Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server

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

***Abstract*—Graph processing is an increasingly important application domain and is typically communication-bound. In this work, we analyze the performance characteristics of three high-performance graph algorithm codebases using hardware performance counters on a conventional dual-socket server. Unlike many other communication-bound workloads, graph algorithms struggle to fully utilize the platform's memory bandwidth and so increasing memory bandwidth utilization could be just as effective as decreasing communication. Based on our observations of simultaneous low compute and bandwidth utilization, we find there is substantial room for a different processor architecture to improve performance without requiring a new memory system.**

I. Introduction

Graph processing is experiencing a surge of interest, as applications in social networks and their analysis have grown in importance [25], [31], [45]. Additionally, graph algorithms have found new applications in recognition [28], [46] and the sciences [39].

Graph algorithms are notoriously difficult to execute efficiently, and so there has been considerable recent effort in improving the performance of processing large graphs for these important applications. Their inefficiency is due to the large volume and irregular pattern of communication between computations at each vertex or edge. When executed on a shared-memory multiprocessor, this large volume of communication is translated into loads and stores in the memory hierarchy. When executed in parallel on a large-scale distributed cluster, this communication is translated into messages across the inter-processor network. Because message-passing is far less efficient than accessing memory in contemporary systems, distributed clusters are a poor match to graph processing. For example, on Graph500, a world ranking of the fastest supercomputers for graph algorithms, the efficiency of each core in a cluster is on average one to two orders-of-magnitude lower than cores in shared-memory nodes [21]. This communication-bound behavior has led to surprising results, where a single Mac Mini operating on a large graph stored in an SSD is able to outperform a medium-sized cluster [26].

Due to the inefficiency of message-passing communication, the only reason to use a cluster for graph processing is if the data is too large to fit on a single node [30]. However, many interesting graph problems are not large enough to justify a cluster. For example, the entire Facebook friend graph requires only 1.5 TB uncompressed [3], which can reside in a single high-end server node's memory today.

In this paper, we focus on the performance of a shared-memory multiprocessor node executing optimized graph algorithms. We analyze the performance of three high-performance graph processing codebases each using a different parallel runtime, and we measure results for these graph libraries using five different graph kernels and a variety of input graphs. We use microbenchmarks and hardware performance counters to analyze the bottlenecks these optimized codes experience when executed on a modern Intel Ivy Bridge server. We derive the following insights from our analysis:

- *Memory bandwidth is not fully utilized* - Surprisingly, the other bottlenecks described below prevent the off-chip memory system from achieving full utilization on well-tuned parallel graph codes. In other words, there is the potential for significant performance improvement on graph codes with current off-chip memory systems.
- *Graph codes exhibit substantial locality* - Optimized graph codes experience a moderately high last-level cache (LLC) hit rate.
- *Reorder buffer size limits achievable memory throughput* - The relatively high LLC hit rate implies many instructions are executed for each LLC miss. These instructions fill the reorder buffer in the core, preventing future loads that will miss in the LLC from issuing early, resulting in unused memory bandwidth.
- *Multithreading has limited potential for graph processing* - In the context of a large superscalar out-of-order multicore, we see only modest room for performance improvement on graph codes from multithreading and that is likely achievable with only a modest number (2) of threads per core.

We also confirm conventional wisdom that the most efficient algorithms are often the hardest to parallelize, and that these algorithms have their scaling hampered by load imbalance, synchronization overheads, and non-uniform memory access (NUMA) penalties. Additionally, we find that different input graph sizes and topologies can lead to very different conclusions for algorithms and architectures, so it is important to consider a range of input graphs in any analysis.

Based on our empirical results, we make recommendations for future work in both hardware and software to improve graph algorithm performance.

II. Graph Background

Graph applications are characterized not only by the algorithms used, but also by the structure of the graphs that make up their workload. A graph's *diameter* is the largest number of vertices that must be traversed in order to travel from one vertex to another when paths that backtrack, detour, or loop are excluded from consideration. The *degree* of a node in a graph is the number of connections it has to other nodes, and the *degree distribution* is the probability distribution of these degrees over the whole graph.

Commonly used graphs can be divided into two broad categories named for their most emblematic members: *meshes* and *social networks* [7]. Meshes tend to be derived from physically spatial sources, such as road maps or the finite-element mesh of a simulated car body, so they can be relatively readily partitioned along the few original spatial dimensions. Due to their physical origin, they usually have a high diameter and a degree distribution that is both bounded and low.

Conversely, social networks come from non-spatial sources, and consequently are difficult to partition using any reasonable number of dimensions. Additionally, social networks have a low diameter ("small-world") and a power-law degree distribution ("scale-free"). In a *small-world graph*, most nodes are not neighbors of one another, but most nodes can be reached from every other by a small number of hops [42]. A *scale-free graph* has a degree distribution that follows a power law, at least asymptotically [5]. The fraction of nodes in a scale-free graph having k connections to other nodes is P(k) $\sim k^{-\gamma}$, where γ is a parameter typically in the range $2 < \gamma < 3$.

Meshes are perhaps the most common mental model for graphs, since they are typically used in textbook figures. Unfortunately, they do not capture the challenges posed by the small-world and scale-free properties of social network topologies. The small-world property makes them difficult to partition (few cut edges relative to enclosed edges), while the scale-free property makes it difficult to load balance a parallel execution since there can be many orders of magnitude difference between the work for different vertices. Although the highest degree vertices are rare, their incident edges constitute a large fraction of the graph.

III. Methodology

To provide a representative graph workload, we chose five popular graph kernels and exercised them with five different input graphs using three high-performance graph codebases running on a modern high-end server. Unless otherwise stated, we measure the full workload of all combinations of codebases, kernels, and graphs (75 data points) for each system configuration.

A. Graph Kernels and Input Graphs

We selected five graph kernels based on their popularity in applications as well as in other research papers:

1) **Breadth-First Search (BFS)** is actually only a traversal order, but it is so fundamental to graph algorithms that we include it in our suite. We convert BFS into a kernel by tracking the parent vertex in the traversal.
2) **Single-Source Shortest Paths (SSSP)** computes the distance to all reachable vertices from a start vertex.
3) **PageRank (PR)** is way of determining influence within a graph, and was initially used to sort search results [38].
4) **Connected Components (CC)** attaches the same label to all vertices in the same connected component.
5) **Betweenness Centrality (BC)** is commonly used in social network analysis to measure the influence a vertex has on a graph. A vertex's betweenness-centrality score is related to the fraction of shortest paths between all vertices that pass through the vertex. To keep runtimes tractable, our BC benchmark starts from only a few root vertices instead of every vertex.

