Energy Characterization of Graph Workloads

Ankur Limaye[a,b,*], Antonino Tumeo[b], Tosiron Adegbija[a]

[a]*Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ - 85721*
[b]*Pacific Northwest National Laboratory, Richland, WA - 99352*

Abstract

Graph algorithms are critical components of the big-data analysis workflow. The graph kernel performance is highly dependent on the input data graphs. The inherently sparse nature of the input graphs often results in irregular memory access patterns, which may not suit the data-locality based cache optimizations featured in current high-performance processors. Much prior research has identified several optimization opportunities by characterizing graph kernels on existing hardware. However, current graph workload characterization studies focus on performance-related observations and optimizations, overlooking the energy implications. In this paper, we address this technology gap by presenting an exhaustive and systematic energy characterization study of graph kernels. We characterize the six GAP benchmark suite kernels with a variety of input graphs on a dual-socket x86-based system. We then analyze how the algorithms, graph characteristics (like graph scale and degree), and system effects (like parallelism, simultaneous multithreading, and multiprocessing) impact the energy. Based on our analysis, we derive observations and insights to develop a basic energy model. The discussions in the paper can enable researchers to advance new models and energy-efficient architectures for graph workloads.

Keywords: Graph applications, workload analysis, energy characterization, big data.

1. Introduction

Graphs are memory-efficient data structures for storing heterogeneous or unstructured data that naturally express relationships among the data elements. Thus, graph algorithms are a vital component of data analytics workflows that span many application domains, including complex network analysis, language understanding, pattern recognition, semantic databases, bioinformatics, and more. Graph algorithms typically employ pointer- or index-based data structures, such as the Compressed Sparse Row (CSR) format, making them prototypical irregular kernels. The traversal of a graph, in fact, usually induces unpredictable data accesses to different segments of the data structures. While there may be some data locality depending on the actual data structure, most of the memory accesses are fine-grained. For example, suppose that an algorithm must load the entire long neighbor lists for some vertices. In such a case, the neighboring vertices may be located at unpredictable non-contiguous memory addresses due to the format of the data structures. The high variability in the size of various memory requests makes latency reduction techniques (and hierarchical memory subsystems) less effective.

For these reasons, there are several ongoing research efforts exploring custom architectures to enhance graph processing. Some solutions include cus-

*Corresponding author
Email addresses: ankurlimaye@email.arizona.edu (Ankur Limaye), antonino.tumeo@pnnl.gov (Antonino Tumeo), tosiron@email.arizona.edu (Tosiron Adegbija)

tom processing elements that decouple computation from communication (e.g., Graphicionado [1], Lincoln Lab graph processor [2]), while other designs explore near memory processing (e.g., Tesseract [3], GraphPIM [4], GraphP [5]). Some systems, like the Cray XMT [6, 7], exploit multithreading to tolerate, rather than reduce, latencies even at a large scale. The EMU system [8] exploits the concept of migrating threads near to the data. The DARPA Hierarchical Identify Verify and Exploit (HIVE) program [9] is looking to build a graph analytics processor that can process (streaming) graphs faster and at much lower power than current processing technology. A particular focus of HIVE is to optimize both performance and power (i.e., the efficiency), trying to reach 1000 times the TEPS/W (traversed edges per second per Watt) of current designs (such as GPUs and conventional CPUs). Ideally, a processor 1000× faster in the same power envelope of a current design, or a processor as fast as a current one, but consuming $1/1000^{\text{th}}$ of the power, would both reach the project objective.

However, to identify the architectural improvements needed to execute these irregular kernels more efficiently, an accurate characterization of their behavior is required. Several works [10, 11, 12, 13, 14, 15, 16, 17] have thus provided detailed characterizations of these workloads on various existing processors, either through profiling or through simulation. The simulation approach may provide more insights about the inner working of the architecture and the opportunity to evaluate behaviors on different configurations; however, it typically only allows the execution and evaluation of small kernels with reduced datasets in a reasonable time. Since graph processing issues become more significant as the datasets grow in size, and graphs used in real-world applications feature data sizes that keep increasing following the big-data trend, simulation-based analyses may provide incomplete insights. On the other hand, graph kernels may be profiled on existing hardware to gather interesting information and drive design considerations.

