Parallel Graph Processing: Prejudice and State of the Art

Assaf Eisenman[1,2], Ludmila Cherkasova[2], Guilherme Magalhaes[3], Qiong Cai[2],
Paolo Faraboschi[2], Sachin Katti[1]

[1]Stanford University, [2]Hewlett Packard Labs, [3]Hewlett Packard Enterprise

assafe@stanford.edu, {lucy.cherkasova, guilherme.magalhaes, qiong.cai, paolo.faraboschi}@hpe.com,
skatti@stanford.edu

Abstract

Large graph processing has attracted much renewed attention due to its increased importance for a social network analysis. The efficient parallel graph processing faces a set of software and hardware issues, discussed in literature. The main cause of these challenges is the "irregularity" of graph computations and related difficulties in efficient parallelization of graph processing. Unbalanced computations, caused by uneven data partitioning, can affect application scalability. Moreover, the issue of poor data locality is another major concern, that makes the graph processing applications memory-bound. In this paper[1], we aim to profile how large, parallel graph applications (based on Galois framework) utilize modern systems, in particular, memory subsystem. We found that modern graph processing frameworks executed on the latest Intel multi-core systems (a single node server) exhibit a good data locality and achieve a good speedup with an increased number of cores, contrary to traditional past stereotypes. The application processing speedup is highly correlated with utilized memory bandwidth. At the same time, our measurements show that the memory bandwidth is not a bottleneck, and the analyzed graph applications are memory-latency bound. These new insights can help us in matching the resource demands of the graph processing applications to future system design parameters.

Categories and Subject Descriptors: C.4 [Computer System Organization] Performance of Systems, D.2.6.[Software] Programming Environments.

General Terms: Measurement, Performance, Design.

Keywords: Parallel graph processing, benchmarking, profiling, hardware performance counters.

1. INTRODUCTION

The interest to large graph processing has gained momentum over last years due to the increased importance of efficient graph processing for the analysis and problem solving in social networks, data mining, and machine learning. The steep increase in volume of data being produced led to a renewed interest in parallel graph processing. The rise of multi-core processors and their dominance in modern Data Centers offers new challenges and opportunities for efficient use of this platform for large graph processing. Unfortunately, most of the graph algorithms have some inherent characteristics that make them difficult to parallelize and execute efficiently.

The detailed survey paper [14] on challenges in parallel graph processing lists a set of software and hardware issues that limit graph processing performance. Among them the **"irregularity"** of graph computations, which makes it difficult to parallelize graph processing by either partitioning the algorithm computation or partitioning the graph data. Moreover, unbalanced computations, caused by uneven data partitioning, can affect and **limit the achievable scalability**. Finally, the issue of **poor data locality** during graph processing is another major concern and challenge described in [14]. All these challenges are side effects of the "irregular" algorithms, which are typically data-driven, with dependencies between tasks and computations defined at runtime.

To better understand the design points of the various future hardware and software components, we have to analyze and investigate a set of workloads that can drive the system design and implementation. In this paper, we aim to profile how large parallel graph applications utilize underlying resources in modern systems (a single server, based on Intel Xeon Ivy Bridge processor). We are especially interested in performance analysis of the memory subsystem and related system bottlenecks caused by parallel graph processing. This understanding can help us in matching the resource demands of the graph processing applications to future system design parameters.

For our study we have chosen the Galois system [18, 11], which was specially designed for parallel processing of irregular algorithms. The Galois framework was successfully applied for parallelizing graph algorithms, which exhibit similar properties. Our profiling approach takes advantage of hardware performance counters implemented in the Ivy Bridge processor. It leverages performance events from the processor Performance Monitoring Units (PMUs), both inside the core (i.e., execution units, L1 and L2 caches) and outside the cores (i.e., L3 cache, Memory Controller).

