Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges

Conference Paper · September 2016

DOI: 10.1109/MASCOTS.2016.66

5 authors, including:

Assaf Eisenman
Stanford University
9 PUBLICATIONS **236** CITATIONS
SEE PROFILE

Ludmila Cherkasova
Arm Research
220 PUBLICATIONS **7,306** CITATIONS
SEE PROFILE

Parallel Graph Processing on Modern Multi-Core Servers: New Findings and Remaining Challenges

Assaf Eisenman[1,2], Ludmila Cherkasova[2], Guilherme Magalhaes[3], Qiong Cai[2], Sachin Katti[1]
[1]Stanford University, [2]Hewlett Packard Labs, [3]Hewlett Packard Enterprise
assafe@stanford.edu, {lucy.cherkasova, guilherme.magalhaes, qiong.cai}@hpe.com, skatti@stanford.edu

***Abstract*—Big Data analytics and new problems in social networks, computational biology, and web connectivity led to a renewed research interest in graph processing. Due to "irregularity" of graph computations, efficient parallel graph processing faces a set of software and hardware challenges debated in literature. In this paper[1], by utilizing hardware performance counters, we characterize system bottlenecks, resource usage, and the efficiency of popular graph applications on the modern commodity hardware. We analyze selected graph applications (implemented in the Galois framework) on a variety of graph datasets: both scale-free graphs and meshes. Our profiling shows that with an increased number of cores the analyzed graph applications achieve a good speedup, which is highly correlated with utilized memory bandwidth. Contrary to traditional past stereotypes, we find that graph applications significantly benefit from hardware prefetchers. Moreover, the use of transparent huge pages (THP) exhibits a "double win" impact: 1) THP significantly decrease the TLB misses and page walk durations, and 2) THP boost the hardware prefetchers' performance. These insights shed light to understand the performance of emerging systems with large memories. Our profiling framework reports hardware counter values over time. It reveals the danger of using averages for a bottleneck and resource usage analysis: many applications have a time-varying behavior and stretch the usage of system resources to their peak. We discuss the new insights and remaining challenges for guiding the design of future hardware and software components for efficient graph processing.**

I. Introduction

Graphs have been used to capture relationships and interactions between different data entities in social networks, web systems, computational biology, etc. Graph algorithms are an integral part of solutions in scientific computing, data mining, and machine learning. The rise of commodity multi-core processors offered new challenges and opportunities for efficient use of this platform for parallel graph processing. Unfortunately, most graph algorithms exhibit characteristics that make them difficult to parallelize and execute efficiently. The survey paper [23] lists a set of software and hardware challenges in parallel graph processing that limit achievable application performance. The main cause of these challenges is the *"irregularity" of graph computations*, which stands on a way to efficient parallelization of graph processing by either partitioning the algorithm computation or the graph data. Uneven data partitioning may cause unbalanced computations, which affect and *limit the achievable application scalability*. *A poor data locality* is another major issue described in [23] and referred as an obstacle to efficient graph processing in many papers. It is often explained by pointer-chasing style of graph computations: they are data-driven, with dependencies between the tasks defined at runtime. This pattern results in independent random memory accesses, and makes graph applications memory-intensive and memory-bound. The next generation of hardware, software, and applications will embrace new technologies, such as non-volatile memory (NVM). This will have far-reaching potential to radically change hardware architecture and systems software. To better understand the design points of future hardware and software components, we have to analyze a set of workloads that can drive the system design and implementation.

In this paper, we aim to *quantify* how parallel graph applications utilize underlying resources in the latest Intel multi-core systems. We are especially interested in the performance analysis of system bottlenecks caused by parallel graph processing and interworking of processor and memory. For our study we have chosen the *Galois system* [28], which is specially tailored to parallelize "irregular algorithms" [20]. Since graph algorithms exhibit similar properties, the Galois framework was successfully applied for parallel graph processing. It was shown to outperform many other graph processing frameworks. In our profiling approach, we leverage performance events from the processor Performance Monitoring Units, both inside and outside the cores.

In our short paper [15], we open up a discussion that some traditional stereotypes portrayed in earlier literature do not hold. In particular, our analysis reveals that available *memory bandwidth is not fully utilized* (therefore, it is not a bottleneck), and that graph processing exhibits a *good data locality* [15] that could be and should be efficiently exploited. In this paper, we continue this discussion and provide in-depth analysis of popular graph algorithms executed on a variety of large graph datasets, including *scale-free graphs* and *meshes*. While application performance is sensitive to the type and properties of processed graphs, there are many common observations in how the resources of the underlying modern systems are used. New key findings are the following:

- *Hardware prefetchers are very useful*: they provide significant performance benefits across all considered applications (ranging between 15%-78% in the application speedup). We observe that the achieved graph application speedup is highly correlated with utilized memory bandwidth and increased cache hit rates.
- The use of *transparent huge pages (THP)* significantly improves graph application performance. The detailed analysis shows 30%-63% of the TLB misses reduction, and 50%-70% reduced page walk durations across all selected graph applications.
- *THP significantly boost performance of hardware prefetchers* for graph applications. This reveals that there is a significant amount of a potential spatial and temporal data locality present in graph data, which is underutilized during graph processing due to hard-

[1]This work was originated and largely completed during A. Eisenman' internship at Hewlett Packard Labs in summer 2015.

ware prefetchers being "limited" by the default 4 KB page boundaries.

- It is *dangerous to use averages for bottleneck analysis and resource usage estimates*: many graph applications have a time-varying behavior during processing. It can stretch system resources usage to their peak and change observed bottlenecks over time.

In our study, we derive that graph applications are memory latency bound. Therefore, among the remaining challenges for improving graph processing performance, we see a need to strive for memory latency reduction and latency hiding techniques. This is especially important for future NVM systems with significantly higher memory latencies.

This paper is organized as follows. Section II outlines selected graph algorithms, introduces our profiling approach and testbed details. Section III characterizes achievable memory performance of the system under test. Section IV presents our findings. Section V outlines related work. Finally, Section VI presents conclusion and future work directions.

II. Galois System and Our Profiling Approach

In this section, we motivate why we have chosen the Galois framework, outline the selected graph algorithms, describe details of our experimental testbed, and introduce our profiling approach based on hardware performance counters.

A. Galois: Parallel Graph Processing Framework

Over the past few years, a number of new graph processing frameworks were offered to simplify the parallel implementation of graph algorithms, e.g., GraphChi[19], GraphLab [21], Apache Giraph [2], and Ligra [31], just to name a few. We chose the *Galois* framework [28], proposed for an automated parallelization of irregular algorithms. Since graph processing highly resembles the irregular algorithms, Galois can be efficiently applied for large graph processing and diverse graph analytics. Galois is a task based parallelization framework, with a graph computation expressed in either *vertex* or *edge* based style. It implements the coordinated and autonomous execution of these tasks and allows the application-specific control of *scheduling policies* (application-specific task priorities). The *runtime library* performs optimizations to avoid synchronization and communication between threads. In the recent study [30], conducted by independent researchers, graph algorithms implemented in Galois have been shown as highly competitive compared to a manually crafted, optimized code (only 1.1-2.5 times performance difference for a diverse set of popular graph algorithms executed on a variety of large datasets).