We selected the input graphs used in our evaluation to be topologically diverse and Table I lists them. *Twitter*, *road*, and *web* are all from real-world data, while *kron* and *uniform* are synthetic. Twitter, web, and kron all have the "social network" topology, as they have low effective diameters and a power-law degree distribution. Road is an example of a "mesh" topology, with its high diameter, low average degree, and low maximum degree. Even though our graph suite includes some of the largest publicly available real-world graphs, they do not fully use the memory capacity of our system. As is done in the Graph500 benchmark, we generate arbitrarily large synthetic graphs to fill our memory capacity. Our parameters for kron are chosen to match those of Graph500 [21]. Uniform is low diameter, like a social network, but its degree distribution is normal rather than a power law. Hence, in our uniform graph each vertex tends to be accessed roughly the same number of times, unlike social networks where a few vertices are accessed disproportionately often. Uniform represents the most adversarial graph, as by design it has no locality, however, it is also the most unrealistic and serves to act as lower bound on performance.

B. Graph Processing Frameworks

For this study, we use three of the fastest available graph codebases, which each use a different parallel runtime.

Galois [36] uses its own custom parallel runtime specifically designed to handle irregular fine-grained task parallelism. Algorithms implemented in Galois are free to use autonomous scheduling (no synchronization barriers), which should reduce the synchronization otherwise needed for high-diameter graphs. Additionally, Galois' scheduler takes into consideration the plaform's core and socket topology.

Ligra [40] uses the Cilk [8] parallel runtime and is built on the abstractions of edge maps and vertex maps. When applying these map functions, Ligra uses heuristics to determine in which direction to apply them (push or pull) and what data structures to use (sparse or dense). These optimizations make Ligra especially well suited for low-diameter graphs.

GAP Benchmark Suite (GAPBS) [6], [19] is a collection of high-performance implementations written directly in OpenMP with C++11. GAPBS is not a framework, so it does not force common abstractions onto all implementations, but instead frees each to do whatever is appropriate for a given algorithm.

Short Name	Description	# Vertices (M)	# Edges (M)	Degree	Degree Distribution	Approximate Diameter	References
road	Roads of USA	23.9	58.3	2.4	bounded	6,304	[15]
twitter	Twitter Follow Links	61.6	1,468.4	23.8	power	14	[25]
web	Web Crawl of .sk Domain	50.6	1,949.4	38.5	power	135	[14]
kron	Kronecker Synthetic Graph	128.0	2,048.0	16.0	power	6	[21], [27]
uniform	Uniform Random Graph	128.0	2,048.0	16.0	normal	7	[18]

TABLE I. GRAPHS USED FOR EVALUATION

All three codebases are competitive, and depending on the input graph or kernel, a different codebase is the fastest. For descriptions of the implementations and their parallelization strategies, we refer the reader to the original publications.

C. Hardware Platform

To perform our measurements, we use a current 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). The server has 128 GB of DDR3-1600 DRAM provided by 16 DIMMS. To access hardware performance counters, we use Intel PCM [24] and PAPI [32]. We compile all code with gcc-4.8, except Ligra that uses Cilk Plus gcc-4.8. To ensure consistency across runs, we disable Turbo Boost (dynamic voltage and frequency scaling).

When reporting memory traffic from the performance counters, we focus on memory requests caused by LLC misses as these are the most problematic for performance. We do not include prefetch traffic measurements because they obscure the results, but benefits of successful prefetching appear indirectly as fewer cache misses. During our study, we observed IVB intelligently prefetching aggressively when the memory bandwidth utilization would otherwise be low, but ceasing prefetching when the application is using a large fraction of the memory bandwidth (the hardware prefetcher does not prevent full memory bandwidth utilization).

IV. MEMORY BANDWIDTH POTENTIAL

Any LLC miss will cause even a large out-of-order processor to stall for a significant number of cycles. Ideally, while waiting for the first cache miss to resolve, at least some useful work could be done, including initiating loads early that will cause future cache misses. Unfortunately, a load must satisfy the following four requirements before it can be issued:

1) *Processor fetches load instruction* - Control flow reaches the load instruction (possibly speculatively).
2) *Space in instruction window* - The Reorder Buffer (ROB) is not full and has room for the load.
3) *Register operands are available* - The load address can be generated.
4) *Memory bandwidth is available* - At the core level there is a miss-status holding register (MSHR) available and there is not excessive contention within the on-chip interconnect or at the memory controller.

If any of the above requirements is not met, the load will be unable to issue. In particular, memory bandwidth cannot be a bottleneck unless the first three requirements are satisfied, thus the other factors can prevent memory bandwidth from being fully utilized.

We use a parallel pointer-chase as a synthetic microbenchmark to measure the achievable memory bandwidth on IVB under various conditions. A parallel pointer-chase exposes the needed parameters but is otherwise quite simple [1], [35]. With a single pointer-chase, there is no memory-level parallelism (MLP) and the memory latency is exposed since requests must be completed serially. To generate more MLP, we simply add more parallel pointer chases to the same thread.

To force loads to access the memory system, we set pointers to point randomly within an array sized large enough such that LLC hit rates are less than 1.5% (typically $\geq$ 2 GB). We report bandwidths in terms of memory references per second as measured by performance counters. We also report achieved bandwidths in terms of *effective MLP*, which is the average number of memory requests in flight according to Little's Law (memory bandwidth $\times$ memory latency). It is worth distinguishing this from *application MLP*, which is how much memory-request parallelism is allowed by the application's data dependencies, which will be always greater than or equal to the achieved effective MLP.

Our simple microbenchmark is designed to trivially satisfy the first two requirements above, allowing us to focus on and measure the last two. Branch mispredictions should be rare since the loop repeats many times, so fetching the load instructions should not be hindered. The microbenchmark is a tight loop, so there should be a relatively high density of loads thus reducing the impact of instruction window size (168 for Ivy Bridge). By changing the number of parallel pointer-chases, we can artificially control the maximum application MLP possible, which allows us to moderate the operand availability requirement. We can then observe what bandwidths are possible and even what the bandwidth limits are.

Figure 1 shows the microbenchmark results. The local memory latency is 86 ns (MLP=1). Local bandwidth for a single thread appears to saturate when MLP$\geq$10, implying the core supports 10 outstanding misses, and this is confirmed by published sources on the Ivy Bridge microarchitecture. Using a second thread on the same core does not change the maximum bandwidth regardless of how the outstanding memory requests are spread across the two threads.

To see the impacts of Non-Uniform Memory Access (NUMA) on our dual-socket system, instead of allocating the memory being used by our microbenchmark on the same socket (local), we allocate on the other socket (remote) or interleaved across both sockets (interleaved). NUMA may introduce bandwidth restrictions, but for a single core in isolation, the primary consequence is twice the latency ($\approx$184 ns). When accessing remote memory, the maximum bandwidth is halved due to the same number of outstanding data requests experiencing twice the latency.