We analyzed five popular graph algorithms executed on two large datasets. We found that some of the traditional stereotypes portrayed in the literature do not hold, and that many irregular algorithms processing issues have been successfully tackled by a novel run-time support and task scheduling introduced in Galois. This makes the Galois' approach even more attractive and interesting for parallel graph processing. The key findings are the following:

- The available *memory bandwidth is not a bottleneck*: it is not fully utilized;
- Applications achieve a *good processing speedup* with an increased number of cores in a socket;
- The *speedup (scalability)* of graph applications is highly correlated with utilized memory bandwidth;

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE 2016, March 12-18, 2016, Delft, Netherlands.
© 2016 ACM. ISBN 978-1-4503-4080-9/16/03 ...$15.00
DOI: http://dx.doi.org/10.1145/2851553.2851572.

- The analysis of execution stall cycles in the system shows that graph processing is *memory-latency bound*;
- Graph applications exhibit *high L1 hit rates* and *significant Last-Level Cache (LLC) hit rates*. This reflects a good data locality that could be efficiently exploited.

The remainder of the paper presents our results in more detail.

2. OUR PROFILING APPROACH

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

2.1 Parallel Graph Processing Framework

Why Galois?.

Out of many available graph processing frameworks (e.g., GraphChi[10], GraphLab [12], Apache Giraph [1], and Ligra [22], just to name a few) we chose Galois [18], which was designed as a system for automated parallelization of irregular algorithms. Since graph processing highly resembles irregular algorithms Galois system 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 coordinated and autonomous execution of these tasks and allows application-specific control of scheduling policies (application-specific task priorities). In the recent study [21], conducted by independent researchers, graph algorithms implemented in Galois have been shown as highly competitive compared to a manually crafted and 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).

Selected Graph Applications.

For our profiling study, we selected five popular graph algorithms implemented in Galois [18] with different characteristics: some of them are *i)* traversal, i.e., topology driven, or *ii)* data-driven. Below is a short summary of these algorithms:

- **PageRank:** this algorithm is used by search engines to rank websites for displaying the output results. PageRank offers a way of measuring the importance and popularity of website pages.
- **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.
- **Betweenness Centrality (BC):** this algorithm measures the importance of a node in a graph. In social networks analysis, it is actively 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.
- **Connected Components (CC):** this algorithm identifies the maximal sets of vertices reachable from each other in an undirected graph.
- **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 rather provide a diameter approximation.

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 [18, 21]. Using an optimized and tuned code in our study is very important for profiling and understanding the real system bottlenecks during large graph processing (rather than discovering the bottlenecks related to an inefficiently written code).

2.2 Experimental hardware platform

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

- **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 last level cache. The system is equipped with 128 GB DDR3-1866 DRAM (i.e., with 4 DDR3 channels).

There are a few challenges 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. However, for energy efficiency many processors are applying DVFS (Dynamic Voltage and Frequency Scaling), where the processor frequency can be increased or decreased dynamically during workload processing depending on the system utilization. Therefore, to preserve a fixed ratio of cycles per time unit we **disable** Turbo Boost and DVFS feature.

We **disable** hyper-threading in our experiments in order to analyze application performance as a function of an increased number of physical cores assigned to application 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 coherency traffic and NUMA considerations[2].

2.3 Profiling system resource usage with hardware performance counters

As a part of our profiling approach we leverage the Performance Monitoring Units (**PMUs**) implemented in Ivy Bridge processors. We select a group of performance events for our analysis as shown in Table 1, using PMUs located inside and outside the cores. Last column provides an exact Intel event names, while the 2nd column shows mnemonic, short names for these events used in the paper.

The first two counters in Table 1 refer to events found in the integrated Memory Controller (**MC**). These events are read for each memory channel. $DRAM_{reqs}$ is used to count the number of outgoing MC requests issued to DRAM, while MC_{cycles} are 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 in the L1 data cache) until the corresponding data is retrieved from the memory hierarchy (caches or memory). Hence LFBs may limit the number of cache misses handled by the core.

Counters 4-5 are used to compute achievable *Instructions per Cycle (IPC)*.

Counters 6-11 are used for the analysis of *hit/miss rates in data caches L1,L2, and LLC*.