B. Selected Graph Applications

For our profiling study, we selected five popular graph algorithms implemented in Galois [28] with different characteristics. Below is a short summary of these algorithms:

- **PageRank (PR):** this algorithm is used by the search engines to rank websites for displaying the output results. PageRank offers a way of measuring the importance and popularity of website pages. This algorithm computes the page rank of every vertex in a directed graph iteratively.
- **Breadth First Search (BFS):** this is a typical graph traversal algorithm performed on an undirected, unweighted graph. A goal is to compute a distance from a given source vertex s to each vertex in the graph, i.e., finding all the vertices which are "one hop" away, "two hops" away, etc. The BFS algorithm is typically implemented iteratively. The Galois BFS implementation [28] blends coordinated and autonomous scheduling with a switch in scheduling defined by the level of graph traversal and a type of the graph, i.e., small- or high-diameter graphs.
- **Betweenness Centrality (BC):** this algorithm measures the importance of a node in a graph. In the social networks analysis, it is used for computing the user "influence" index. The vertex index reflects the fraction of shortest paths between all vertices that pass through a given vertex. The Galois implementation [28] exercises a priority scheduling with a priority function based on the BFS level of a vertex.
- **Connected Components (CC):** this algorithm identifies the maximal sets of vertices reachable from each other in an undirected graph. The Galois framework provides a topology-driven implementation [28], where each edge is visited once for adding it to a concurrent union-find data structure.
- **Approximate Diameter (DIA):** the graph diameter is defined as a maximum length of the shortest paths between any pair of vertices in the graph. The precise (exact) computation of a graph diameter can be prohibitively expensive for large graphs, and this is why many implementations provide a diameter approximation. The Galois implementation [28] is based on BFS computed from an arbitrary vertex. Then it computes another BFS from a vertex with a maximum distance in the first BFS. The process continues until the maximum distance does not increase.

We chose these graph applications in Galois for a few reasons: *i)* these problems represent popular graph kernels (they can be utilized as modules for solving more complex graph problems), and *ii)* the Galois implementation of these kernels was optimized and tuned by the Galois team to produce an efficient code as shown in [28], [30].

Using an optimized and tuned code in our study is very important for understanding the *real system bottlenecks* during large graph processing (rather than discovering the bottlenecks related to an inefficiently written code).

C. Experimental Hardware Platform

In our profiling experiments, we use a dual-socket system with one of the latest Intel Xeon-based processor:

- **Intel Xeon E5-2660 v2 with Ivy Bridge processor**: each socket supports 10 *two-way hyper-threaded* cores running at 2.2 GHz and 25 MB of LLC. The system has 128 GB DDR3-1866 DRAM (i.e., with 4 DDR3 channels) and the peak memory bandwidth of 59.7 GB/s reported by the system specification.

There are a few nuances in using hardware performance counters for accurately measuring the system hardware memory access latencies and characterizing memory performance. Performance counter measurements are provided in cycles, e.g., stall cycles. We need to translate durations measured in cycles to durations in nanoseconds, especially for profiling the application characteristics over time. Typically, the specified processor frequency can be used for this translation.

However, for energy efficiency many processors are applying DVFS (Dynamic Voltage and Frequency Scaling) during workload processing depending on the system utilization. Therefore, to preserve a fixed relationship between cycles and time we *disable Turbo Boost and the DVFS feature*.

We *disable hyper-threading* in our experiments in order to analyze the application performance as a function of an increased number of physical cores assigned for processing. We are especially interested in understanding how these added compute resources translate in the application speedup, and how it changes utilized memory bandwidth in the system.

In this work, we are concentrating on the bottleneck analysis of Galois applications executed on a *single multi-core socket*. In such a way, we can see the best possible multi-threaded code execution with its performance not being impacted by the coherency traffic and NUMA considerations.

D. Profiling Methodology

In our profiling approach, we leverage the Performance Monitoring Units (**PMUs**) implemented in the Ivy Bridge processor to observe micro-architectural events, which expose the inner working of the processor. For our analysis we select the performance events shown in Table I, using PMUs located inside and outside the cores. The 3d column provides the exact Intel event names, while the 2nd column shows the mnemonic (short) event names, we use in the paper[2].

Counters 1-2 in Table I refer to events found in the integrated Memory Controller (**MC**). These events are read per each memory channel. $DRAM_{reqs}$ is used to count the number of outgoing MC requests issued to DRAM per channel, while MC_{cycles} is used for measuring the run time.

Counter 3 measures the number of occupied *Line Fill Buffers* (**LFB**) in each cycle, from which we deduce the *average LFB occupancy*. LFBs accommodate outstanding memory references (which missed L1 data cache) until the corresponding data is retrieved from the memory hierarchy (caches or memory). Therefore, LFBs may limit the number of cache misses handled by the core. Counters 4-5 are used to compute achievable *Instructions per Cycle (***IPC***)*.

Because stores typically do not delay other instructions directly, counters 6-17 concentrate on loads. Counters 6-11 are used for the analysis of hit/miss rates in data caches L1, L2, and LLC. Counters 12-14 refer to *Data Translation Lookaside Buffer (***DTLB***)* accesses and to the page walk duration in the case of the DTLB miss.

Counters 15-18 provide measure the execution stall cycles incurred by the system, typically caused by waiting for the memory system to return data (including caches).

Counters 19-20 refer to different types of stalls due to resource allocation: *Reservation Station* (**RS**) and *Reorder Buffer* (**ROB**) being full. We will provide more details in Section IV-F.

In order to read performance counters, we use `PAPI` [6] framework. We instrumented PAPI into the algorithms source code in such a way that it initializes the counters when the algorithm starts the computation part (after the setup period). Due to a limited number of programmable PMU events per run, we execute these experiments multiple times for collecting and profiling different event sets.

[2]We explicitly define hw counters and performance metrics used in our study to enable other researchers to reproduce our results and perform their own studies.

1	$DRAM_{reqs}[i]$	ivbep_unc_imc0::UNC_M_CAS_COUNT:ALL
2	MC_{cycles}	ivbep_unc_imc0::UNC_M_DCLOCKTICKS
3	$FB_{occupancy}$	L1D_PEND_MISS:PENDING
4	$Instructions$	ix86arch::INSTRUCTION_RETIRED
5	$Cycles$	ix86arch::UNHALTED_CORE_CYCLES
6	$L1_{loads}$	perf::L1-DCACHE-LOADS
7	$L1_{misses}$	perf::L1-DCACHE-LOAD_MISSES
8	$L2_{loads}$	L2_RQSTS:ALL_DEMAND_DATA_RD
9	$L2_{hits}$	L2_RQSTS:DEMAND_DATA_RD_HIT
10	LLC_{loads}	perf::LLC-LOADS
11	LLC_{misses}	perf::LLC-LOAD-MISSES
12	$DTLB_{loads}$	perf::DTLB-LOADS:precise=1
13	$DTLB_{misses}$	DTLB_LOAD_MISSES:MISS_CAUSES_A_WALK
14	$DTLB_{miss\_duration}$	DTLB_LOAD_MISSES:WALK_DURATION
15	$Stalls_{mem}$	CYCLE_ACTIVITY:STALLS_LDM_PENDING
16	$Stalls_{L1}$	CYCLE_ACTIVITY:STALLS_L1D_PENDING
17	$Stalls_{L2}$	CYCLE_ACTIVITY:STALLS_L2_PENDING
18	$Stalls_{total}$	CYCLE_ACTIVITY:CYCLES_NO_EXECUTE
19	$Stalls_{RS}$	RESOURCE_STALLS:RS
20	$Stalls_{ROB}$	RESOURCE_STALLS:ROB

TABLE I. SELECTED PERFORMANCE EVENTS IN IVY BRIDGE PROCESSOR FAMILY AND MEMORY SUBSYSTEM.