After exploring how application MLP changes bandwidth

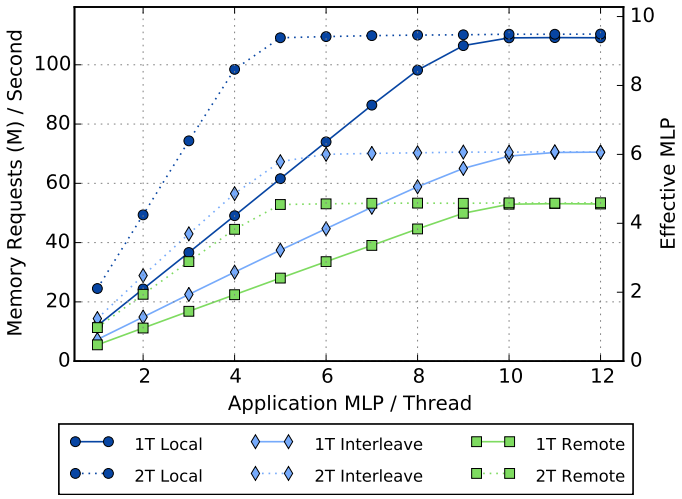

Fig. 1. Memory bandwidth achieved by parallel pointer chase microbenchmark (random) in units of memory requests per second (left axis) or equivalent effective MLP (right axis) versus the number of parallel chases (application MLP). Single core using 1 or 2 threads and differing memory allocation locations (local, remote, and interleave).

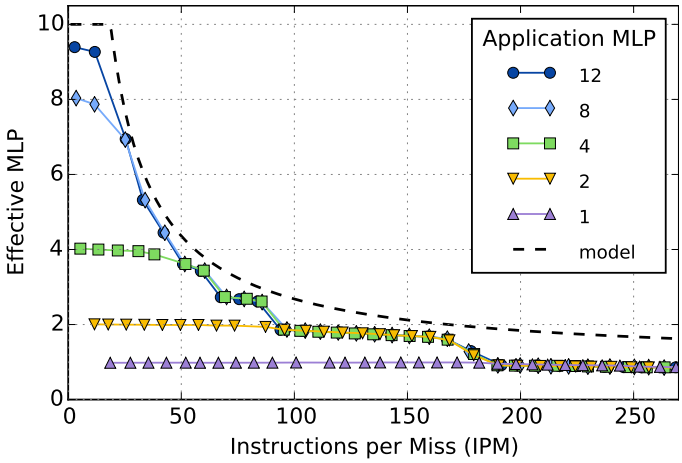

Fig. 2. Memory bandwidth achieved by parallel pointer chase microbenchmark with varying number of nops inserted (varies IPM). Using a single thread with differing numbers of parallel chases (application MLP).

(requirement 3) and how many outstanding misses the hardware supports (requirement 4), we now return to the impact of the instruction window size (requirement 2). Using inline assembly, we add `nops` to our pointer-chase loop, thus moving the loads farther apart in the instruction stream. To examine the net result, we use the metric instructions per miss (IPM), which is the inverse of the common misses per kilo-instruction metric (MPKI = 1000/IPM).

As shown in Figure 2, window size is an important constraint on our platform, as bandwidth is inversely related to IPM, which confirms our intuition that memory requests must fit in the window in order to be issued. Assuming the loads are evenly spaced, we obtain a simple model for an upper-bound (with w as the window size):

$$MLP_{model} = \min(MLP_{max}, w/IPM + 1)$$

For our IVB core, $MLP_{max} = 10$ and $w = 168$. The

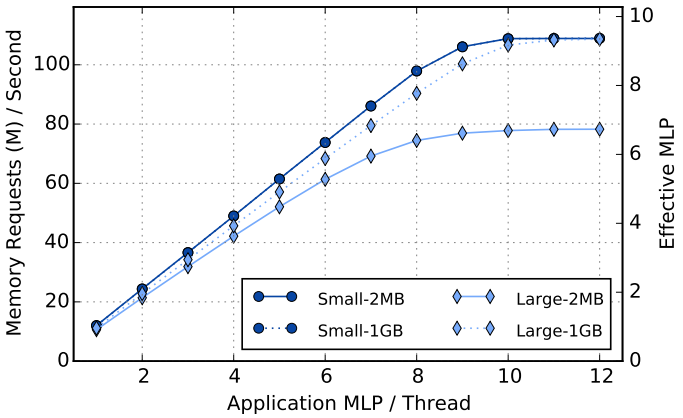

Fig. 3. Impact of 2 MB and 1 GB page sizes on memory bandwidth achieved by single-thread parallel pointer chase for array sizes of *small* (1 GB) and *large* (16 GB).

curved region adds one because if the window can hold n IPM-sized intervals, it can hold $n + 1$ endpoints. Our model is pessimistic as it assumes cache misses are evenly spaced. If there is substantial variation in the miss interval (jitter), it is possible to exceed the model bound, but we find this simple model instructive for the rest of the study as we observe bandwidth is inversely related to IPM.

Memory bandwidth can also be constrained by frequent TLB misses. The four requirements above are necessary for a load to issue, but once issued, missing in the TLB incurs a latency penalty for its refill, which in turn will decrease bandwidth for the same number of outstanding memory requests. IVB's Linux distribution supports Transparent Huge Pages (THP), which eagerly combines consecutive 4 KB pages into 2 MB pages when possible. IVB also supports 1 GB pages, but these must be set aside by Linux in advance and require substantial application code modifications. Larger pages not only reduce the chance of a TLB miss, but they also reduce the time per refill by needing fewer hops to walk the page table and by reducing the size of the page-table working set (better cache locality).

Figure 3 varies the page size (2 MB or 1 GB) and the array size (1 GB or 16 GB) for our pointer-chase synthetic microbenchmark. With 2 MB pages from THP, most loads for both array sizes will result in a cache miss and a TLB miss (IVB has only 32 2 MB TLB entries), but the maximum bandwidth obtained with the larger array is substantially reduced due to increases in TLB refill time (confirmed by performance counters). Using 1 GB pages restores the bandwidth even though there will still be frequent TLB misses (only 4 1 GB TLB entries) because the refill time is reduced enough to not be problematic. With 1 GB pages, the page table will only need 16 entries and will likely remain in the L1 cache. Our random microbenchmark exemplifies the worst case for the TLB, so any form of locality will reduce the performance penalties from TLB misses.

