GAIL: The Graph Algorithm Iron Law

Scott Beamer Krste Asanović David Patterson
Electrical Engineering & Computer Sciences Department
University of California, Berkeley, California
{sbeamer,krste,pattrsn}@eecs.berkeley.edu

ABSTRACT

As new applications for graph algorithms emerge, there has been a great deal of research interest in improving graph processing. However, it is often difficult to understand how these new contributions improve performance. Execution time, the most commonly reported metric, distinguishes which alternative is the fastest but does not give any insight as to why. A new contribution may have an algorithmic innovation that allows it to examine fewer graph edges. It could also have an implementation optimization that reduces communication. It could even have optimizations that allow it to increase its memory bandwidth utilization. More interestingly, a new innovation may simultaneously affect all three of these factors (algorithmic work, communication volume, and memory bandwidth utilization). We present the Graph Algorithm Iron Law (GAIL) to quantify these tradeoffs to help understand graph algorithm performance.

1. INTRODUCTION

With growing interest in applications using graph algorithms, there has been a corresponding growth in graph processing acceleration research. This interest in improving graph processing performance is not confined to any one area and is ongoing at all layers of the computational stack (algorithms, implementation, frameworks, and hardware platforms). As each new innovation is shown to be beneficial by demonstrating a speedup, it is often unclear what causes the speedup since many of the recent innovations span multiple layers of the stack.

Execution time is the most important metric, but is unfortunately not instructive on its own. It allows us to quantify how much faster one execution is than another, but for those executions to be comparable using time, many parameters should be kept the same (input graph and graph kernel). Using a rate, such as traversed edges per second (TEPS) [11], gives an idea of execution speed and the ability to compare executions using different input graphs. However, TEPS can be misleading, especially if the ways edges are counted for TEPS differ. Whether undirected input edges are counted as one edge or two directed edges changes TEPS by a factor of two, which is often greater than the margin of improvement for a new optimization. A new algorithm could substantially reduce the number of traversed edges, allowing for a lower TEPS score to actually represent a faster execution.

IA^3 2015, November 15-20, 2015, Austin, TX, USA
Copyright 2015 ACM 978-1-4503-4001-4/15/11 ...$15.00
DOI: http://dx.doi.org/10.1145/2833179.2833187

The shortcomings of TEPS demonstrates that a single metric cannot be used, and instead it should be complemented by other metrics. For example, reporting TEPS would be better if the number of traversed edges was also reported. Our Graph Algorithm Iron Law (GAIL) does this and more by also including memory requests since the memory system is often the biggest architectural bottleneck. With GAIL, we concisely factor out the differences between algorithm, implementation, and hardware platform. In the rest of this work, we describe GAIL in detail and demonstrate its utility with two case studies considering the impact of algorithm, implementation, input graph, and hardware platform.

2. GRAPH ALGORITHM IRON LAW

Algorithms continue to be the most important factor for performance, since a better algorithm can typically outperform a better software implementation or hardware platform. Complexity analysis, the classic tool for evaluating algorithms, can sometimes be less instructive for high-performance graph processing evaluations. Worst-case analysis often yields overly pessimistic performance bounds that differ greatly from the common case. Additionally, the amount of algorithmic work for many advanced graph algorithms depends heavily on the input graph topology, which is often difficult to quantify. Furthermore, as the field matures, many innovations will be implementation optimizations and not algorithmic optimizations, but these innovations would appear to provide no improvement if compared analytically. A good alternative in these challenging scenarios is to empirically measure the amount of algorithmic work.