In this paper, we follow the second approach. Following in the footsteps of previous research works, we perform the workload characterization of a publicly available benchmark suite, the Graph Algorithm Platform (GAP) benchmark suite [18], on a commodity dual-socket x86 system. However, unlike other works, this paper presents an *exhaustive and systematic energy characterization study*. We discuss the energy variations observed due to parallelism, simultaneous multithreading, and multiprocessing by analyzing the Running Average Power Limit (RAPL) performance counters. We also consider the graphs with different graph characteristics (scales and degrees) and investigate these characteristics' sensitivity to the energy. Thus, we derive additional energy insights related to the data graphs' actual structure using our analysis. Our work advances state-of-the-art graph profiling research by providing valuable insights and developing energy models that consider the graph characteristics and structure. Furthermore, our work provides a baseline for future energy-efficient system design for graph processing.

The paper proceeds as follows: Section 2 briefly recaps the related works. Section 3 presents the experimental setup, briefly describing the algorithms, the datasets, the profiled system, and the profiling environment. To tractably represent our analysis results, Section 4 first summarizes the key insights derived from our analysis and then presents the experimental evaluation discussion in greater detail. Section 5 illustrates the use of our data to generate energy models. Finally, Section 6 concludes the paper.

2. Related works

In recent years, there has been an active interest in graph workload characterization studies conducted on different commercial computing systems. These studies typically characterize a subset of popular graph kernels—*betweenness centrality* (*bc*), *breadth-first search* (*bfs*), *biconnected components* (*bicc*), *connected components* (*cc*), *approximate diameter* (*dia*), *graph coloring* (*color*), *k-core* (*kc*), *list ranking* (*list*), *page rank* (*pr*), *single-source shortest path* (*sssp*), and *triangle count* (*tc*)—on a variety of computing systems like *flat shared-memory multiprocessors* (e.g., Cray MTA-2), *symmetric multiprocessors* (e.g., Sun E4500 UltraSPARC II, Sun UltraSPARC T2, IBM

Table 1: Summary of recent studies' experimental setups

Paper	Kernels, Frameworks, and Graphs	Computing platforms
[10]	2 kernels: *cc*, *list* 4 graphs: *random* (4)	Sun E4500 UltraSPARC II Cray MTA-2
[11]	2 kernels: *bicc*, *cc* 5 graphs: *RMAT* (5)	Sun UltraSPARC T2 IBM Power 7
[12]	5 kernels: *bc*, *bfs*, *cc*, *pr*, *sssp* 3 frameworks: Galois, Ligra, GAP 5 graphs: *kron*, *road*, *twitter*, *uniform*, *web*	Intel Xeon E5-2667 v2
[13, 14]	5 kernels: *bc*, *bfs*, *cc*, *dia*, *pr* 1 framework: Galois 3 graphs: *PLD*, *road*, *twitter*	Intel Xeon E5-2660 v2
[15]	3 kernels: *bfs*, *pr*, *sssp* 3 graphs: *kron*, *GTA-T2*, *patents*	–
[16]	6 kernels: *bfs*, *color*, *kc*, *pr*, *sssp*, *tc* 1 framework: IBM System-G 10 graphs: *delaunay*, *kron*, *large*, *road*, *social*	Intel Xeon E5-4655 v4 NVidia Tesla P40 Intel Xeon Phi 7210
[17]	6 kernels: *bc*, *bfs*, *cc*, *pr*, *sssp*, *tc* 1 framework: GAP 3 graph: RMAT (3)	Intel Xeon Platinum 8170 Intel Xeon Phi 7250 Cray XMT (*simulated*)
This work	6 kernels: *bc*, *bfs*, *cc*, *pr*, *sssp*, *tc* 12 graphs: *kron* (9), *road*, *twitter*, *web*	Intel Xeon E5-2687W

Power 7, Intel Xeon), *many-core processors* (e.g., Intel Xeon Phi), and *GPUs* (e.g., NVidia Tesla P40). Table 1 summarizes the graph kernels, the input graphs, and the computing systems used in recent studies. The key observations from these studies are summarized as follows:

1. *Graph algorithms are memory-bound* – Beamer *et al.* [12] and Eyerman *et al.* [17] conclude that the graph kernels are memory-bound since they exhibit high cache miss rates.
2. *Memory accesses show some data locality* – Beamer *et al.* [12], Eisenman *et al.* [13, 14], and Eyerman *et al.* [17] report that few memory accesses in graph kernels show some data locality (e.g., loading neighbor lists). Hence, they can still benefit from caching and prefetching techniques, if adequately applied.
3. *Prefetching data improves performance* – Cong *et al.* [11] show that software prefetching improves graph processing performance. However, software prefetching requires human intervention, compiler modifications, or a change in the programming models. Eisenman *et al.* [13] report that the execution time speedup shows a high correlation with the cache hit rate improvements due to data prefetching.
4. *Memory bandwidth is not the performance bottleneck; but bandwidth utilization is* – Cong *et al.* [11] report that the memory access latency, rather than the processor memory bandwidth, is the performance bottleneck. They also report that the processors did not support enough threads to mask the memory latency entirely. Beamer *et al.* [12] observe that the graph algorithms do not fully utilize the Intel Xeon processor's memory bandwidth. The small instruction window fails to produce enough outstanding memory requests to saturate the bandwidth, and the reorder buffer size limits the achievable memory throughput. Eisenman *et al.* [14] also report that the processors do not fully utilize the available memory bandwidth. Jiang *et al.* [16]

observe that all kernels do not benefit from the high-bandwidth MCDRAM memory, suggesting that bandwidth is not the performance bottleneck. Eyerman *et al.* [17] conclude that although providing high-bandwidth on-chip memory can boost graph processing performance, the memory access latency needs to be kept low. A sufficient number of threads need to be active to generate enough concurrent memory accesses required to saturate the available memory bandwidth.

5. *Multithreading has limited benefits on current architectures* – Beamer *et al.* [12] conclude that increasing the thread count only moderately improves modern processors' performance with deep cache hierarchies and limited memory parallelism. Jiang *et al.* [16] report that the graph applications have a variable optimal thread count. Different input graphs also require a different number of threads to achieve minimum execution times. Eyerman *et al.* [17] find that some graph algorithms do not scale linearly due to load imbalance and a limited number of parallel tasks at high thread counts, depending on the actual parallelism exposed by the algorithm and the specific input graph. They also report that most applications scale well up to the core count, and do not benefit much from simultaneous multithreading.

Most of these studies focus on analyzing and optimizing the graph kernels' performance on their experimental systems; they have two significant drawbacks. First, they do not provide any analysis of the energy or power cost of graph processing. Only Pollard and Norris [15] briefly report the average power consumption during the graph kernel execution. Our study fills this research gap by performing an exhaustive energy characterization study of the kernels.

Second, the previous studies use a limited choice of the kernels and input data graphs. The graph kernels' performance is highly data-dependent; hence, it is necessary to cover a variety of input graphs in the characterization studies. Only Beamer *et al.* [12] and Jiang *et al.* [16] have characterized five or more graph kernels with five or more input graphs in their studies. Moreover, only Jiang *et al.* [16] have considered the same input graphs with different degrees. Our study focuses on observing the impact of different graph characteristics on energy. To this end, we characterize six kernels from the GAP benchmark suite [18], with nine *kron-** synthetic graphs and three real-world graphs—*road*, *twitter*, and *web*—in our experiments. We discuss our experimental setup in detail in the following section.

3. Experimental Setup

This section describes the kernels, input data graphs, the computing system, the profiling environment, and the system configurations used in our experiments. Section 3.1 provides a brief description of the kernels and data graphs used, while Section 3.2 provides an overview of the computing system and data collection methodology used in our study. Section 3.3 describes the four system configurations we explored to identify the system effects on the energy.

3.1. Kernels & Data graphs

We characterize six kernels—*bc*, *bfs*, *cc*, *pr*, *sssp*, and *tc*—from the GAP benchmark suite in our experiments. Beamer *et al.* [18] detail the algorithms and reference code considerations. A brief description of the kernels is as follows:

1. *Betweenness centrality* (*bc*) – The betweenness centrality of a vertex measures its importance in the graph by calculating the fraction of the shortest paths that pass through the vertex. Instead of computing all the possible shortest paths, the *bc* algorithm approximates the betweenness centrality score by computing the shortest paths only from a subset of the vertices using the Brandes [19] algorithm.
2. *Breadth-first search* (*bfs*) – The *bfs* algorithm traverses all reachable vertices from the source vertex, incrementally increasing the depths with every repetition. It traverses to the reachable vertices at the current depth and then moves onto the next depth in the next iteration.