We further parallelize our microbenchmark and run it across all of the cores, and the maximum bandwidths we achieve are visible in Figure 10. The data in this section shows the maximum achievable memory bandwidth for a core, socket, or entire system given the amount of application MLP, IPM,

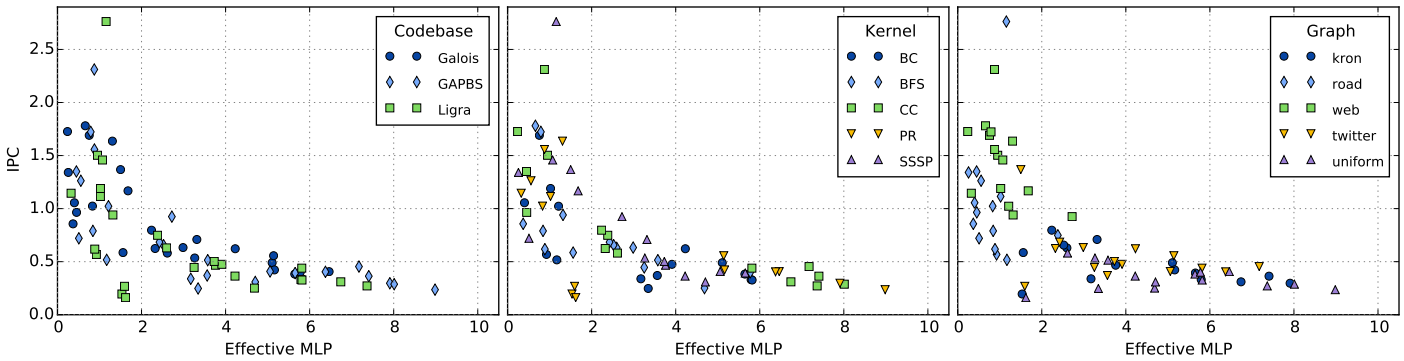

Fig. 4. Single-thread performance in terms of instructions per cycle (IPC) of full workload colored by: codebase (left), kernel (middle), and input graph (right).

memory location (NUMA), and page size. In the following section, we measure the memory bandwidths achieved by the high-performance graph codebases.

V. Single-Core Analysis

In this section, we begin to characterize our workload using only a single thread on a single core in order to remove any parallel execution effects (multithreading, poor parallel scaling, load imbalance, NUMA penalties). Despite being amongst the highest-performance implementations, all three codebases often execute instructions at a surprisingly low IPC (Figure 4), and this disappointing performance observed is not specific to any graph algorithm or codebase. The input graph does have a large impact as we will discuss later in this section.

Figure 4 shows that there is an unsurprising tradeoff between computation and communication, as no executions sustain a high IPC and a high memory bandwidth. A processor can only execute instructions at a high rate if it rarely waits on memory, and hence consumes little memory bandwidth. Conversely, for a processor to use a great deal of memory bandwidth, it must have many memory requests outstanding, causing it to be commonly waiting on memory and will thus execute instructions slowly. Although some executions do use an appreciable amount of compute (upper left of Figure 4) or use an appreciable fraction of the memory bandwidth (lower right), most do not. Many executions are actually in the worst lower-left quadrant, where they use little memory bandwidth, but their compute throughput is also low, presumably due to memory latency.

In general across our codebases, kernels, and inputs graphs, a single core struggles to use all of the raw bandwidth available (10 outstanding misses). With the same communication volume, utilizing more bandwidth should lead to higher performance. Using the four requirements from Section IV, we investigate what is limiting the core's bandwidth utilization for what should be a memory-bound graph processing workload.

To have many loads outstanding, the processor must first fetch those load instructions, and this typically requires correctly predicting the control flow. Although frequent branch mispredictions will be harmful to performance in theory, if the processor is already waiting on memory (achieving moderate memory bandwidth utilization), performance is insensitive to the branch misprediction rate (Figure 5), implying many of these branches are miss independent. When the processor is

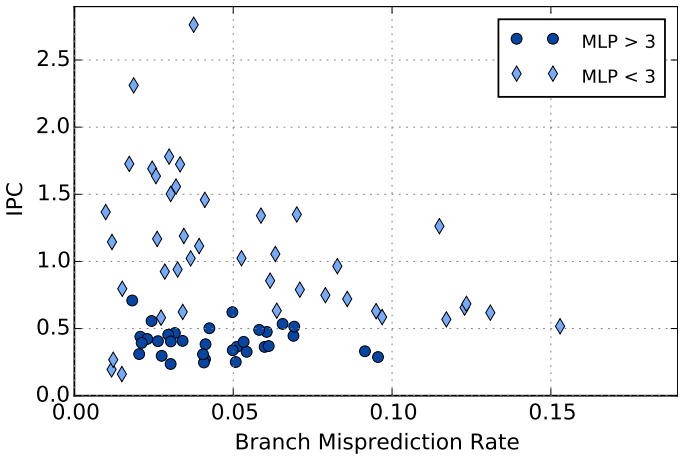

Fig. 5. Single-thread performance of full workload relative to branch misprediction rate colored by memory bandwidth utilization.

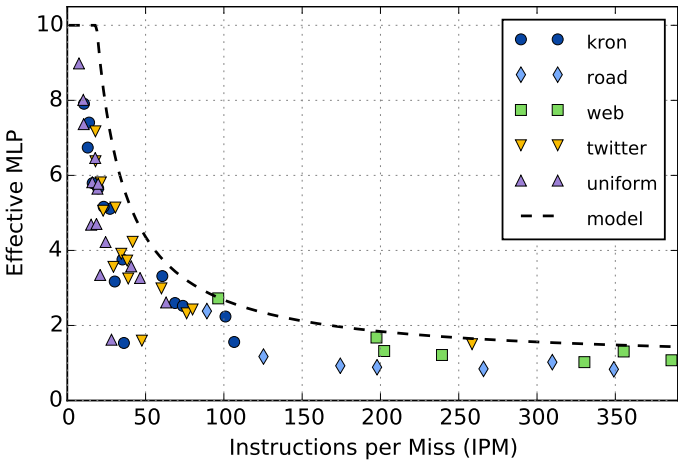

Fig. 6. Single-thread achieved memory bandwidth of full workload relative to instructions per miss (IPM). *Note: Some points from road & web not visible due to IPM>1000 but model continues to serve as an upper bound.*)

not memory-bound, frequent branch mispredictions will hurt performance, but a low misprediction rate is no guarantee for good performance, implying there are remaining unaccounted bottlenecks.

Once the processor fetches the future outstanding loads, those loads need to be able to fit into the instruction window,

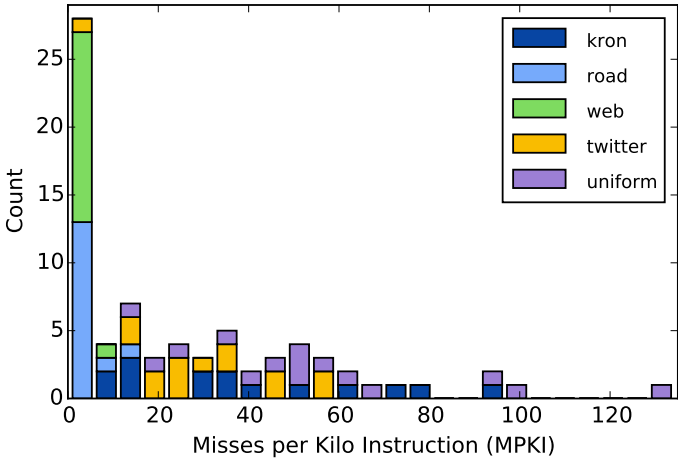

Fig. 7. Single-thread MPKI (in terms of LLC misses) of full workload.