We use $DRAM_{reqs}[i]$ to count the number of MC requests issued to DRAM over memory channel i (out of 4 channels). Modern Intel processors have a memory line size of 64 bytes, thus we multiply the sum of $DRAM_{reqs}$ by 64 to get the byte traffic between the MC and DRAM. We then calculate memory bandwidth with the following formula:

$$Memory\_BW(bytes/s) = \frac{mem\_line\_size * \sum_{i=0}^{3} DRAM_{reqs}[i]}{Time}$$

For the *memory-bound* and *DRAM_bound* application characterization, we use a methodology first introduced in [32] and later described in the Intel Optimization Manual [4]. We define the memory-bound metric as the cycles where the execution is stalled and there is at least one outstanding memory demand load in the memory hierarchy:

$$Memory\_bound = \frac{Stalls_{mem}}{Cycles}$$

Because Ivy Bridge does not have a counter for the number of execution stalls that happen due to LLC pending loads, we use the following formula for defining $DRAM\_bound$ metric. This formula approximates the number of cycles where the execution is stalled and there is at least one outstanding memory reference in DRAM:

$$DRAM\_bound = \frac{Stalls_{L2} * LLC\_miss\_fraction}{Cycles}$$

We utilize LLC_{misses} to estimate $LLC\_miss\_fraction$, and apply a correction factor *WEIGHT* to reflect the latency ratios between DRAM and LLC. For the Ivy Bridge processor, the empirical factor value is 7 as defined in [4]:

$$LLC\_miss\_fraction = \frac{WEIGHT * LLC_{misses}}{LLC_{hits} + WEIGHT * LLC_{misses}}$$

The PAPI tool reports average values of counters collected during the application execution time interval: $[app\_begin, app\_end]$. However, average values could be misleading for a bottleneck analysis: they do not expose the execution "imbalance", especially along multiple threads, and limit useful insights into the program resource usage. We have added an additional sampling feature for measuring values of performance counters over time, based on the system wide monitoring program from the libpfm4 library

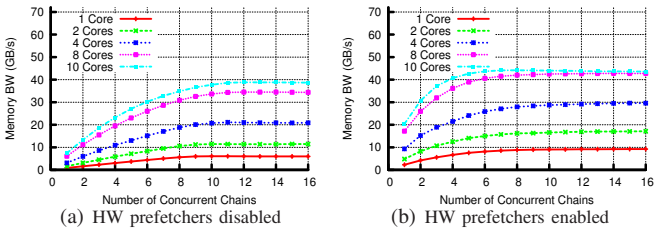
[5]. We set it to run at the background and sample the selected set of performance counters every X ms, e.g., 10 or 100 ms.

III. System Performance Characterization

The goal of our study is to understand system bottlenecks and memory system resource utilization caused by parallel graph processing. In order to accomplish this goal, we need to set realistic expectations on the achievable performance (peak resource usage) of the system under study. In many traditional cases, the peak memory bandwidth is measured using a traditional STREAM benchmark [8]. These measurements characterize memory bandwidth achievable under the *sequential access pattern*. However, the real graph applications executed on current multi-core systems exhibit a very different access pattern, where concurrent threads executed on different cores utilize the memory system by issuing a set of *concurrent* and *independent* memory references.

To achieve this goal, we utilize the open source *pChase benchmark* [7], [25]. pChase is a memory-latency bound pointer-chasing benchmark. It creates a pointer chain, such that the content of each element dictates which memory location is read next. This ensures that the next memory reference cannot be issued until the result of the previous one is returned. The benchmark can create *multiple* independent chains per thread, with memory references from different chains issued *concurrently*. To avoid prefetching and caching side effects it is important that **hardware prefetchers are disabled** during the pChase experiments in the system.

Fig. 1 (a) shows the achievable memory bandwidth (Y-axis) measured with hardware performance counters as described in Section II-D. Five different lines reflect measurement results of the pChase benchmark executed with a different number of threads (1, 2, 4, 8, and 10), which are processed by available cores in a socket, i.e., 1, 2, 4, 8, and 10 cores. Each thread is executed with an increased number of concurrent chains (X-axis). The results show that a single pChase thread achieves the paeak memory bandwidth of 5.5 GB/s with approximately 9-10 concurrent pointer chains. This result makes sense because for each load miss in L1 cache a Line Fill Buffer (LFB) should be allocated. Modern Intel processors (Sandy Bridge, Ivy Bridge, Haswell) have 10 LFBs per core, and therefore, **a single core is limited to issuing 10 concurrent memory references** [4].



(a) HW prefetchers disabled (b) HW prefetchers enabled

Fig. 1. Measured memory bandwidth with the pChase benchmark executed with multiple cores, where each thread performs an increasing number of concurrent pointer chains.

Figure 1 (a) shows that with two pChase threads and two cores the achievable memory bandwidth increases perfectly to 11 GB/s, i.e., 2 times higher. However, for four pChase threads and four cores the peak memory bandwidth is 20 GB/s, and for 10 cores, it is 39 GB/s. This shows that when four cores are issuing memory references at their "maximum speed" an additional system *bottleneck starts to form in the memory subsystem*, most likely in the memory controller. This bottleneck could be related to memory requests queueing on existing 4 memory channels in DDR3.

Now, we will demonstrate the importance of understanding and evaluating the **performance impact of hardware prefetchers** that cause increased memory bandwidth usage as a result. Figure 1 (b) shows the achievable memory bandwidth (Y-axis) measured by the pChase benchmark executed with a different number of threads (1, 2, 4, 8, and 10), and a different number of concurrent chains per thread (X-axis). It shows almost double memory bandwidth for pChase executed with a single thread (on 1-core configuration) compared to Figure 1 (a) with hardware prefetch disabled. For a higher number of pChase threads (cores) the amount of an additional prefetch memory traffic decreases. Overall, hardware prefetching increases the achievable socket memory bandwidth (when all 10 cores execute the pChase threads) by 12.8% and it reaches 44 GB/s.

Finally, Table II compares the memory bandwidth measured with the STREAM benchmark based on a sequential access pattern and the pointer-chasing pChase benchmark stressing concurrent independent memory accesses. These results show a significant performance difference in the measured memory bandwidth under STREAM and pChase for 1-core experiments: both with hardware prefetchers enabled and disabled. The memory bandwidth is up to 50% higher under STREAM, utilizing a sequential access pattern. The peak memory bandwidth (measured with 10 cores) is very close under STREAM and pChase: in both cases, the memory bandwidth is limited by the memory system bottleneck.

Benchmark	HW Prefetchers Disabled		HW Prefetchers Enabled	
	1 core	10 cores	1 core	10 cores
STREAM	7.5 GB/s	40 GB/s	13.5 GB/s	45 GB/s
pChase	5.5 GB/s	38.5 GB/s	10 GB/s	44.5 GB/s

TABLE II. Memory bandwidth measured with STREAM and pChase in a single socket multi-core configuration.

Setting realistic expectations on memory performance is important: the official system specification of 59.7 GB/s is 30% higher than our peak memory bandwidth measurements.

IV. Performance Study

In this section, we analyze the resource usage of selected graph applications executed on three different graph datasets.

A. Graph Datasets: Scale-Free Graphs and Meshes

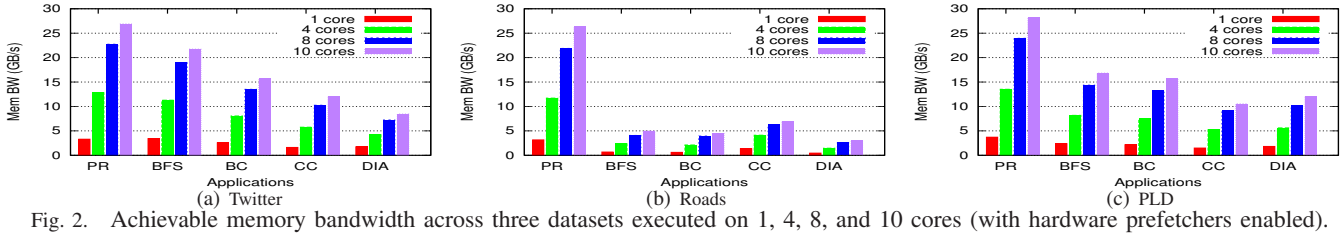
Performance of graph algorithms and applications may significantly depend on the structure and properties of the graphs used for processing. For our profiling study, we use the following three datasets described in Table III.

