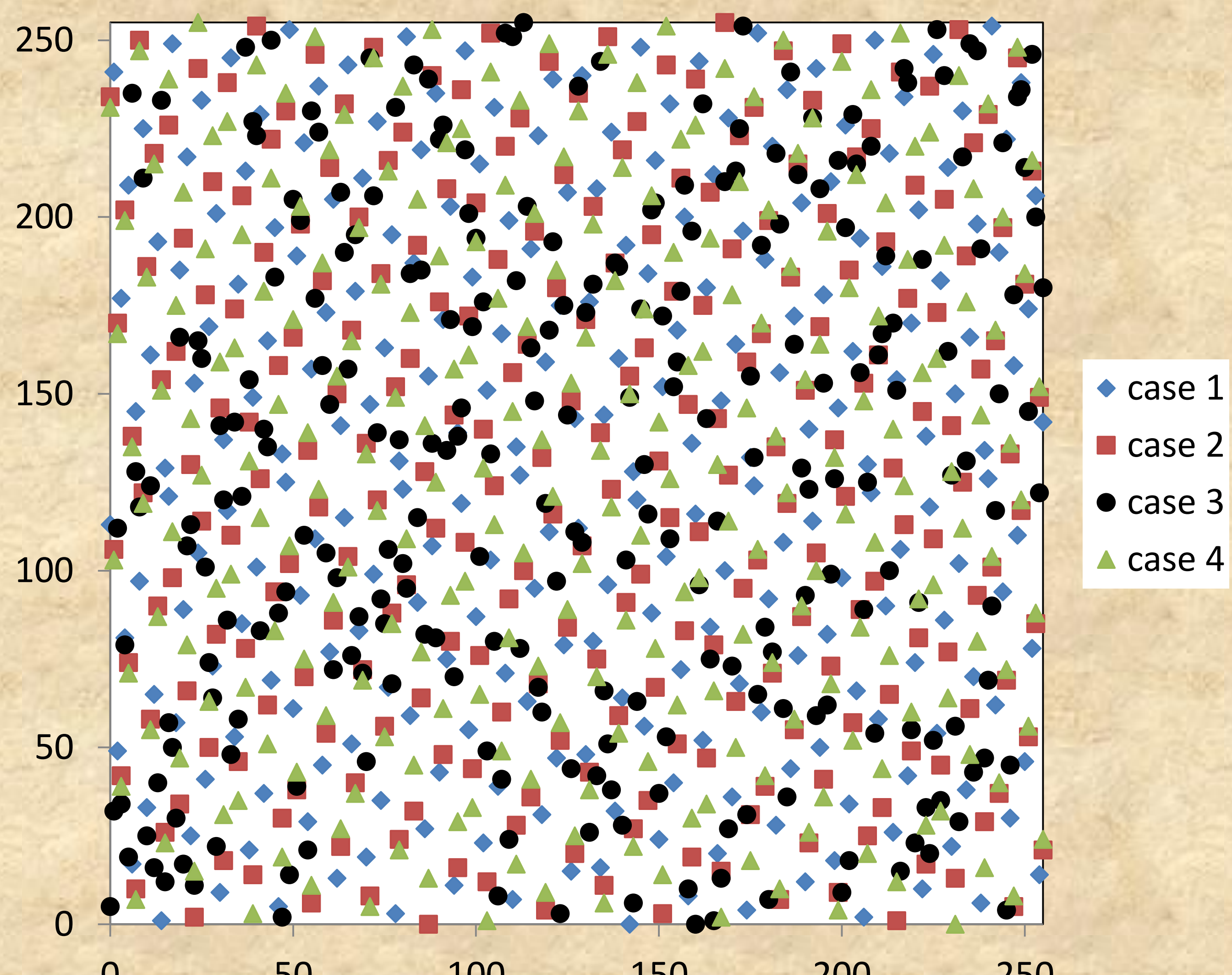
Comparing vertex degree distributions



2. Vertex permutation not reproducible

- Original implementation for n vertices:
  - Each thread generates (n/nthreads) PRNs independently
  - Each pair (i,PRN(i)) defines vertex swap pair
  - If any other thread tries to swap overlapping pair simultaneously, abandon swap (locks); race, even if no swap ever abandoned (swaps don't commute)
- Fix:
  - Explicit permutation function F, mapping vertex i to vertex F(i)
  - F(i) = bit-reverse(i) XOR pseudo-random mask
- The good, the bad:
  - Deterministic, cheap to compute (vectorizable), no locks (OpenMP) or communication (MPI), good scattering.
  - Can be reverse engineered. More regular than random swaps

Permutation functions



3. Source vertex selection not reproducible

- Uses permutation of vertex indices
- Fix:
  - Explicit permutation function F, mapping vertex i to vertex F(i)

Intermezzo

After introduction of single stream of PRNs in R-MAT and of explicit vertex permutation function:

- Verification value ΣBC identical from run to run, for different numbers of threads, if duplicate edges allowed
- ΣBC different from run to run if duplicate edge filtering turned on
- Conclusion: race in duplicate edge filtering ✓