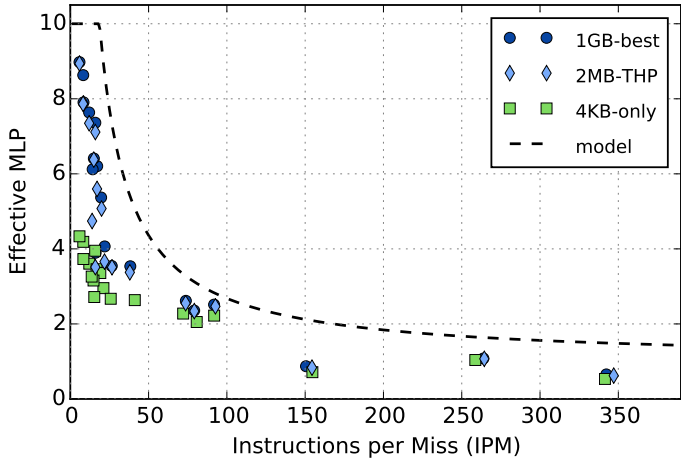

Fig. 8. Single-thread achieved memory bandwidth of GAPBS for all kernels and graphs varying the operating system page size. *2 MB-THP* uses Transparent Hugepages (THP) and lets the operating system choose to promote 4 KB to 2 MB pages (happens frequently). *1 GB-best* is the fastest execution using manually allocated 1 GB pages for the output array, the graph, or both.

and the model from Section IV serves as an upper bound for our workload (Figure 6). Although the model is technically a pessimistic upper bound since it assumes outstanding loads are evenly spaced apart, in practice this seems to be a suitable approximation. In spite of the core being capable of handling 10 outstanding misses, an IPM of greater than 18.7 will not allow all these loads to fit in the window according to our model. Most of the executions have an IPM greater than this cutoff, and thus have their effective bandwidth limited by the instruction window size. The caches achieve a modest hit rate (Figure 7), which raises the IPM by absorbing much of the memory traffic.

As mentioned above, the properties of the graph can have a substantial impact on the cache performance, which in turn will affect not only the amount of memory traffic, but also how fast it can be transferred. For example, in Figure 6 the graph road has a high IPM because it is much smaller than the other graphs. The topology can also have an impact, as the graphs kron and uniform are about the same size and diameter, and yet uniform typically uses more bandwidth because it has a lower IPM caused by more cache misses. The graph kron experiences fewer cache misses because it is scale-free, as a few high degree vertices will be accessed frequently (great temporal locality). Finally, the graph web has a higher degree, which allows for longer contiguous reads (better spatial locality) causing more cache hits and thus a higher IPM.

Although there is not typically substantial benefit from using 1 GB pages, using 4 KB pages does have quite a performance penalty. Fortunately, THP is on by default and requires no application modifications. We vary the operating system page size for the GAPBS codebase in Figure 8. Relative to the baseline using THP (2 MB pages), using 1 GB pages improves performance by more than 10% in only 4/25 cases but disabling THP, which forces all pages to be 4 KB, decreases performance by at least 10% in 19/25 cases. To use 1 GB pages, we modify GAPBS to allocate 1 GB pages for the graph, the output array, or both (typically the best) and pick whichever one is fastest. The general insensitivity to the 1 GB page size for our graph workload is another indication of locality.

We compare data dependencies versus branch mispredictions to explain performance slowdown, and while difficult to disentangle, the evidence points much more strongly to the former than to the latter. With a combination of knowledge of IVB's architecture and confirmation from performance counters, we eliminate other possible performance limiters. Due to sophisticated hashing of memory addresses, there is not significant bank contention in the LLC or at the memory controllers. The load buffer can hold 64 entries, so it rarely limits outstanding loads before the ROB (168 entries) or the MSHRs (10 per core). Mis-speculated loads are already counted by the performance counters we utilize. The graph workloads we measure have clearly dominant application phases (no substantial temporal variation).

None of executions of actual graph processing workloads are able to achieve a memory bandwidth corresponding to the 10 outstanding misses our synthetic microbenchmarks demonstrate the cores are capable of sustaining, and most are not even close. For a single thread, the biggest bandwidth limiter is fitting loads into the instruction window, which prevents off-chip memory bandwidth from becoming a bottleneck.

VI. Parallel Performance

With an understanding of the limits and capabilities of a single thread, we move on to the whole system. Running the codebases at full capacity delivers speedups for all executions, and with 32 threads on 16 cores we achieve a speedup greater than 8× (relative to single-thread) in 49 of 75 cases and a median speedup of 9.3× (Figure 9). Unfortunately, some of the executions (typically road and web) increase their bandwidth consumption by more than they improve runtime, implying their parallel executions have more memory traffic than their single-threaded counterparts.

The compute and throughput utilization for the parallel executions (Figure 10) is strikingly similar to utilizations for a single core (Figure 4). Although web and sometimes road appear to break the trend by simultaneously using more compute throughput and memory bandwidth, they do move extra data. The similarities between parallel utilization and serial utilization suggest that the bottlenecks of the core persist

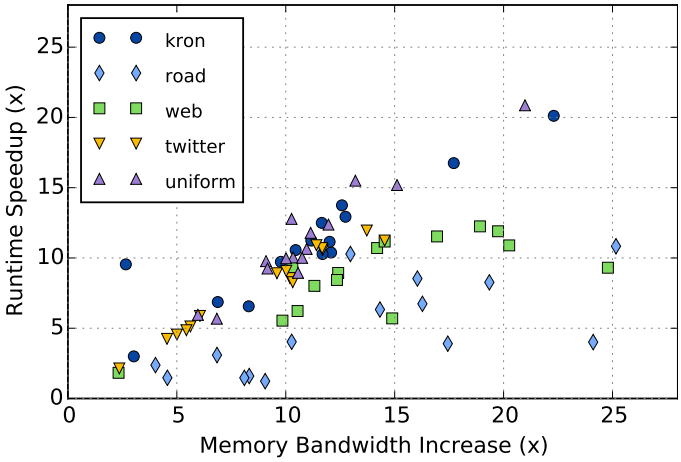

Fig. 9. Improvements in runtime and memory bandwidth utilization of full workload for full system (32 threads on 16 cores) relative to 1 thread. Points along unit slope transfer the same amount of data, so points below the unit slope (often road and web) transfer extra data.