Dataset	*Brief Description*	*# Vertices*	*# Edges*	*Reference*
Twitter	Twitter Follower Graph	61.5 M	1,458 M	[18]
Roads	Roads of USA	23.9 M	58.3 M	[1]
PLD	Web Hyperlink Graph	39 M	623 M	[26]

TABLE III. Graph datasets used in the evaluation study.

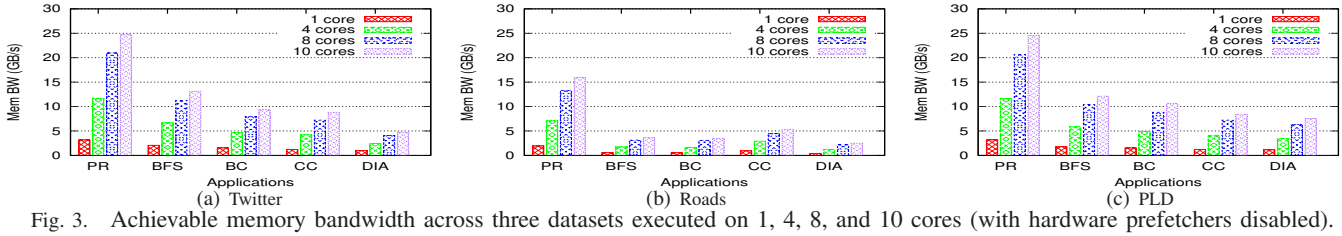
The selected datasets represent two broad classes of graphs [14] with different properties: relational graphs (social networks) and spatial graphs (meshes):

- *Social networks.* The two datasets *Twitter* and *PLD* belong to a category of *social networks*. Social networks are difficult to partition because they come



(a) Twitter (b) Roads (c) PLD

Fig. 2. Achievable memory bandwidth across three datasets executed on 1, 4, 8, and 10 cores (with hardware prefetchers enabled).



(a) Twitter (b) Roads (c) PLD

Fig. 3. Achievable memory bandwidth across three datasets executed on 1, 4, 8, and 10 cores (with hardware prefetchers disabled).

from the non-spatial sources. They are often called "scale-free networks" or "small-world" graphs due to a low diameter. In the "small-world" graphs, most nodes can be reached from each other by a limited number of hops. Another property of scale-free networks is that their degree distribution follows a power-law, at least asymptotically. Thus, there is a group of vertices with a very high number of connections (edges), while the overall majority of vertices are connected to fewer neighbors.

- *Meshes.* The third dataset *Roads* belong to a category of *meshes* or *planar graphs*. These graphs are often defined by physical, spatial sources (e.g., "Roads of USA" or maps), and therefore, they can be much easier partitioned along the original spatial dimensions. Because of their physical origin, these graphs tend to have a high diameter, while a degree distribution in such graphs is typically low.

B. Benefits of Hardware Prefetchers

One of the traditional views and stereotypes [13], [9] related to graph processing and its random memory access pattern is that hardware prefetchers are not useful. Driven by this opinion, multiple research efforts aim to design explicit software prefetching for improving graph processing performance. However, implementing sophisticated software prefetching policies for a variety of graph algorithms might be a challenging and difficult task since this approach requires the detailed profiling and human insights.

Table IV shows the *application speedup* for processing with hardware prefetchers enabled compared to results with hardware prefetchers disabled (we show the speedup ranges across different configurations with 1, 4, 8, and 10 cores).

Dataset	*PR*	*BFS*	*BC*	*CC*	*DIA*
Twitter	21-22%	36-41%	35-41%	20-26%	61-78%
Roads	58-62%	29-32%	17-27%	27-48%	18-20%
PLD	17-20%	30-37%	34-41%	17-21%	58-70%

TABLE IV. APPLICATION SPEEDUP WHEN PROCESSED WITH HW PREFETCHERS ENABLED COMPARED TO PROCESSING WITH HW PREFETCHERS DISABLED FOR DIFFERENT CONFIGURATIONS: WITH 1, 4, 8, AND 10 CORES.

While the efficiency of hardware prefetchers depends on both the graph dataset properties and the nature of performed algorithms, there are significant performance improvements across all the algorithms and datasets. The application speedup is ranging from 15% to 78% as shown in Table IV.

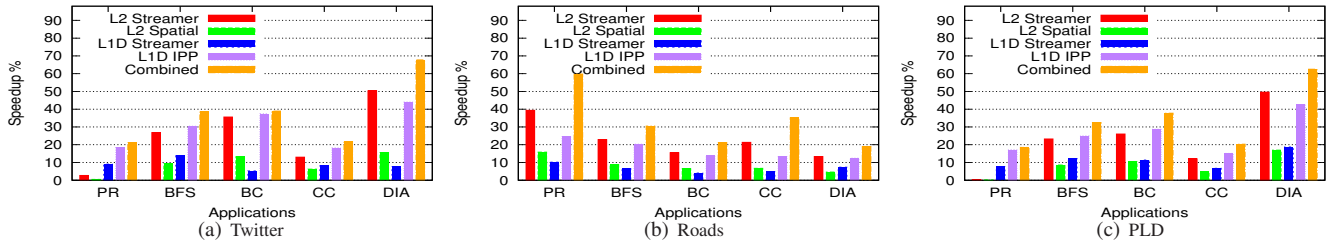
Now, we analyze the memory bandwidth used by applications under study. Figures 2 (a)-(c) show the memory bandwidth utilization for each profiled application with hardware prefetchers enabled. Each figure shows four bars for each profiled application. These bars represent the average memory bandwidth measured during the application processing in configurations with 1, 4, 8, and 10 cores on three datasets: *Twitter*, *Roads*, and *PLD* respectively.

Figures 3 (a)-(c) show memory bandwidth utilization with hardware prefetchers disabled for each profiled application.

We observe that the application speedup is highly correlated with an increased memory bandwidth utilization when processed with hardware prefetchers enabled. For example, for the PageRank algorithm executed on the *Roads* dataset, we can see that memory bandwidth with prefetchers enabled (shown in Figure 2 (b)) is 63% higher compared to memory bandwidth with prefetchers disabled (shown in Figure 3 (b)). Moreover, the *L2 and LLC hit rates increase* from 13% and 8.2% (prefetchers disabled) to 64% and 18.6% (prefetchers enabled) respectively This corresponds to 58%-62% of the application speedup presented in Table IV. Indeed, when hardware prefetchers are able to identify the current access pattern and bring a significant amount of additional useful data from memory to caches – this leads to a correlated application performance speedup.

It is worth noting that the memory bandwidth utilization is much lower when the selected applications are executed on mesh-based dataset *Roads* compared to scale-free graphs *Twitter* and *PLD*. Partially, it is due to the fact that *Roads* is a smaller dataset. The other reason for using less memory bandwidth on *mesh*-based datasets can be attributed to the nature of the traversal algorithms: the low number of edges associated with each vertex limits the concurrency at the next step during graph processing. For *Roads* dataset, the vertex-based algorithm *PageRank* achieves highest memory bandwidth of 27 GB/s, while for the other four applications the average memory bandwidth is below 7 GB/s.