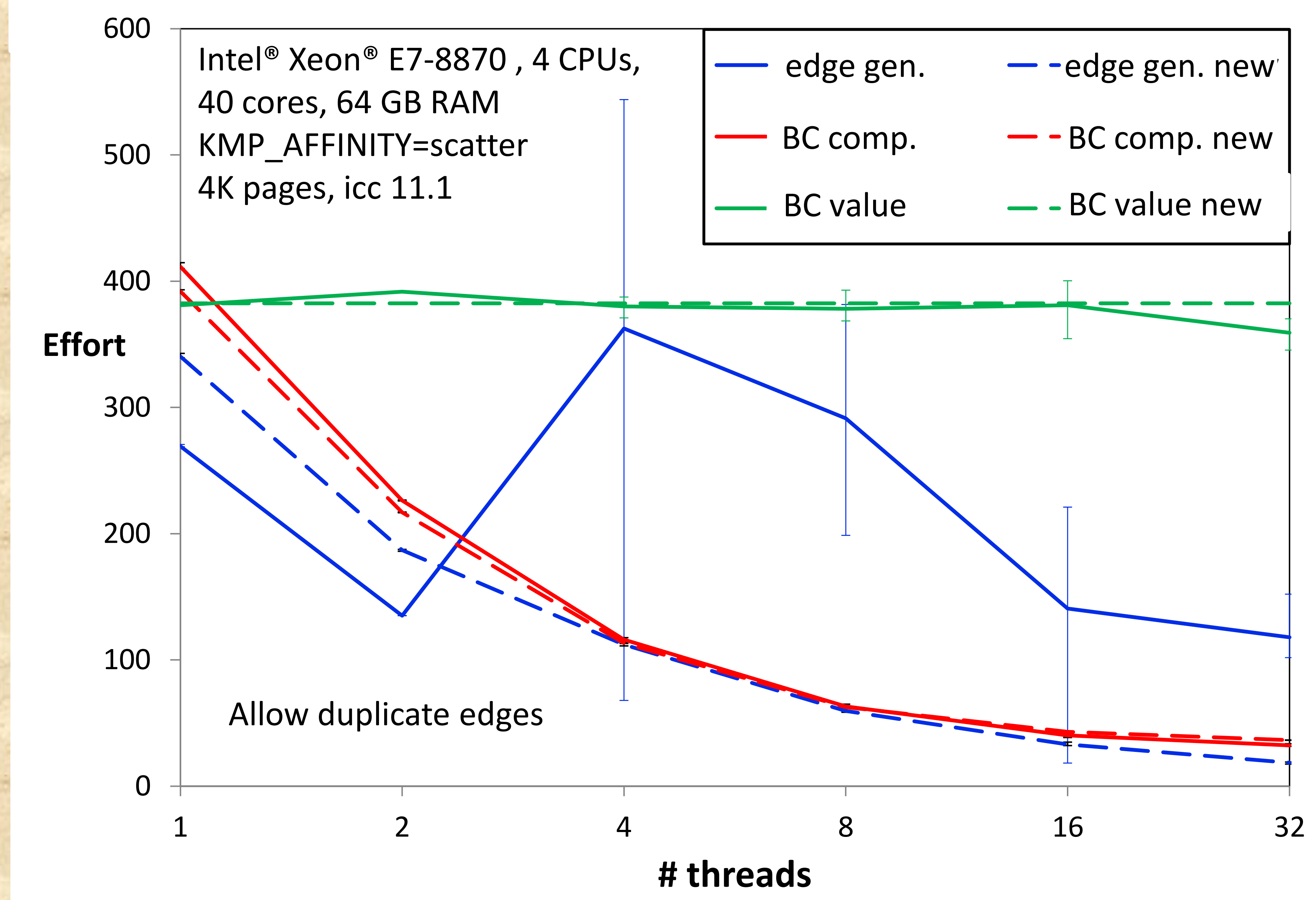
4. Edge weight assignment has races

- BFS ignores edges if: weight (mod 8) = 0
- Weights assigned uniformly, randomly, using PRNs
- PRNs based on edge sequence number (and thread ID)
- If duplicate edges filtered:
  - Any specific edge inserted only once
  - First thread to arrive inserts edge in its local edge list
  - Local edge lists merged through concatenation
- Result: edge weights experience race

4. Edge weight assignment has races

- Fix1: sort edges before assigning weights
  - O(nlog(n)) algorithmic complexity
- Fix 2: remove duplicates afterwards, keeping only first occurrence
  - need to repeat until enough unique edges added
  - O(nlog(n)) algorithmic complexity
  - higher programming complexity
- Fix 3: ✓
  - employ physical parameters (end points) to assign weight
  - Simplest solution: seed = start_id+end_id; apply LCG once
  - Number of ignored edges very close to required $1/8^{th}$.
  - Uniform?