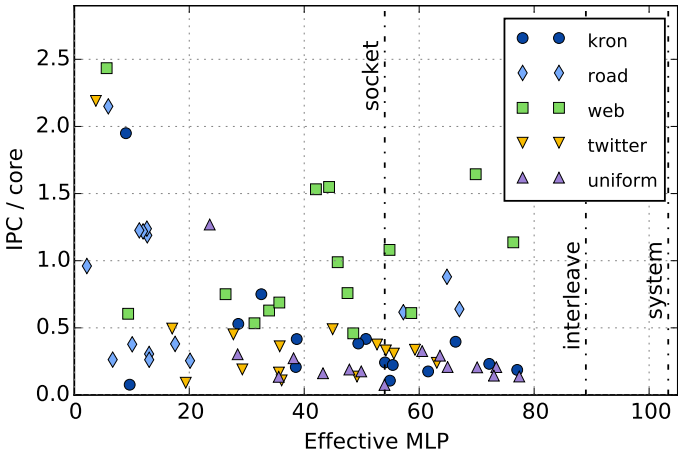

Fig. 10. Full system (32 threads on 16 cores) performance of full workload. Vertical lines correspond to maximum achieved bandwidths from Section IV for a single socket (socket), both sockets with memory interleaved (interleave), and both sockets with locally allocated memory (system).

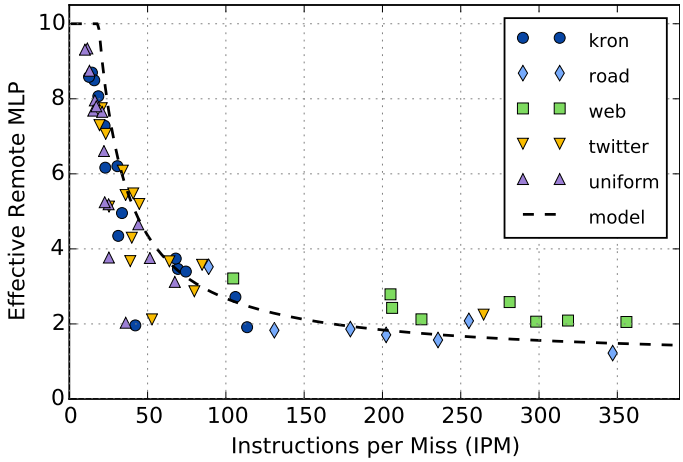

Fig. 11. Single-thread achieved memory bandwidth of full workload executing out of remote memory. Calculating effective MLP with remote memory latency (instead of local memory latency) returns a result similar to local memory results (Figure 6).

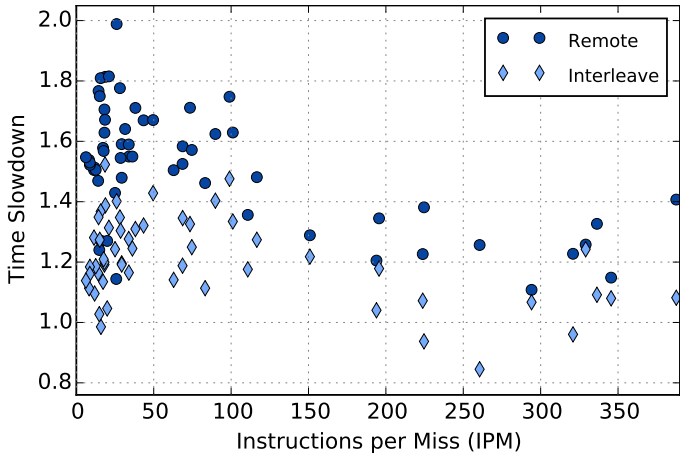

Fig. 12. Single-socket (8 cores) slowdown relative to local memory of full workload executing out of remote memory or interleaved memory.

and hurt utilization at the system scale. Due to the generally linear relation between performance and memory bandwidth, fully utilizing the off-chip memory system could improve performance by 1.3–47× (median 2.4×).

There may be graph algorithm implementations with better parallel scaling properties, but these advanced algorithms are used because they deliver better absolute performance. Parallel scaling can be hampered by software issues (poor scalability, load imbalance, synchronization overheads, and redundant communication), but in the remainder of this work we will consider hardware imposed complications for parallelization: NUMA and multithreading.

VII. NUMA Penalty

With multi-socket systems, non-uniform memory access (NUMA) penalties are a common challenge. From the results of Section IV, it would appear that NUMA should halve performance, but our results indicate the penalty for NUMA may be substantially less severe in practice.

For a single thread using only remote memory, performance is halved as it transfers the same amount of data with the same number of outstanding memory requests but at twice the latency for effectively half the bandwidth. Calculating the effective MLP with the remote memory latency instead of the local memory latency shows the workload still obeys the simple bandwidth model (Figure 11).

With more cores, this NUMA penalty is reduced (Figure 12), and for executions that use less memory bandwidth (higher IPM), the NUMA penalty is reduced further. A core using only remote memory is clearly an adversarial worst case. For a full system workload without locality, half of the traffic should still go to local memory. Consequently, the *interleaved* pattern in Figure 12 is more realistic and it has one third the performance loss of *remote* (median 1.16× slowdown vs. 1.48× slowdown).

We confirm that NUMA has a moderate performance penalty. Unfortunately, many graphs of interest are low diameter and hard to partition effectively [20] so it is challenging to avoid inter-socket communication. Therefore, efforts to

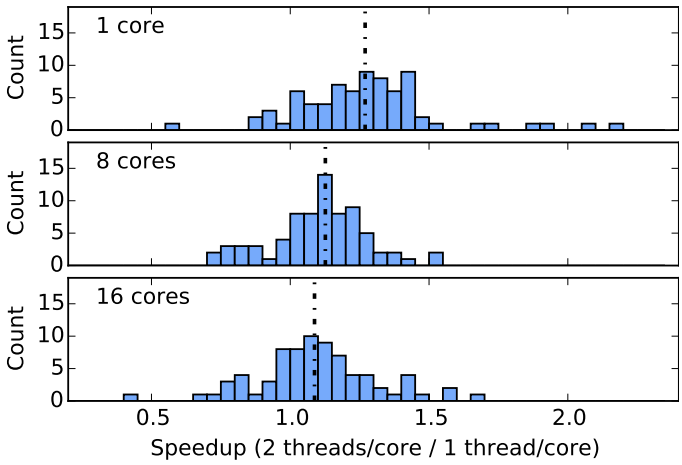

Fig. 13. Distribution of speedups of using two threads per core relative to one thread per core of full workload for one core, one socket (8 cores), and whole system (2 sockets)). Dotted line is median.

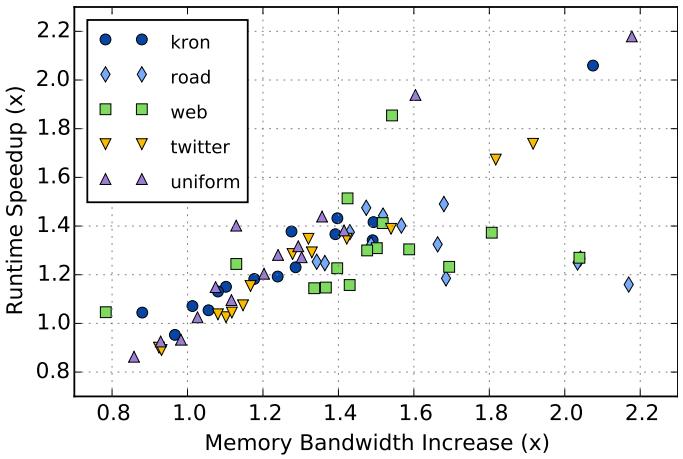

Fig. 14. Improvements in runtime and memory bandwidth utilization of full workload for one core using two threads relative to one thread.