We can see that memory bandwidth values and the application speedup due to hardware prefetchers strongly resemble each other on two "small-world" graph datasets (*Twitter* and *PLD*) for corresponding graph applications. Indeed, the graph application performance and the measured resource usage are strongly impacted by the type of the graph dataset used for processing. For the "small-world" datasets *Twitter* and



(a) Twitter (b) Roads (c) PLD

Fig. 4. Applications speedup across three datasets executed on 10-core configuration with different hardware prefetchers enabled.

PLD, the *PageRank* application achieves highest memory bandwidth on both datasets: 27-28 GB/s. *BFS* and *Betweenness Centrality* utilize 16-22 GB/s, followed by *Connected Components* and *Approximate Diameter* applications.

While the three datasets used in the study are quite different, the measured memory bandwidth scales in a similar way for selected graph applications processed with an increased number of cores. For 10 cores, memory bandwidth is increased 6-8 times compared to 1-core configuration. The memory bandwidth characterization with the pChase benchmark in Section III demonstrates 44 GB/s peak bandwidth on 10 cores with hardware prefetchers enabled. Therefore, apparently all five graph applications *do not fully utilize available memory bandwidth* in the system for all datasets used in the study, and *memory bandwidth is not a bottleneck*.

C. Benefits of Different Prefetchers: In-Depth Analysis

There is a variety of different hardware prefetchers available to be used in Intel processors. We performed in-depth analysis of the following four hardware prefetchers [4] available on the Ivy Bridge processor:

- *L2 Streamer* - it monitors requests from the L1 cache for ascending or descending access sequences. If there are too many outstanding requests or cache lines are too far ahead, it prefetches to LLC only.
- *L2 Spatial Prefetcher* - it strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk.
- *L1D Streamer* - it identifies streaming memory accesses, triggered by ascending access to a recently loaded data.
- *L1D Instruction Pointer Prefetcher (IPP)*- it tracks individual load instructions to detect patterns. If a stride is found, the next address will be prefetched (which is a sum of the current address and the stride).

Examining each prefetcher contribution separately, we see different characteristics. Figures 4 (a)-(c) show the average performance gains of each prefetcher for three dataset respectively, while the other prefetchers are disabled.

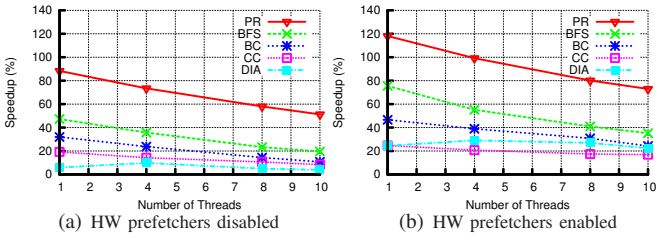
L2 Streamer and *L1D IPP* provide the biggest improvements. In some cases, it would be enough to use only one of them. Since prefetchers generate speculative traffic, they may (potentially) reduce application performance. A simple example is when a useful cache lines are evicted for allocating speculative lines which are never used. At the same time, for all the selected applications and considered hardware prefetchers, we observe positive performance improvements.

In summary, a joint work of the four hardware prefetchers available on the Ivy Bridge processor provides significant application performance benefits.

D. Benefits of Transparent Huge Pages

An efficient way to enable the system to manage large amount of memory is to increase the page size. The use of huge pages covers a larger memory range than small pages, and can potentially reduce the number of TLB misses. Moreover, in case of a TLB miss, fewer page table levels are traversed to translate the page address. This reduces the TLB miss penalty. Modern processors usually contain a smaller number of TLB entries for huge pages. In Ivy Bridge, there are 64 and 32 DTLB entries for 4 KB and 2 MB pages respectively. Usually, to efficiently use huge pages, the application code requires significant changes.

In this paper, we utilize the Transparent Huge Pages (THP) support in Linux, which hides much of the complexity by providing an abstraction layer that manages huge pages automatically. Figure 5 (a) presents the application speedup achieved by using THP (i.e., 2 MB pages) versus traditional 4 KB pages when processing the *Twitter* dataset with *hardware prefetchers disabled*.



(a) HW prefetchers disabled (b) HW prefetchers enabled

Fig. 5. Application speedup when using 2 MB pages compared to traditional 4 KB pages on the *Twitter* dataset.

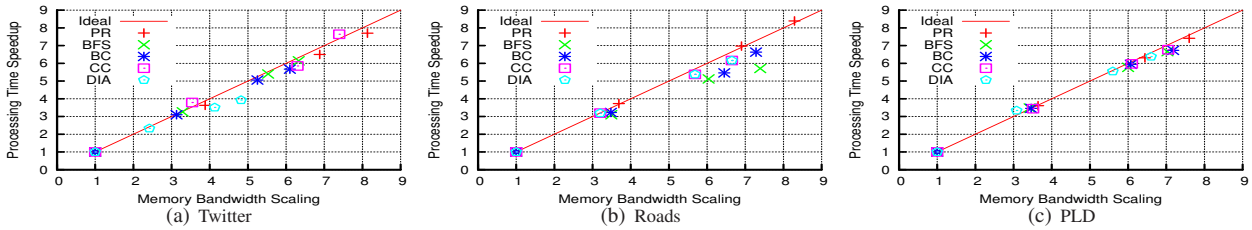
For PageRank, BFS and BC applications we can see very significant performance gains due to use of huge pages. The speedup is higher when the application is executed with a smaller number of concurrent threads.

As expected, one factor for the performance gains of using THP is the reduction of TLB misses. Table V shows some interesting details for TLB miss rates and page walk durations, both with traditional 4 KB pages and 2 MB huge pages. There is a significant reduction of 30%-63% in the TLB misses, and the page walk duration is reduced by 50%-70%, across **all** selected applications.

App	Page Walk Duration (cycles)			TLB Miss Rate		
	4 KB	2 MB	Reduction(%)	4 KB	2 MB	Reduction (%)
PR	51.54	15.79	69.35%	16.4%	10%	39.11%
BFS	40.34	14.62	63.74%	7.41%	2.75%	62.83%
BC	42.42	13.65	67.82%	4.16%	2.43%	41.52%
CC	44.40	13.21	70.25%	3.16%	1.60%	49.20%
DIA	31.07	15.03	49.83%	0.76%	0.52%	30.22%

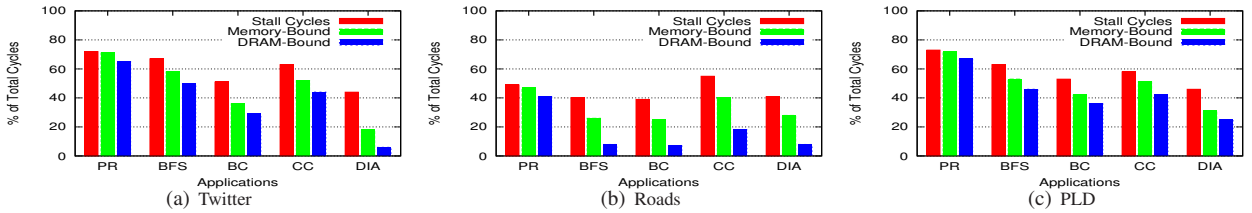
TABLE V. AVERAGE PAGE WALK DURATIONS AND TLB MISS RATES WHEN USING 2 MB PAGES (THP) COMPARED TO 4 KB PAGES ON THE *Twitter* DATASET.

Another factor for application performance gains is the improved efficiency of hardware prefetchers as shown in



(a) Twitter (b) Roads (c) PLD

Fig. 6. Scalability: memory bandwidth scaling vs application speedup across three datasets with hardware prefetchers enabled.