Counter 12 provides the total number of execution stall cycles

[2]Evaluating an additional NUMA impact on the performance of graph processing applications, analyzing bottlenecks and utilized memory bandwidth in multi-socket configuration is a direction for our future work.

incurred by the system. Counter 13 is used to measure the execution stall cycles caused by waiting for the memory system to return data (including caches), while counters 14 and 15 are used to categorize them into stall cycles caused by misses in the data caches L1 and L2, respectively.

Because stores typically do not delay other instructions directly, counters 6-15 concentrate on loads.

1	$DRAM_{reqs}$	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	$Stalls_{total}$	CYCLE_ACTIVITY:CYCLES_NO_EXECUTE
13	$Stalls_{mem}$	CYCLE_ACTIVITY:STALLS_LDM_PENDING
14	$Stalls_{L1}$	CYCLE_ACTIVITY:STALLS_L1D_PENDING
15	$Stalls_{L2}$	CYCLE_ACTIVITY:STALLS_L2_PENDING

Table 1: Selected performance events in Ivy Bridge processor family and memory subsystem.

In order to read performance counters, we use the PAPI [3] framework. PAPI provides a fully programmable, low level interface for dealing with processor hardware counters. We instrumented PAPI into the algorithm's 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.

We use $DRAM_{reqs}$ to count the number of outgoing MC requests issued to DRAM. Modern Intel processors have a memory line size of 64 bytes, thus we multiply the sum of $DRAM_{reqs}$ over the 4 memory channels by 64 to get the byte traffic sent to 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* application characterization, we use similar definitions to those described in the Intel Optimization Manual [2]. We define the memory bound metric as the cycles where the execution is stalled and there is at least one outstanding memory demand load:

$$Memory_bound = \frac{Stalls_{mem}}{Cycles}$$

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 $LLC_miss_fraction$. We 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 [2]:

$$LLC_miss_fraction = \frac{WEIGHT * LLC_{misses}}{LLC_{hits} + WEIGHT * LLC_{misses}}$$

3. SYSTEM PERFORMANCE CHARACTERIZATION

In order to analyze system bottlenecks while executing graph applications, we need to set realistic expectations on achievable performance (peak resource usage) of the system under study. In particular, we need to measure peak achievable memory bandwidth and memory latency, as well as to characterize how the memory bandwidth and memory access latency change under increased memory traffic issued by multiple cores.

In many traditional cases, peak memory bandwidth is measured using the STREAM benchmark [5]. These measurements characterize memory bandwidth achievable under the *sequential access pattern*. However, real graph applications, processed by current multi-core systems, exhibit a quite different access pattern, where concurrent threads, executed on different cores, utilize the memory system by issuing a set of *independent* memory references. We need to measure the ability of the memory system to serve a high number of *concurrent, random access memory operations* present in current multi-core systems and identify the related system bottlenecks.

To achieve this goal, we utilize an open source *pChase benchmark* [4] which was originally introduced by Pase and Enckl [20] to measure the memory latency and bandwidth of IBM systems. The enhanced version of pChase benchmark was successfully used for characterizing and modeling memory performance of modern multi-core and multi-socket systems [15]. pChase benchmark enables measuring both memory latency and throughput under controlled degree of issued concurrent memory references.