Once the impact of the algorithm is understood, it is important to understand the potential and limitations of the hardware platform. Depending on the input graph and the algorithm, there can be a variety of hardware performance bottlenecks; however, graph algorithms are typically memory bound and not compute bound. Contrary to popular belief, the memory bottleneck is due to memory latency much more often than it is due to memory bandwidth because memory bandwidth is typically underutilized [4]. A modest cache hit rate can cause cache misses to be so rare that an insufficient number of them fit into the instruction window to fill all of the outstanding memory request slots. With an insufficient number of outstanding memory requests, the processor is unable to fully utilize the memory bandwidth. This leads to a common tradeoff: improving cache hit rates results in memory bandwidth becoming further underutilized. Other factors (branch mispredictions, low instruction-level parallelism, and temporal variation) can also hinder memory bandwidth, but good locality is often the most impactful. Given all of this, the most important architectural features affecting graph processing execution are cache effectiveness and memory bandwidth utilization.

Taking the insights we have learned above, we present a simple model to understand graph algorithm performance. Analogous to how the Iron Law of CPU performance [9, 16] factors execution time into the product of the number of instructions executed, cycles per instruction, and the cycle time, our Graph Algorithm Iron Law (GAIL) factors execution time into the product of the number of traversed edges, the number of memory requests per traversed edge, and the inverse bandwidth[1]:

$$\frac{\text{time}}{\text{kernel}} = \frac{\text{edges}}{\text{kernel}} \times \frac{\text{memory requests}}{\text{edge}} \times \frac{\text{seconds}}{\text{memory requests}}$$

This model combines the three characteristics we find most relevant for graph algorithm performance: algorithmic efficiency, cache effectiveness, and memory bandwidth utilization. To measure the amount of algorithmic work, we use edges examined instead of vertices examined because the amount of work per edge is roughly constant, while the amount of work per vertex can vary dramatically (it is often a function of its degree). With the help of execution time and edges examined, we break down communication volume to DRAM (memory requests) into two instructive rates that measure cache effectiveness (second term) and memory bandwidth utilization (third term).

Although GAIL may appear simple, there are some noteworthy elements hidden within. The product of the first two terms is the number of memory requests per kernel, which is proportional to the amount of data moved from DRAM. The product of the second term and the third term is seconds per traversed edge, which is the inverse of TEPS from Graph500. Beyond runtime, the only additional data that needs to be collected for a GAIL analysis is the number of memory requests and the number of traversed edges.

When counting memory requests, GAIL users should include prefetches in addition to cache misses.[2] Although including prefetches can be more arduous than collecting only cache misses, including prefetch traffic is worthwhile because it increases the accuracy of GAIL. When the prefetches are beneficial, accounting for that traffic prevents underreporting the number of memory requests necessary to complete the kernel. When the prefetches are not helpful, most modern processors are designed to decrease the amount of hardware prefetching to keep these unneeded prefetches from being too detrimental to performance. Unused prefetches will consume memory bandwidth, but a reasonably designed system will give cache misses priority over the hardware prefetcher. Throughout our experiments on IVB (evaluation system in Section 3), we never observe the hardware prefetcher consuming all of the remaining memory bandwidth [4].

[1]The last term (seconds per memory request) is best thought of as inverse of bandwidth rather than the average time per memory request. If there is more than one memory request outstanding, this metric will understandably become smaller than the actual average memory latency.

[2]We do not count cache hits since we assume out-of-order execution can hide their latency.

Description	$\lvert V\rvert$ (M)	$\lvert E\rvert$ (M)	References
Kronecker Synthetic	128.0	2,111.6	[11, 15]
Web Crawl of .sk	50.6	1,949.4	[7]
Roads of USA	23.9	58.3	[8]

Table 1: Input graphs with abbreviations bolded

GAIL can be instructive to a variety of potential users. Algorithm designers can use it to demonstrate the work reduction (fewer edges examined), and doing this empirically can be helpful when it is difficult to quantify analytically. A framework developer can demonstrate the efficiency of their implementation by achieving either higher memory bandwidth or fewer memory requests per traversed edge. Even a hardware platform designer could use GAIL to understand the tradeoffs in the design space such as those between core count, thread count, cache sizes, and memory bandwidth, for example.