(a) Twitter (b) Roads (c) PLD

Fig. 7. Analysis of execution stall cycles for a 10-core configuration across three datasets with hardware prefetchers enabled.

Table VI. We found that huge pages and hardware prefetchers in graph processing are deeply influenced by each other, and disabling either of them reduces the performance gain of the other: this can be clearly seen in Table VI and by comparing Figures 5 (a)-(b). A major reason is that hardware prefetchers are usually limited by the page boundaries, and this results in smaller opportunities for prefetching in case of 4 KB pages.

Page Size	*PR*	*BFS*	*BC*	*CC*	*DIA*
4 KB	4-7%	18-23%	21-25%	15-18%	53-63%
2 MB	21-22%	36-41%	35-41%	20-26%	61-78%

TABLE VI. HW PREFETCHERS GAIN ON THE *Twitter* DATASET, USING DIFFERENT PAGE SIZE CONFIGURATIONS.

In our previous paper [15], we opened up a discussion that some traditional stereotypes portrayed in the literature do not hold. In particular, against the popular belief that graph applications have a poor locality, our analysis [15] revealed that the studied graph applications exhibit a good data locality (*high L1 hit rates* and *moderate Last-Level Cache hit rates*).

Our current analysis and the observed performance improvements of hardware prefetchers combined with THP reveal that there is a significant amount of potential *spatial and temporal data locality* present in the graph data, which is underutilized during graph processing due to hardware prefetchers being "limited" by the (4 KB) page boundaries.

E. Graph Applications: Memory Bandwidth-Bound or Memory Latency-Bound?

In our experiments with an increased number of cores for processing the selected five applications, we observed a very good application speedup: 6-8 times is achieved on 10 cores compared to 1 core performance. Figures 6 (a)-(c) show memory bandwidth scaling vs the application speedup. X-axis reflects memory bandwidth scaling with respect to 1 core-configuration, while Y-axis shows the corresponding application speedup under the same configuration.

The red line in Fig. 6 shows the ideal correlation between memory bandwidth scaling and the application speedup. Note, that all the points in these figures follow closely the diagonal line. This shows a *very strong correlation between memory bandwidth scaling and the application speedup*. This leads us to a natural question: does this mean that the application performance is memory bandwidth-bound?

In order to answer this question, we analyze the percentage of execution stall cycles during the application computation, and provide the stall cycles' breakdown with respect to a system functionality that caused the observed stalls, by utilizing the methodology described in section II for memory-bound and DRAM-bound. Figures 7 (a)-(c) show three bars for each profiled application. The red bar represents the total percentage of execution stall cycles during the application processing, i.e., cycles when the corresponding processor core executes nothing. The green bar shows the percent of stall cycles that are likely caused by the memory subsystem. Note, that the memory hierarchy includes a set of caches (L1, L2, LLC) and DRAM. The last blue bar reflects the percentage of stall cycles that are likely caused by outstanding DRAM references.

First, let us analyze the breakdown of stall cycles for the scale-free graph datasets *Twitter* and *PLD*. The percentage of stall cycles is high across all the applications: reaching 71% for PageRank and being above 60% for *BFS* and *CC*. The next green bar shows that the most stall cycles are likely to be caused by the memory subsystem (except the DIA execution on the *Twitter* dataset). The last blue bar provides an additional insight that the stall cycles due to outstanding DRAM references represent the majority of stall cycles in the memory hierarchy, and it means that the processor is waiting for the outstanding memory references to be served (the *DIA* execution on the *Twitter* dataset suffers from the unbalanced processing across the cores, and the measured averages do not convey the accurate story).

The earlier results, shown in Figures 2 (a)-(c), indicate that all five graph applications utilize memory bandwidth significantly less than 60% of its peak. Combining these observations with the analysis of stall cycles breakdown, we can conclude that for scale-free graphs the considered graph applications *are not memory bandwidth-bound* (as often assumed in literature) but are rather *memory latency-bound*.

For the *Roads* dataset the situation is slightly different. While for PageRank the same observation holds as for scale-free graphs, the remaining four applications have a different stall cycles breakdown. The fraction of stall cycles caused by the memory subsystem is still high. However, the stall

cycles due to outstanding L1, L2, LLC references represent the majority of stall cycles in the memory hierarchy (partially, because *Roads* is a smaller dataset).

F. Utilized LFBs and Back-End Allocation Stalls

For the *pChase* benchmark executed by a single core, 10 available Line Fill Buffers were the limiting factor for utilized memory bandwidth and were the system bottleneck as shown in Section III. Table VII presents the average LFBs occupancy across configurations with 1, 4, 8, and 10 cores and three processed datasets (we show combined results for two scale-free datasets due to their similarity).

Dataset	*PR*	*BFS*	*BC*	*CC*	*DIA*
Twitter and *PLD*	4.7-5.5	3.3-3.5	1.8 -2.2	1.4-1.6	0.2-1
Roads	0.9-1	0.3	0.4	0.2-0.4	0.3-0.4

TABLE VII. LFB OCCUPANCY (FOR 1,4,8,AND 10-CORE CONFIGURATIONS).

In a recent paper [11], the authors perform a bottleneck analysis of graph processing workloads on Intel processors. They observe that (on average) memory bandwidth is not fully utilized and the average LFBs occupancy is low. Their hypothesis is that the current size of *Reorder Buffer (ROB)* in the Intel processors might play a role of a limiting resource, i.e., limit the number of outstanding memory references, the LFB occupancy, and memory bandwidth. Below we present our analysis and question the validity of the hypothesis offered in [11] .

Indeed, when the instruction window of an out-of-order processor is not capable to accommodate enough *load* operations in parallel, it might limit the LFB occupancy and as a result, decrease the memory bandwidth. In modern processors, the pipeline contains an *in-order front-end* which fetches and decodes instructions from the path, that the program will most likely execute, and provides them to the *out-of-order back-end*. The back-end contains:

- *Reorder Buffer (ROB)*, which tracks the program order and updates the architectural state in order.
- *Reservation Station (RS)*, which schedules and dispatches ready operations to the execution unit.

When the ROB or the RS become full, the allocation of new instructions to the back-end becomes stalled. ROB typically becomes full due to a non-completed instruction holding all "younger" instructions from retiring, as retirement must be done in-order. RS might become full when it is waiting for inputs and resources to become available.

Using counters 19-20 in Table I, we track the stalls due to ROB and RS **being full**. Table VIII presents the *allocation stall* rates due to **full** RS and ROB across the three datasets. Comparing the allocation stall rates with the

Dataset	*Alloc. Stalls*	*PR*	*BFS*	*BC*	*CC*	*DIA*
Twitter	*RS*	9-13%	8-10%	11-16%	4-6%	1-5%
	ROB	24-26%	9%	4-5%	18-20%	2-5%
Roads	*RS*	4-5%	11-19%	17-24%	5-7%	16-25%
	ROB	52-56%	7-9%	6-7%	6-8%	10-13%
PLD	*RS*	8-10%	10-13%	12-16%	4-6%	4-6%
	ROB	33-35%	11-12%	6-7%	17-19%	4-5%

TABLE VIII. RS AND ROB STALLS FOR 1,4,8,10-CORE CONFIGURATIONS.

LFB occupancy rates in Table VII suggests that allocation stalls are not likely a major factor limiting LFB occupancy in graph processing. For most of the applications under study, the low LFB occupancy coincides with very low stall rates when ROB is being full.