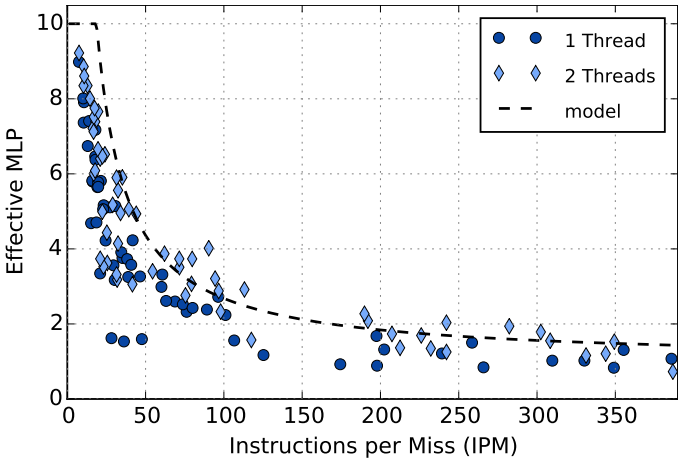

Fig. 15. Achieved memory bandwidth of full workload relative to instructions per miss (IPM) with one or two threads on one core.

move computation (rather than data) have fared the best when optimizing graph processing for NUMA [1], [11].

VIII. Limited Room for SMT

Multithreading, and in this work's context of a superscalar out-of-order processor, simultaneous multithreading (SMT) [17], aims to increase utilization. The additional software-exposed parallelism threads provide can be used to mitigate unresolved data dependences by increasing application MLP as well as reducing the demand placed on branch prediction since each thread will have fewer instructions in flight. Using IVB, we measure the performance gains of using a second thread per core, which evaluates how well SMT reduces the performance loss from unresolved data dependencies and branch predictions without incurring new overheads.

Across all scales (single core, single socket, or single system), the second thread is usually beneficial, but only to a modest degree (Figure 13) as most speedups are less than $1.5\times$. Even so, these modest speedups from SMT are not inconsequential, as SMT economically improves system performance.

Multithreading also has the potential to introduce new performance challenges. More threads increase parallelism, which in turn can worsen the damage caused by load imbalances and synchronization overheads. Worse yet, more threads can end up competing for capacity in the cache resulting in increased memory traffic. Analogous to the results for multicore (Section VI), the road and web graphs in Figure 14 are examples of this competition as the improvement in bandwidth is greater than the improvement in runtime.

For a single thread, we find the biggest performance limiter to be fitting loads into the instruction window, and SMT is no different as the addition of a second thread to the same core still mostly obeys our simple model (Figure 15). If the workload of the two threads is heterogenous it is possible for an SMT core to exceed our simple model. One thread could generate most of the cache misses sustaining a high effective MLP while the other thread (unencumbered by cache misses) could execute instructions quickly to increase IPM. In practice, the variation between threads is modest and thus most points are not far above our model.

Multithreading can improve performance, but in the context of this study (graph processing workload on a superscalar out-of-order multi-socket system), it has limited potential. The modest improvements two-way multithreading provides in this study cast doubts on how much more performance is to be gained by additional threads.

IX. Related Work

Our study touches on many aspects of computer architecture, so we focus this section specifically on prior work relevant to graph algorithms. Compared to prior work on the architectural requirements for graph algorithms, our study has a much larger and more diverse graph workload. We study 5 kernels from 3 codebases with 5 input graphs (some which are real and not synthetic).

A survey [29] of both hardware and software concerns for parallel graph processing lists 'poor locality' as one of its chief concerns. Although it is cognizant of the greater cost of heavily multithreaded systems, it argues they are better for

graph algorithms due to their memory latency tolerance and support for fine-grained dynamic threading. Bader et al. [4] also endorse heavily threaded systems because of concerns of memory accesses being mostly non-contiguous (low locality).

Cong et al. [12] compare a Sun Niagara 2 to a IBM Power 7 when executing graph algorithms to understand architectural implications. They find memory latency (not memory bandwidth) to be a bottleneck for both platforms, and neither platform has enough threads to fully hide it.

To better understand graph algorithm architectural requirements, prior work has explicitly examined the locality behavior of graph algorithms. Cong et al. [13] study several Minimum Spanning Tree algorithms with a reuse distance metric (temporal locality). They find graph algorithms do have less (but not no) locality, but observe some algorithms with less locality sometimes perform better, and hypothesize this is due to not accounting for spatial locality. Analytical models for BFS can accurately predict the reuse distance of BFS on certain random graphs [47]. Murphy et al. [33] examine serial traces from a variety of benchmark suites including graph algorithms. Despite locality metrics based on an extremely small cache for the time of publication, they observe that integer applications tend to have less locality than floating-point applications, but are still far better than random.

Efforts to improve performance by explicit NUMA optimizations typically require complicated manual modifications and are not generally applicable to all graph algorithms. Agarwal et al. [1] improve BFS performance using custom inter-socket queues. With a high-end quad-socket server, they are able to outperform a Cray XMT. Satish et al. [11] minimize inter-socket communication for BFS, and provide a detailed performance model for their implementation.

Although hardware prefetchers may struggle to predict non-streaming memory accesses, explicit software prefetching has been investigated as a means to improve graph algorithm performance [1], [12], [23]. Not unlike explicit NUMA optimizations, for graph algorithms, using software prefetching requires human intervention. Software prefetching can be difficult to implement effectively for all graph algorithms because it is often hard to generate the addresses desired sufficiently before they are needed.

Green et al. investigate improving graph algorithm performance by reducing branch mispredictions using conditional moves [22]. They conclude that branch mispredictions are responsible for a 30%–50% performance loss, but in our results (Section V) we do not observe such a large penalty when considering the limitations imposed by data dependences and fitting loads into the instruction window.

Runahead execution is a technique to improve processor performance in the presence of cache misses [16], and in the case of an out-of-order core, runahead execution attempts to economically obtain the benefits of a larger instruction window [34].

The Cray XMT, and its predecessor the MTA-2 [2], are systems explicitly designed to handle irregular problems including graph algorithms [41]. Designed for workloads without locality, they feature many hardware threads and no data caches.