3. CASE STUDIES USING GAIL

For any evaluation, measurements should be taken to attempt to explain the results, and some of the metrics used may be specific to that study. GAIL is useful as a simple way to verify if the improvements reduce the amount of algorithmic work, increase locality, or improve memory bandwidth utilization. To demonstrate the utility of GAIL, we perform two case studies to investigate tradeoffs in software implementation and changes to the hardware platform.

For our case studies, we use the breadth-first search (BFS) implementation from the GAP Benchmark Suite's reference code [2, 10]. We select three input graphs for size and topological diversity (Table 1). A proper evaluation will use more input graphs and graph kernels, but we believe these brief case studies can still demonstrate GAIL's utility. All of the results are averaged across 64 searches that each start from a randomly selected non-zero degree vertex.

We use a dual-socket Intel Ivy Bridge server (IVB) with E5-2667 v2 processors, similar to what one would find in a datacenter. Each socket contains eight 3.3 GHz two-way multithreaded cores and 25 MB of last-level cache (LLC). Using a synthetic microbenchmark we are able to achieve a maximum memory bandwidth of 1210 M random memory requests per second (77.4 GB/s) which corresponds to an inverse bandwidth of 0.826 ns/memory request [4]. We use Intel PCM to access hardware performance counters [13].

3.1 Understanding Implementation Tradeoffs

We first use GAIL to examine the impacts of algorithmic innovations and implementation optimizations on BFS on the kron graph, and Table 2 shows the GAIL metrics (last three columns) as well as the raw data that produces them (second, third, and fourth columns). As a baseline, we implement a classic top-down approach (TD) within the reference code that uses a shared queue for the frontier with thread-local buffers to reduce false sharing for vertex ap-

Ver.	Time (s)	Memory Requests (M)	Traversed Edges (M)	$\frac{\text{Memory Requests}}{\text{Edge}}$	$\frac{\text{ns}}{\text{Memory Request}}$
TD	3.964	3,292.88	4,223.22	0.780	1.204
BM	2.255	1,024.31	4,223.22	0.243	2.201
DO	0.424	209.50	183.78	1.140	2.022

Table 2: BFS on kron with 16 cores

pends. As expected, the number of traversed edges is twice the number of input edges since each undirected input edge is traversed once in both directions. The cache provides some benefit as the number of memory requests per traversed edge is less than one. From the last GAIL term (1.204 ns / memory request) we can see that the execution uses 68% of the platform's memory bandwidth when compared to the maximum we achieve with the microbenchmark.

We improve upon the performance of top-down with the simple optimization of using a bitmap to reduce the number of reads to the parent array (*BM*) [1]. The bitmap indicates if a vertex has already been visited, so the parent array only needs to be examined in the rare case the bitmap indicates a vertex is unvisited. The bitmap is 32× smaller than the parent array, so it is more likely to stay in the cache. The bitmap optimization does improve performance relative to top-down (1.75× speedup), but from the GAIL metrics it is clearly not an algorithmic improvement since it traverses the same number of edges. The benefit of the bitmap optimization is visible in the reduction in memory requests per edge, as fewer edge traversals miss in the LLC because fewer edge traversals need to check the parent array. In spite of utilizing less memory bandwidth (higher ns per memory request), bitmap is still faster than top-down because of the larger reduction in memory requests per traversed edge. The GAIL metrics easily decompose this beneficial tradeoff of one metric (cache locality) improving by more than another metric (memory bandwidth utilization) worsens.

As an example of an algorithmic optimization, we use the direction-optimizing implementation (*DO*) included in the benchmark suite [3]. The direction-optimizing approach provides a 5.3× speedup over the BM implementation, and using GAIL we see this is due to algorithmic improvements since it traverses 23× fewer edges. The speedup in execution time is substantially less than the reduction in traversed edges because DO has worse cache locality as visible with GAIL by the increase in the number of memory requests per traversed edge.