Many graph applications have a time-varying behavior during processing, imposed by the graph properties and related computations. It can stretch system resources usage to their peak over short time intervals and change the observed bottlenecks over time, as we discuss in the next section.

G. The Danger of Using Averages for the Bottleneck Analysis and Resource Usage Estimates

A traditional profiling approach withn the PAPI tool is to measure and report average values of counters collected during the application execution time interval: $[app\_begin, app\_end]$. However, we believe that average values could be misleading for the bottleneck analysis and quantifying the resource usage. The average values do not expose the execution "imbalance", especially along multiple threads and cores, and limit useful insights into the actual program resource usage over time. We have added a sampling feature for measuring values of performance counters over time. We performed two sets of experiments, where selected performance counters are sampled every 100 ms and 10 ms.

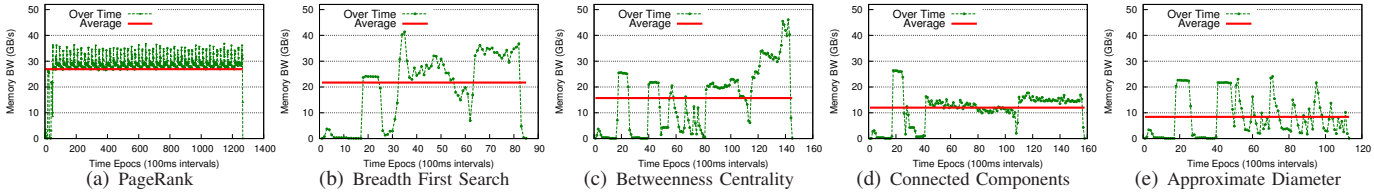
Figures 8-9 show the time-varying memory bandwidth utilization for selected graph applications under study, with 100 ms and 10 ms sampling epochs respectively. The solid red line presents the reported average memory bandwidth utilization, while the dotted green line reflects the measured memory bandwidth over time. Apparently, the use of average values provides a very inaccurate report of application memory bandwidth requirements. In case of PR, BFS and BC applications, there are prolonged time intervals with close to the peak memory bandwidth utilization. The resource usage burstiness is even more pronounced at 10 ms epochs.

In order to provide an "easy to grasp" summary of measured values over time, we compute CDF (Cumulative Distribution Function) of collected counter values over time. A set of computed CDFs for selected metrics defines an *application signature* which can be used for compact characterization of the application and for comparing different applications and their resource requirements. Fig. 10 (a)-(e) presents an example of such a signature for BFS application (with 100 ms and 10 ms epochs). Five metrics include memory bandwidth, LFB occupancy, IPC, execution stall cycles, and memory-bound stall cycles. For example, it shows, that for almost 90% of time intervals the IPC is very low (below 0.5). However, for remaining 10% of time IPC is close to 1.7. Similarly, for 60% of the time, 80% of cycles are the execution stall cycles, and these stalls are mostly due to memory references.

CDFs with 10 ms sampling epochs show very interesting details for the memory bandwidth usage and LFB occupancy, e.g., 10% of the time 6 to 10 Line Fill Buffers are in use, while with 100 ms sampling the maximum LFB occupancy is 6.2. It shows that usage of these resources is very bursty. The use of average values provides a very inaccurate report.

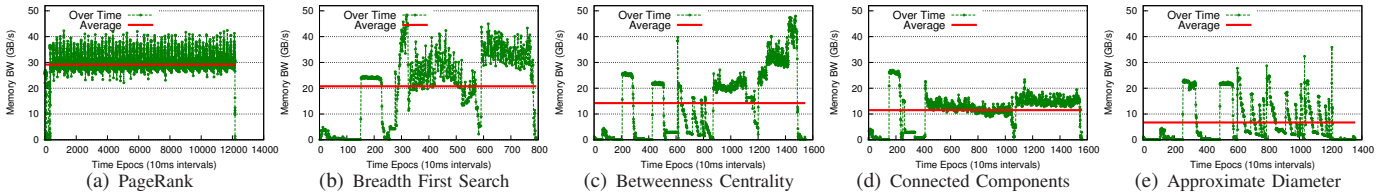
V. RELATED WORK

Due to a renewed interest to large graph analysis and applications, a variety of graph processing frameworks [2], [19], [21], [24], [28], [29], [31] were offered to simplify the implementation of graph algorithms and help with parallel and distributed graph processing. Due to a difficulty to objectively compare different frameworks, a few research groups [17], [22], [12] set a higher level goal of creating



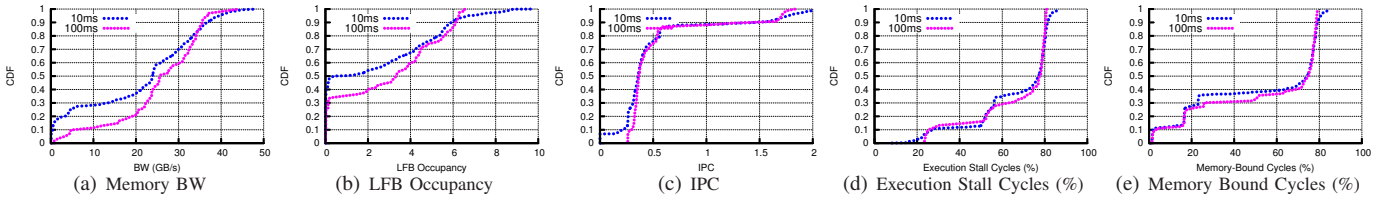
(a) PageRank (b) Breadth First Search (c) Betweenness Centrality (d) Connected Components (e) Approximate Diameter

Fig. 8. Memory bandwidth utilization over time: 100 ms epochs, 10-core configuration, Twitter dataset (HW prefetchers enabled).



(a) PageRank (b) Breadth First Search (c) Betweenness Centrality (d) Connected Components (e) Approximate Diameter

Fig. 9. Memory bandwidth utilization over time: 10 ms epochs, 10-core configuration, Twitter dataset (HW prefetchers enabled).



(a) Memory BW (b) LFB Occupancy (c) IPC (d) Execution Stall Cycles (%) (e) Memory Bound Cycles (%)

Fig. 10. CDF of selected metrics for BFS application processed with 10-core configuration and a Twitter dataset: 10ms vs 100ms epochs.

a common large graph-processing benchmark, which can be used for measuring and comparing the existing platforms.

At the same time, a parallel set of efforts [28], [30], [16] in research community aim to compare the efficiency of existing frameworks in order to understand their benefits, applicability, and performance optimization opportunities by using a set of popular, well-known graph algorithms. For example, the authors in [30] provide a native, hand-optimized implementation of four graph algorithms to set a base for performance expectations, and then use them as a reference point. According to the paper results, the Galois system (which we use in our study) shows a very close to optimal performance. In this paper [30], the authors refer to either memory system or network being a bottleneck for different frameworks and their configurations. While many evaluation studies hint on the memory system being a bottleneck, they do not provide a detailed workload analysis of how a memory subsystem is used and what are the causes of the system inefficiencies during large graph processing. Our paper aims to provide this missing analysis and insights.