There has been substantial effort characterizing graph processing workloads on GPUs. Since GPUs are optimized for regular data parallelism, Burtscher et al. propose metrics to quantify control-flow irregularity and memory-access irregularity and they perform performance counter measurements on real hardware [9]. For some graph algorithms, they observe the performance characteristics depend substantially on the inputs. A continuation of that research uses a software simulator to change GPU architectural parameters and observes performance is more sensitive to L2 cache parameters than to DRAM parameters, which suggests there is exploitable locality [37]. Xu et al. also use a simulator and identify synchronization with the CPU (kernel invocations and data transfers) as well as GPU memory latency to be the biggest performance bottlenecks [44]. Che et al. profile the Pannotia suite of graph algorithms and observe substantial diversity across algorithms and inputs [10]. Wu et al. investigate the most important primitives needed for higher-level programming models for graph algorithms [43]. Contrasting these GPU works from our work, in addition to the difference in hardware platform (CPU versus GPU), we use much larger input graphs enabled by executing on real hardware (no downsizing to reduce simulation time) and by using server-sized memory (not constrained by GPU memory capacity).

X. Conclusion

Our diverse workload (varied implementations, algorithms, and input graphs) demonstrates there is no single representative benchmark and we find the input graph to have the largest impact on the performance characteristics.

Most of our workload fails to fully utilize IVB's off-chip memory bandwidth due to having an insufficient number of outstanding memory requests. The biggest bandwidth bottleneck is the instruction window, because it cannot hold a sufficient number of instructions to incorporate the needed number of rare cache-missing instructions. A high LLC hit rate makes these cache misses rare, and we find this challenges the misconception that graph algorithms have little locality. TLB misses are only measurably detrimental when at least a moderate amount of memory bandwidth is utilized, and we find transparent huge pages to be effective at ameliorating much of the performance loss due to TLB misses. Branch mispredictions and unresolved data dependences can also hinder memory bandwidth utilization, but they are secondary to the interaction between the cache hit rate and the instruction window size. Bandwidth is also moderately hindered by NUMA effects, so software techniques to increase intra-socket locality or hardware techniques to decrease inter-socket latency will be beneficial.

The parallel scaling of our workload indicates that performance typically scales linearly with memory bandwidth consumption. Since our workload fails to fully utilize IVB's memory bandwidth, an improved processor architecture could use the same memory system but improve performance by utilizing more memory bandwidth. For our workload on IVB, SMT is typically beneficial, and when it improves performance, it does so by using more memory bandwidth. Unfortunately, in the context of an out-of-order core, SMT helps only modestly, and additional techniques will be needed to utilize the rest of the unused memory bandwidth.

Overall, we see no perfect solution to the performance challenges presented by graph algorithms. Many techniques can improve performance, but all of them will have quickly diminishing returns, so greatly improving performance will require a multifaceted approach.

ACKNOWLEDGEMENTS

We thank the reviewers and David Ediger for their helpful feedback. 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. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] V. Agarwal *et al.*, "Scalable graph exploration on multicore processors," *Supercomputing (SC)*, 2010.

[2] W. Anderson *et al.*, "Early experience with scientific programs on the Cray MTA–2," in *Supercomputing (SC)*, 2003.

[3] L. Backstrom *et al.*, "Four degrees of separation," *ACM Web Science Conference*, pp. 45–54, 2012.

[4] D. A. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," *International Conference on Parallel Processing*, Jul 2005.

[5] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, pp. 509–512, Oct 1999.

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

[7] S. Beamer, K. Asanović, and D. A. Patterson, "Direction-optimizing breadth-first search," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[8] R. D. Blumofe *et al.*, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing (JPDC)*, vol. 37, no. 1, pp. 55–69, 1996.

[9] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," *IISWC*, pp. 141–151, 2012.

[10] S. Che *et al.*, "Pannotia: Understanding irregular GPGPU graph applications," in *IISWC*, 2013.

[11] J. Chhugani *et al.*, "Fast and efficient graph traversal algorithm for CPUs: Maximizing single-node efficiency," *IPDPS*, 2012.

[12] G. Cong and K. Makarychev, "Optimizing large-scale graph analysis on multithreaded, multicore platforms," *IPDPS*, Feb 2011.

[13] G. Cong and S. Sbaraglia, "A study on the locality behavior of minimum spanning tree algorithms," in *High Performance Computing (HiPC)*. Springer, 2006, pp. 583–594.

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

[15] "9th DIMACS implementation challenge - shortest paths." `http://www.dis.uniroma1.it/challenge9/`, 2006.

[16] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *International Conference on Supercomputing (ICS)*, 1997.

[17] S. J. Eggers *et al.*, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[18] P. Erdős and A. Réyni, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

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

[20] J. E. Gonzalez *et al.*, "Powergraph: Distributed graph-parallel computation on natural graphs," *Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 17–30, 2012.

[21] "Graph500 benchmark." `www.graph500.org`.

[22] O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[23] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," *Parallel Architectures and Compilation Techniques (PACT)*, 2011.

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

[25] H. Kwak *et al.*, "What is Twitter, a social network or a news media?" *International World Wide Web Conference (WWW)*, 2010.

[26] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," *Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–17, Oct 2012.

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

[28] Y. Low *et al.*, "GraphLab: A new framework for parallel machine learning," *Uncertainty in Artificial Intelligence*, 2010.

[29] A. Lumsdaine *et al.*, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[30] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[31] A. Mislove *et al.*, "Measurement and analysis of online social networks," *ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[32] P. J. Mucci *et al.*, "PAPI: A portable interface to hardware performance counters," in *Department of Defense HPCMP Users Group Conference*, 1999.

[33] R. C. Murphy and P. M. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *IEEE Transactions on Computers*, vol. 56, no. 7, pp. 937–945, 2007.

[34] O. Mutlu *et al.*, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[35] J. Nelson *et al.*, "Crunching large graphs with commodity processors," *USENIX conference on Hot topic in parallelism (HotPAR)*, 2011.

[36] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," *Symposium on Operating Systems Principles (SOSP)*, 2013.

[37] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," *IISWC*, 2014.

[38] L. Page *et al.*, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[39] J. B. Pereira-Leal, A. J. Enright, and C. A. Ouzounis, "Detection of functional modules from protein interaction networks," *PROTEINS: Structure, Function, and Bioinformatics*, vol. 54, no. 1, pp. 49–57, 2004.

[40] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[41] K. D. Underwood *et al.*, "Analyzing the scalability of graph algorithms on Eldorado," in *IPDPS*, 2007, pp. 1–8.

[42] D. Watts and S. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, pp. 440–442, June 1998.

[43] Y. Wu *et al.*, "Performance characterization for high-level programming models for GPU graph analytics," in *IISWC*, 2015.

[44] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on GPUs: Where are the bottlenecks?" *IISWC*, 2014.

[45] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[46] K. You *et al.*, "Scalable HMM-based inference engine in large vocabulary continuous speech recognition," *IEEE Signal Processing Magazine*, 2010.

[47] L. Yuan *et al.*, "Modeling the locality in graph traversals," in *International Conference on Parallel Processing (ICPP)*, 2012.