Experimental results, scale=24, 7 runs



Conclusions I

SSCA2:

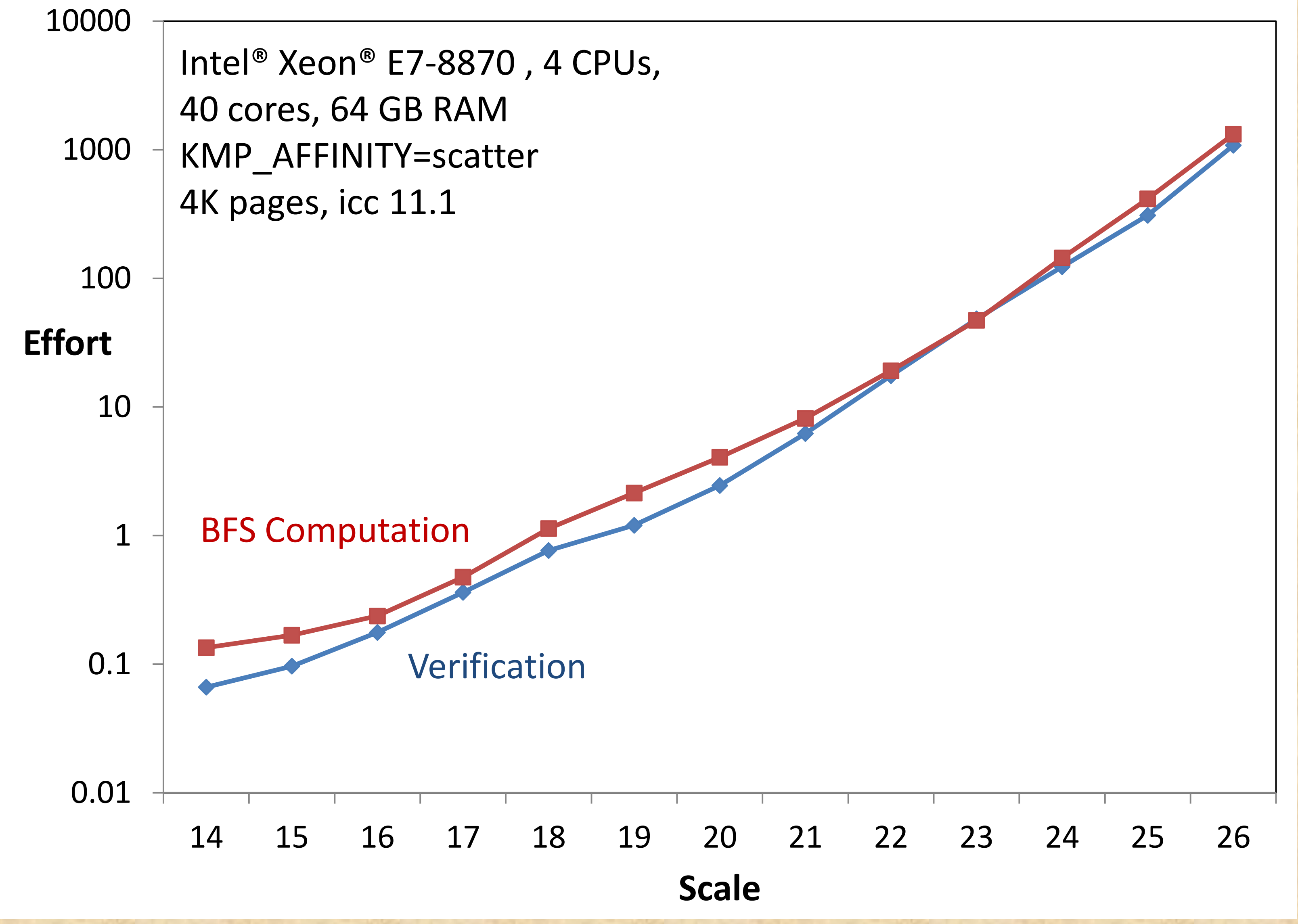
- Can be made reproducible *without reading graph from file*
- Can be made verifiable
  - Anyone running benchmark must:
    - match prior results, or
    - publish newly obtained result
- Has been used with success after modifications
  - Very quick debugging of performance optimizations
  - Original runtime variations ≈ optimization gains
- Has some open issues
  - Vertex permutation sufficiently random?
  - Weight assignment sufficiently uniform?

Graph500

- Does BFS like SSCA2, but no Betweenness Centrality; Single-Source Shortest Paths being considered
- PRNG: Multiple Recursive Generator for random number generation $x_k = (a_1 * x_{k-1} + \ldots + a_n * x_{k-n}) \% m$ with jump ahead
- Permutation:
  - Parallel Random Sort based permutation (v1.x¶)
  - Suggested bit reversal in October 2010
  - Moved to bit reversal-based permutations for vertex scrambling in April 2011 (v2.x)
- No edge weights assignment or filtering

¶graph not reproducible; by design or because of races

Benchmark + verification time



Conclusions II

Graph500:

- Fixes reproducibility ills of SSCA2
- Should spend less time verifying solution
- Should be augmented with more demanding kernel à la BC

Acknowledgments

Thanks to Kamesh Madduri (PSU) and Roger Golliver (Intel) for explaining the structure of SSCA2 and helping to track down race conditions.