There are quite a few large-scale system based studies that consider different hardware architecture design for efficient large graph processing. The system and memory latencies in such distributed systems are very different compared to a single multi-core commodity system used in our study. Thus, we do not aim to generalize our observations and results to such systems. At the same time, there are interesting intersections and observations in spite of these differences. Paper [10] compares performance of two graph algorithms on symmetric multiprocessors (SMPs), such as Sun Enterprise servers, and multi-threaded architectures (MTAs), such as Cray MTA-2. The authors discuss performance benefits of caching and multithreading for graph algorithms. Caching is typically beneficial when data exhibit some spatial or temporal locality, while multithreading is often applied for creating a higher processing concurrency and used as a "memory latency hiding" technique. To minimize the memory access latency, the authors in [10] make a conclusion that flat, shared-memory multiprocessors, such as Cray MTA-2, are more suitable than SMPs that rely on caching and prefetching technique. In [27], the authors share a similar view that graph analytics exhibits little locality. They believe that the efficient system for processing large graphs should be able to tolerate higher latencies with higher concurrency. While we agree that higher concurrency and latency hiding techniques are desirable, our findings show that graph algorithms can benefit from hardware prefetching due to significant data locality, which should be exploited. The authors in [27] outline a high-level system design, where multiple nodes (based on commodity processors) communicate over an InfiniBand network, manage high number of concurrent threads, and may efficiently serve memory requests to the global memory space shared across the nodes. The authors justify the proposed solution by evaluating the achievable performance with a pointer chasing benchmark, which is similar to pChase used in our study. As we have shown in our paper, concurrent independent memory chains generated by pChase could deliver much higher memory bandwidth compared to the real graph processing applications that have additional dependencies limiting the available parallelism in the program. It will be very interesting to see a detailed evaluation of the proposed system on real graph applications.

The authors of the paper [13] follow a similar approach that multithreading should help in hiding memory latency. At the same time, by studying IBM Power7 and Sun Niagara2 they make observations that the number of hardware threads in either platform is not sufficient to fully mask memory latency. Our experiments with graph processing on modern Ivy Bridge-based servers expose a similar behavior that the available concurrency cannot efficiently hide the incurred memory latency in the system. In addition, papers [13], [9] investigate explicit software prefetching for improving graph processing performance, because they believe that hardware prefetchers might not be effective. However, implementing

software prefetching for a diverse set of graph algorithms might be very difficult since this approach requires human intervention and insights. On the contrary, our study reveals that hardware-based prefetching can be quite efficient and does improve graph algorithms performance by 15%-78%.

In a recent paper [11], the authors perform the analysis of graph processing workloads on modern servers. Some of their findings are similar to our results in the earlier work [15] (also mentioned in the current paper). The authors observe that there is a locality in graph workloads and that memory bandwidth is not fully utilized. Their hypothesis is that other bottlenecks, such as the size of *Reorder Buffer (ROB)* limits the number of outstanding memory references and memory bandwidth as a result. In our paper, we quantify how often the processor is stalled due to a "ROB is full" condition. These results question whether a ROB size is a bottleneck and an increased ROB size will alleviate the problem. We also show that the authors expectations on a limited temporal variation in graph processing behavior do not always hold. We stress the danger of considering the "averages" for characterizing system bottlenecks and demonstrate high differences in resource usage over time. In particular, we show significant variations in Line Fill Buffers occupancy and achievable memory bandwidth over time for multiple graph applications used in the study. Finally, we demonstrate and quantify the significant benefits of hardware prefetching, especially combined with the *transparent huge pages* use, against the existing traditional stereotypes.

VI. Conclusion and Future Work

In this paper, we characterize and quantify how parallel graph applications utilize underlying resources in the latest Intel multi-core systems with a focus on interworking of processor and memory. Contrary to traditional past stereotypes, we found that graph applications significantly benefit from hardware prefetchers and achieve a good speedup with an increased number of cores. This happens due to complementary reasons: *i) advances in the modern hardware* offering a smart collection of powerful prefetchers, and *ii) new software approaches* implemented in Galois: a novel runtime library that performs efficient task scheduling and aims to minimize synchronizations and communications between threads to optimize and increase the application execution parallelism. Moreover, we show that the THP use significantly boosts the performance of hardware prefetchers. This reveals the additional spatial data locality in graph data, which is underutilized during graph processing by hardware prefetchers being "limited" by the default 4 KB page boundaries.

Using hardware performance counters is a labor-intensive task. It involves careful counters selection, their thorough validation, and an instrumentation of the analyzed programs. Moreover, it is a "black-box" approach: it does not provide an additional useful feedback on effectiveness of program data structures or the program code. In our future work, we would like to connect the analysis of the profiled application with the collected performance metrics and measurements. We plan to merge our profiling approach and functionality of open-source HPCToolkit [3], which can be used for scalability analysis and detailed resource usage profiling of emerging Big Data analytics for processing on modern multiprocessor machines.

Acknowledgements

We are extremely grateful to the Galois team (Andrew Lenharth, Muhammad Amber Hassaan, and Roshan Dathathri) for their generosity in sharing the graph applications' code and patiently answering our numerous questions over the cause of the project. Our sincere thanks to Anirban Mandal, Rob Fowler, and Allan Porterfield (the authors of [25]) for sharing the pChase code and their help with related scripts for measuring the memory system performance.

References

[1] 9th DIMACS Implementation Challenge: Shortest Paths. http://www.dis.uniroma1.it/challenge9/.

[2] Apache Giraph. http://giraph.apache.org/.

[3] HPCToolkit. http:/http://hpctoolkit.org/.

[4] Intel 64 and ia-32 architectures optimization reference manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[5] libpfm4. http://perfmon2.sourceforge.net/.

[6] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[7] pChase. https://github.com/maleadt/pChase.

[8] STREAM benchmark. http://www.cs.virginia.edu/stream/.

[9] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proc. of the 2010 ACM/IEEE Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[10] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP)*, 2005.

[11] S. Beamer, K. Asanovicn, and D. Patterson. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *Wiil appear in Proc. of the Intl. Symposium on Workload Characterization (IISWC)*, Oct, 2015.

[12] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proc. of the GRADES'15*, 2015.

[13] G. Cong and K. Makarychev. Optimizing Large-Scale Graph Analysis on a Multi-threaded, Multi-core Platform. In *Proc. of the 2011 IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[14] Duncan J. Watts. Small Worlds: The Dynamics of Networks Between Order and Randomness. Princeton University Press, 2003.

[15] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, and S. Katti. Parallel Graph Processing: Prejudice and State of the Art. In *Proc. of the 7th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*, 2016.

[16] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Benchmarking Graph-processing Platforms: A Vision. In *Proc. of the 5th ACM/SPEC Intl. Conference on Performance Engineering (ICPE)*, 2014.

[17] M. Han et al. An Experimental Comparison of Pregel-like Graph Processing Systems. *Proc. VLDB Endow.*, 7(12), 2014.

[18] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proc. of 19th Intl. Conference on World Wide Web*, 2010.

[19] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[20] A. Lenharth and K. Pingali. Scaling Runtimes for Irregular Algorithms to Large-Scale NUMA Systems. *IEEE Computer Journal*, Aug, 48, 2015.

[21] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8), Apr. 2012.

[22] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.*, 8(3), 2014.

[23] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1), 2007.

[24] G. Malewicz et al. Pregel: A System for Large-scale Graph Processing. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 2010.

[25] A. Mandal, R. Fowler, and A. Porterfield. Modeling Memory Concurrency for Multi-Socket Multi-Core Systems. In *Proc. of Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[26] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna. Extracting the Hyperlink Graphs from the Common Crawl. http://webdatacommons.org/hyperlinkgraph.

[27] J. Nelson et al. Crunching Large Graphs with Commodity Processors. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, 2011.

[28] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight Infrastructure for Graph Analytics. In *Proc.of 24th Symp. on Operating Systems Principles, (SOSP)*, 2013.

[29] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Proc. of 25th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, 2013.

[30] N. Satish et al. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proc. of Intl. Conference on Management of Data, (SIGMOD)*, 2014.

[31] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP)*, 2013.

[32] A. Yasin. A Top-Down Method for Performance Analysis and Counters Architecture. In *Proc. of Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.