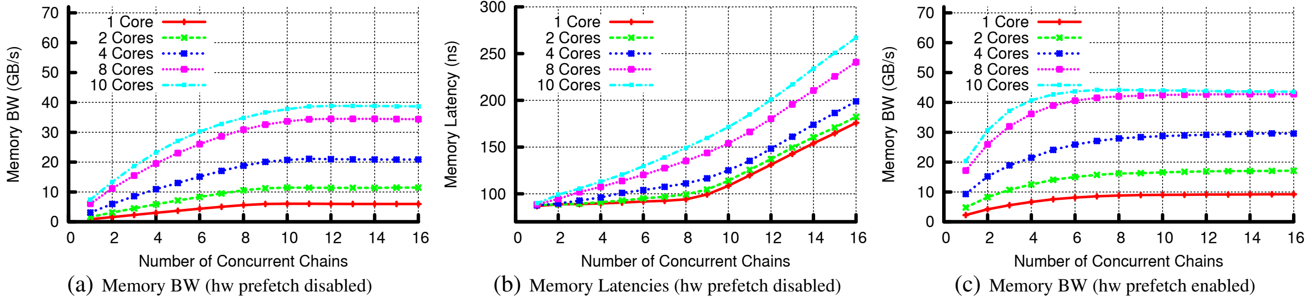
pChase is a memory-latency bound pointer-chasing benchmark. The benchmark creates a pointer chain, such that the content of each element dictates which memory location is read next. This ensures that the next memory reference cannot be issued until the result of the previous one is returned. The benchmark can create *multiple* independent chains per thread, with memory references from different chains issued concurrently.

pChase allocates a pointer chain in a page by page manner: each page is filled before proceeding to the next page. Using this pattern helps in minimizing TLB misses, which is important for accurate measurements of memory access latencies. Inside a page, it creates a chain between all its cache lines (data blocks) in a semi-random manner. However, this pattern is still partly prefetchable because there are partial strides. Only after finishing the page, pChase goes to the next page (it creates 64 pointers with size of 8 bytes in each 4 KB page). For accurate measurements of memory latency, we need to guarantee that all issued pointers are served from memory. To avoid prefetching and caching side effects it is important that **hardware prefetching is disabled** during pChase experiments in the system.

Figure 1 (a) shows achievable memory bandwidth (Y-axis) measured with hardware performance counters as described in Section 2.3. Five different lines reflect measurement results of 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 (executed by 1 core) achieves maximum 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**.

Figure 1 (a) shows that with two pChase threads and two cores achievable memory bandwidth increases perfectly to 11 GB/s, i.e., 2 times higher. However, for four pChase threads and four cores



(a) Memory BW (hw prefetch disabled) (b) Memory Latencies (hw prefetch disabled) (c) Memory BW (hw prefetch enabled)

Figure 1: Measured memory bandwidth and memory access latencies with pChase benchmark executed on 1, 2, 4, 8, and 10 cores, where each thread is configured with increasing number of concurrent pointer chains.

Dataset	*Brief Description*	*# Vertices*	*# Edges*	*Source*	*Reference*
Twitter	Twitter Follower Graph	61.5 M	1,458 M	http://an.kaist.ac.kr/traces/WWW2010.html	[9]
PLD	Web Hyperlink Graph	39 M	623 M	http://webdatacommons.org/hyperlinkgraph/2012-08/download.html	[16]

Table 2: Datasets: graphs used in the algorithm evaluation.

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.

Figure 1 (b) shows measured memory access latencies by pChase threads (1, 2, 4, 8, and 10 threads) with concurrent pointer chains. A line with a single pChase thread clearly shows the power of memory level parallelism (MLP): memory references from 9-10 concurrent chains could be processed with the same access latency of 86 ns by memory system. After all 10 LFBs are used, the core is stalled until the issued memory references are served by DRAM and the core's LFBs are released and made available for processing next outstanding memory references. Figure 1 (b) exhibits significantly increased memory latencies for pChase configurations with more than 4 concurrent threads. It is indicative of the increased contention in the memory system when a high number of cores are issuing concurrent memory loads. When 10 cores are issuing a maximum number of concurrent requests (with 10 LFBs full) the measured memory access latency is almost doubled. This increased memory latency and contention in memory system explains lower memory bandwidth scaling for higher number of cores in pChase benchmark as shown in Figure 1 (a).

Now, we demonstrate the importance of understanding and evaluating the **performance impact of hardware prefetching** that causes increased memory bandwidth usage as a result. Figure 1 (c) shows the achievable memory bandwidth (Y-axis) measured by 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 additional prefetch memory traffic decreases. Overall, hardware prefetching increases the achievable socket memory bandwidth (when all 10 cores execute pChase threads) by 12.8% and it reaches 44 GB/s.