3.2 Understanding Impact of HW Platform

To demonstrate the utility of GAIL for evaluating hardware platforms, we vary the number of cores our workload uses. By varying the number of cores we allocate, we are not only increasing the potential computational throughput, but we are also increasing the potential for cache thrashing in the LLC and increasing synchronization overheads due to an increased number of participants. For these BFS strong scaling measurements, we use two new graphs (road and web) in addition to kron.

Figure 1 shows the results of the second two GAIL terms. We do not visualize the first term (traversed edges) since we use the same implementation for all three input graphs so

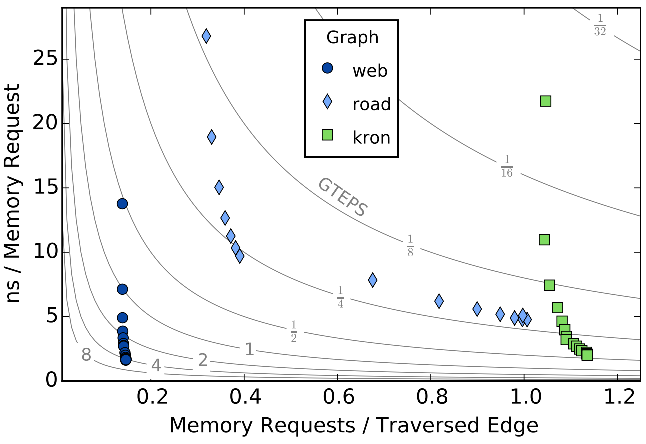

Figure 1: Strong scaling BFS from 1 core (upper left) to 16 cores (lower right). GTEPS (billion traversed edges per second) shown by contours. Single-core point for road (0.316, 46.62) clipped by view.

the number of traversed edges is the same. In Figure 1, moving down the graph represents utilizing more memory bandwidth. The contours represent traversed edges per second and moving towards the origin represents better throughput. Since the number of traversed edges is constant (for a given input graph and start vertex), moving towards the origin represents a speedup, and our BFS implementation generally delivers great speedups on all three graphs.

Traversing the web graph achieves over a 10× speedup with 16 cores. As more cores are used, they are able to generate more memory requests (lower on graph) and this improves performance because this does not result in excess memory traffic (same number of memory requests per edge shown by little horizontal change). Traversing the input graph kron does move extra data, as using additional cores results in an increase in the number of memory requests per traversed edge (curving to the right). In spite of this extra communication, our BFS implementation on kron still achieves a greater than 11× speedup due to the extra cores successfully utilizing more memory bandwidth.

Traversing the road graph demonstrates less parallel scalability which is in large part due to road's high diameter and small size. Since it is a high diameter graph, the direction-optimizing implementation will never switch into the bottom-up approach and will perform the entire search top-down. Since the graph road has fewer edges and a high diameter, there is less work per step and more steps (more synchronizations), further complicating parallel scalability. Using only one socket (8 cores), achieves a 5× speedup by utilizing more bandwidth than it moves additional data (close to vertical). Once traversing the road graph starts using the other socket, it actually goes slower (elbow to the right). With the help of GAIL, we can clearly see the cause is a sharp increase in the number of memory requests per edge (moving right) without much increase in the amount of utilized memory bandwidth (small vertical change). This is an indication to the hardware designer that the cache is likely thrashing and an indication to the software implementor that the graph may be too small to continue strong scaling on this platform.

4. FREQUENTLY ASKED QUESTIONS

In talking to others about using GAIL, some questions have arisen:

Q: What if GAIL users have different definitions for what constitutes a traversed edge?
A: Different traversed edge definitions will of course change the numerical results of GAIL, but by whatever factor the number of traversed edges is scaled will also inversely scale the number of memory requests per traversed edge. For example, deliberately undercounting traversed edges to appear algorithmically better will result in appearing to perform more memory requests per edge (worse). This concern should generally not be problematic since what constitutes a traversed edge is typically unambiguous and most evaluations will be performed by the same evaluator.