4. EVALUATION

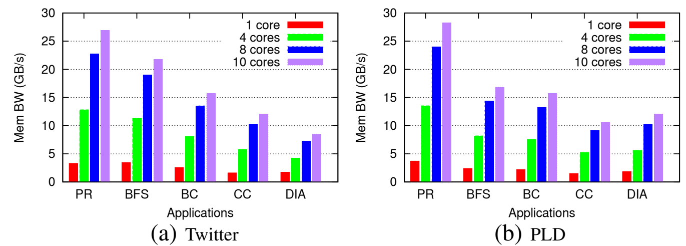
Graph Datasets.

In this section, we analyze the profiling results of five selected graph applications that are executed using two datasets described in Table 2.

Performance of graph applications may significantly depend on the structure and properties of the graphs used for processing. In this study, we choose to concentrate on processing graphs that belong to a category of *social networks*. Social networks are difficult to partition because they come from non-spatial sources. They are often called "small-world" graphs due to a low diameter. In "small-world" graphs, most nodes can be reached from each other by a limited number of hops. Another property of "small-world" graphs 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 majority of vertices are connected to fewer neighbors. Both datasets *Twitter* and *PLD*, shown in Table 2 and used in the evaluation study, are small-world graphs.

Utilized Memory Bandwidth.

First, we analyze memory bandwidth used by applications under study. We execute the selected graph applications with an increased number of cores in the configuration and profile the utilized memory bandwidth in the system. Figures 2 (a)-(b) show four bars for each profiled application. These bars represent average memory bandwidth measured during the application processing in configurations with 1, 4, 8, and 10 cores on two datasets: *Twitter* and *PLD* respectively. PageRank achieves highest memory bandwidth on both datasets: 27-28 GB/s. BFS and Betweenness Centrality



(a) Twitter (b) PLD

Figure 2: Average memory bandwidth (with prefetch enabled).

utilize 16-22 GB/s, followed by Connected Components and Approximate Diameter applications. Our memory characterization with pChase benchmark in Section 3 demonstrates 44 GB/s peak bandwidth on 10 cores with hardware prefetch enabled. Therefore, apparently all five graph applications *do not fully utilize available memory bandwidth* in the system.

While two 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 7-8 times compared to 1-core configuration.

Table 3 presents the average LFB occupancy (across configurations with 1, 4, 8, and 10 cores and two processed datasets respec-

tively). Clearly, this data shows that LFBs are not a system bottleneck. Here, they do not cause the memory bandwidth not being fully utilized.

Metrics	*PR*	*BFS*	*BC*	*CC*	*DIA*
LFB occupancy	4.7-5.5	3.3-3.5	1.8 -2.2	1.4-1.6	0.2-1
IPC	0.5-0.6	0.5-0.8	0.6-0.9	0.7-1	0.7-1.2

Table 3: Average LFB occupancy and IPC across 1, 4, 8, and 10 cores configurations and two datasets *Twitter* and *PLD*.

The IPC metric (Instructions per Cycle) is used to assess the computation efficiency of a processor by the application. The achieved IPC is low for all five applications. It is not-surprising: memory-bound applications typically have lower IPC.

Data Locality.

Table 4 presents surprising and unexpected results on measured cache hit rates for L1 and LLC. The measurements are performed with hardware prefetch disabled in order to observe the cache hit rates caused by application memory loads only. We can see that graph applications exhibit *high L1* and *significant LLC hit rates* that indicates a good data locality that could be efficiently exploited [3].

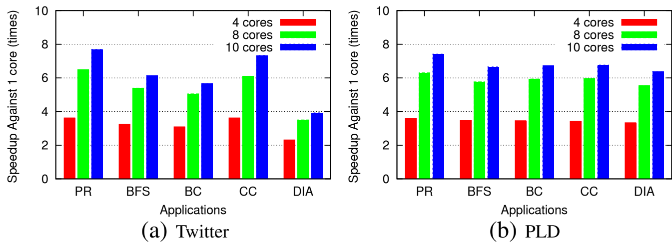
Metrics	*PR*	*BFS*	*BC*	*CC*	*DIA*
L1 hit rates	74-77%	89-90%	93-98%	95-96%	96-98%
LLC hit rates	35-39%	34-37%	30-33%	29-31%	10-22%

Table 4: Cache Hit Rates (L1 and LLC) across 1, 4, 8, and 10 cores configurations and two datasets *Twitter* and *PLD*.

These are very interesting results reflecting that traditional stereotypes about poor data locality do not hold for modern graph processing frameworks executed on the latest Intel multi-core systems. [4].

Application Scalability.

Figures 3 (a)-(b) show the application speedup for processing in configurations with 1, 4, 8, and 10 cores on two datasets: *Twitter* and *PLD* respectively. All the applications (except DIA on *Twitter*) show a very good speedup: 6-8 times is achieved on 10 cores compared to 1 core performance. The analyzed applications do *show a good scalability* as a function of increased compute resources.



(a) Twitter (b) PLD

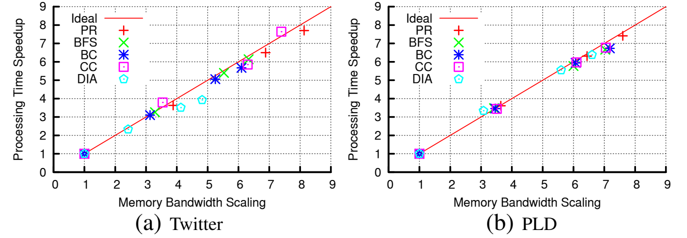
Figure 3: Application scalability: speedup compared to 1 core-configuration performance (with prefetch enabled).

By comparing the memory bandwidth scaling trends in Figure 2 with application speedup shown in Figure 3, one can see that these trends are correlated. Figures 4 (a)-(b) show memory bandwidth scaling vs application speedup. X-axis reflect memory bandwidth scaling with respect to 1 core-configuration, while Y-axis show the corresponding application speedup under the same configuration.

The red line in Figures 4 (a)-(b) shows the ideal correlation between memory bandwidth scaling and application speedup. Note, that all the points in these figures follow closely the diagonal line.

[3] L2 cache counters had some issues, and we omit reporting their results.

[4] In our experimental system, the size of LLC is 25 MB. Therefore, the application working set for both graphs in Table 2 cannot be cached since they significantly exceed available LLC size.



(a) Twitter (b) PLD

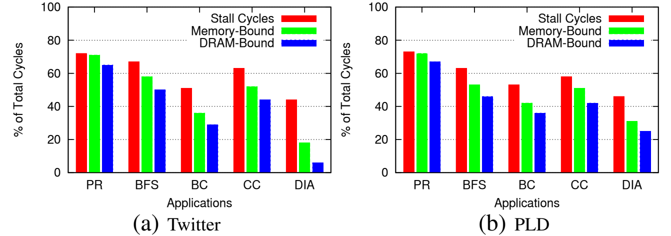
Figure 4: Memory bandwidth scaling vs application speedup (with prefetch enabled).

This shows a *very strong (almost ideal) correlation between memory bandwidth scaling and application speedup*.

This leads us to a natural question: does this mean that the application performance is memory bandwidth-bound?

Memory bandwidth-bound or memory latency-bound.

In order to answer this question, we analyze the percentage of execution stall cycles during the application computation, and provide stall cycles' breakdown with respect to system functionality that caused the observed stalls. Figures 5 (a)-(b) 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 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 most execution stall cycles are caused by the memory subsystem (except DIA execution on the *Twitter* dataset). We refer the reader to Section 2.3 for definitions of memory-bound and DRAM-bound metrics.



(a) Twitter (b) PLD

Figure 5: Execution Stall Cycles (10 cores, with prefetch enabled).

Note, that the memory hierarchy includes a set of caches (L1, L2, and LLC) and DRAM. The last blue bar provides an additional insight that execution stall cycles due to outstanding DRAM references represent the majority of stall cycles in the memory hierarchy (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)-(b)) 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 considered graph applications *are not memory bandwidth-bound (as often assumed in literature)* but are rather *memory latency-bound*.