Q: Are the inputs to GAIL (time, traversed edges, and memory requests) the only important metrics for graph algorithm performance?
A: No, but we believe those to be the three most important. Branch mispredictions can definitely hinder performance [12]. Considering the number of instructions per traversed edge is another interesting metric for implementation efficiency, but we find it to be less instructive than memory requests per edge. Instruction throughput for graph algorithm execution is much more likely to be stalled by waiting on memory requests than a lack of available function units to execute ready instructions [4].

Q: Could GAIL be modified to consider energy?
A: Yes. Replacing time with energy in the GAIL equation results in an interesting new term: joules per memory request. Unfortunately, this modification to the GAIL equation may be less instructive due to tradeoffs between dynamic power and static power [6].

Q: Does GAIL work for distributed memory or semi-external memory implementations?
A: GAIL is designed for single-node shared memory systems since its metrics focus on the platform's most impactful architectural features (cache utility and memory bandwidth). For other platforms, substituting memory requests within GAIL for the other platform's most important bottleneck could make a similarly instructive equation. For example, for distributed memory (cluster) implementations [5], counting network packets will be more helpful. For semi-external memory (SSD/hard drive) implementations [14], counting blocks read from storage could be more useful.

Q: Does GAIL work for non-traditional architectures (those without caches)?
A: Yes. The Cray XMT is such an architecture, with a high thread count and no processor caches [17]. Even without the cache acting as a filter, the number of memory requests and the rate they execute can vary substantially. For example, different algorithms could result in different numbers of memory requests per traversed edge. Different implementations or compiler optimizations could change whether variables are kept in architectural registers or re-read from memory.

Q: Does GAIL work for all algorithms?
A: GAIL is designed for graph algorithms and it only requires a notion of traversing an edge. For algorithms that do not operate on graphs, we believe other analogous iron laws could be defined.

5. CONCLUSION

More specialized metrics should be a part of many graph processing evaluations, but GAIL provides a simple starting point by factoring out the tradeoffs between algorithmic work (traversed edges), cache locality (memory requests per traversed edge), and memory bandwidth utilization. If widely adopted, GAIL could allow for concise comparisons of approaches with metrics already familiar to the community. Fundamentally, GAIL is encouraging the use of deeper and more instructive metrics. A new research contribution is much more useful if the community understands why the contribution is faster, rather than simply knowing it outperforms the predecessor.

Acknowledgements

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung.

6. REFERENCES

[1] V. Agarwal, F. Petrini, et al. Scalable graph exploration on multicore processors. *SC*, 2010.

[2] S. Beamer, K. Asanović, and D. A. Patterson. The GAP benchmark suite. arXiv:1508.03619, 2015.

[3] S. Beamer, K. Asanović, and D. A. Patterson. Direction-optimizing breadth-first search. *SC*, 2012.

[4] S. Beamer, K. Asanović, and D. A. Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. *International Symposium on Workload Characterization (IISWC)*, 2015.

[5] A. Buluç and J. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(4):496–509, 2011.

[6] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A roofline model of energy. In *IPDPS*, 2013.

[7] T. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1 – 1:25, 2011.

[8] 9th DIMACS implementation challenge. `http://www.dis.uniroma1.it/challenge9/`, 2006.

[9] J. Emer and D. Clark. A characterization of processor performance in the VAX-11/780. *ISCA*, 1984.

[10] GAP benchmark suite reference code v0.6. `https://github.com/sbeamer/gapbs`.

[11] Graph500 benchmark. `www.graph500.org`.

[12] O. Green, M. Dukhan, and R. Vuduc. Branch-avoiding graph algorithms. *SPAA*, 2015.

[13] Intel performance counter monitor. `www.intel.com/software/pcm`.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. *Symposium on Operating Systems Design and Implementation*, 2012.

[15] J. Leskovec, D. Chakrabarti, et al. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[16] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 4th edition, 2008.

[17] K. D. Underwood, M. Vance, J. Berry, and B. Hendrickson. Analyzing the scalability of graph algorithms on Eldorado. In *IPDPS*, 2007.