5. RELATED WORK

In the past few years, graph algorithms have received much attention and have become increasingly important for meaningful analysis of large datasets. A number of different graph processing frameworks [1, 10, 12, 18, 22] were offered for optimized parallel and distributed graph processing. This caused multiple efforts [18, 21, 8] in research community to compare the efficiency of these frameworks in order to understand their benefits, applicability, and performance optimization opportunities. For setting a reasonable base for performance expectations, the authors in [21] provide a native,

hand-optimized implementation of four algorithms and use them as a reference point. According to the paper results, the Galois system (which we chose for our study) shows very close to optimal performance. In this study [21], the authors refer to either memory system or network being a bottleneck for different frameworks and their configurations. In related studies [13, 6], the authors strive towards creating a benchmark for comparing graph-processing platforms. 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 system inefficiencies during large graph processing. Our paper aims to provide this missing analysis and insights.

In [17], the authors share a view that graph analytics algorithms exhibit little locality and therefore present significant performance challenges. They asssume that graph-processing algorithms are memory-latency bound and the efficient system for processing large graphs should be able to tolerate higher latencies with higher concurrency. They 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 latest HP Labs project "The Machine" [19] promotes a similar high-level system design. The authors in [17] justify the proposed solution by evaluating achievable performance with pointer chasing benchmark, which is similar to pChase that we use in our study for assessing memory processing capabilities. As we have shown in our paper, concurrent independent memory chains generated by pChase could deliver much higher memory bandwidth compared to real graph processing applications that have additional dependencies limiting the available parallelism in the program.

The authors of the paper [7] follow a similar intuition 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.

6. CONCLUSION AND FUTURE WORK

In this paper, we discuss a set of software and hardware challenges accompanying the efficient parallel graph processing, which were highlighted in earlier literature. The core of these issues is the "irregularity" of graph computations which makes the efficient parallelization of graph processing more difficult.

By careful profiling with hardware performance counters available in modern Intel processors, we analyzed how parallel implementation of graph applications use resources of modern multi-core system, in particular a memory system.

We found that Galois graph processing framework executed on latest Intel Ivy Bridge multi-core processor exhibits a good data locality and achieves a good application speedup with an increased number of cores, contrary to traditional past stereotypes, and that a memory bandwidth is not a bottleneck. In our current work, the focus was on the multi-core processor performance and its memory subsystem. In our future work, we plan to analyze a multi-socket configuration and its bottlenecks, as well as the impact of non-unified memory access (NUMA) latencies on performance of large graph processing applications,

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 [15]) for sharing the pChase code and their help with related scripts and questions for measuring the memory system performance.

7. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.
[2] 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.
[3] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.
[4] pChase. https://github.com/maleadt/pChase.
[5] STREAM benchmark. http://www.cs.virginia.edu/stream/.
[6] 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.
[7] 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.
[8] 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.
[9] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proc. of the 19th International Conference on World Wide Web*, 2010.
[10] 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.
[11] A. Lenharth and K. Pingali. Scaling Runtimes for Irregular Algorithms to Large-Scale NUMA Systems . *IEEE Computer Journal*, Aug, 48, 2015.
[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8), Apr. 2012.
[13] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.*, 8(3), 2014.
[14] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1), 2007.
[15] 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.
[16] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna. Extracting the Hyperlink Graphs from the Common Crawl. http://webdatacommons.org/hyperlinkgraph.
[17] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, 2011.
[18] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *Proc.of the 24th Symposium on Operating Systems Principles, (SOSP)*, 2013.
[19] K. Packard. The Machine Architecture. http://keithp.com/blogs/the_machine_architecture.
[20] D. M. Pase and M. A. Eckl. Performance of the IBM System x3755. *Technical Report, IBM, August*, 2006.
[21] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proc. of Intl. Conference on Management of Data, (SIGMOD)*, 2014.
[22] 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.