3. *Connected components* (*cc*) – The *cc* algorithm labels all the vertices by their connection. The vertices share the same component label only if an undirected path exists between them. The isolated vertices (vertices with zero degree) are not connected to any other vertices and hence get their own label.
4. *Page rank* (*pr*) – The *pr* algorithm computes the popularity of the vertices in the graph. The vertex's popularity is not only dependent on the number of vertices that point to it, but also factors in the popularity of those vertices. The *pr* algorithm iterates over the entire graph multiple times until the page rank scores of all vertices converge within a specified tolerance threshold.
5. *Single-source shortest path* (*sssp*) – The *sssp* algorithm computes the shortest paths (minimum distances) to all the reachable vertices from the source vertex. The distance between two vertices is the minimum sum of the edge weights along the connected path.
6. *Triangle count* (*tc*) – The *tc* algorithm measures the graph's interconnectedness by counting the total number of triangles (cliques of size 3) found in the graph. The triangles are invariant to permutations. The *tc* algorithm consists of two compute-intensive steps: relabelling the graph by degree and counting triangles by summing the overlaps between each vertex's neighbor list and its neighbor's neighbor lists.

The graph kernels can be classified into two groups: the *single-source* kernels (*bc*, *bfs*, and *sssp*) that require a source vertex to start the execution, and the *whole-graph* kernels (*cc*, *pr*, and *tc*) that process the entire graph in parallel and in the same way for every execution. We chose the default source vertices for the *single-source* kernels to generate deterministic results between the runs (details in Section 3.2). The *bfs*, *cc*, and *sssp* kernels are primarily *traversal-centric* kernels involving minimal computations; however, *bc*, *pr*, and *tc* kernels are highly *compute-centric* in addition to needing graph traversals.

Table 2 summarizes the characteristics of the input data graphs we used in our experiments. The *kron-* graphs are synthetically generated using the Kronecker graph generator [20], while *road* [21], *twitter* [22], and *web* [23] are real-world graphs. The *kron-* data graphs are undirected, whereas *road*, *twitter*, and *web* are directed graphs. All of the graphs except *road* are unweighted. Only the *sssp* kernel executes on weighted graphs, so we used the reference code [20] to generate weights for the unweighted *kron-* graphs. The reference code generates uniformly distributed weights from integers 1 through 255. We classify the synthetic graphs into three categories based on their scales (i.e., nodes), as shown in Table 2: the '*small-scale*' (*kron-g20-**), the '*medium-scale*' (*kron-g22-**), and the '*large-scale*' (*kron-g24-**) graphs. We also categorize the synthetic graphs based on their degrees as the '*small-degree*' (*kron-*-k4*), the '*medium-degree*' (*kron-*-k8*), and the '*large-degree*' (*kron-*-k16*) graphs, for the experimental results discussed in Section 4.

Table 2: Summary of input data graphs' characteristics

Data graph	Nodes	Edges	Degree
kron-g20-k4	1,048,575	4,087,377	3
kron-g20-k8	1,048,575	8,042,557	7
kron-g20-k16	1,048,575	15,699,687	14
kron-g22-k4	4,194,303	16,493,941	3
kron-g22-k8	4,194,303	32,621,964	7
kron-g22-k16	4,194,303	64,155,718	15
kron-g24-k4	16,777,215	66,358,331	3
kron-g24-k8	16,777,216	131,715,222	7
kron-g24-k16	16,777,216	260,376,709	15
road	23,947,348	57,708,624	2
twitter	61,578,415	1,468,364,884	23
web	50,636,151	1,930,292,948	38

3.2. Experimental System

We executed all the kernels on a dual-socket server featuring Intel Xeon E5-2687W processors with the Sandy Bridge architecture. Each socket consists of eight 3.10 GHz hyper-threaded cores, with each core featuring a private 32 KB L1 data (L1d), a private 32 KB L1 instruction (L1i), and a unified 256 KB L2 cache. All the cores within each socket dynamically share a 20 MB last level cache (LLC). The server also features a 64 GB DDR3-1600 RAM.

The server runs Ubuntu 18.04 LTS operating system with Linux 4.15 kernel. All the kernels were compiled using `gcc 7.3`, with the default `-O3 -Wall -std=c++11 -fopenmp` flags. We used the `OMP_NUM_THREADS` OpenMP environment variable to set and sweep through different thread counts, and the OpenMP thread affinity control variable, `OMP_PLACES`, to pin threads to the desired physical and virtual cores in specific processors. We disabled the CPU frequency scaling to eliminate the impact of frequency variation on the execution characteristics and enable a fair comparison across the executions. We used the `perf stat` utility to collect the per-process hardware performance counters and RAPL data. The RAPL interface provides platform software with the ability to monitor, control, and get notifications about the system-on-chip (SoC) power consumption. We used the `energy-pkg` RAPL counter data in our observations, which report the energy consumed (in Joules) by the sockets (including all the cores and uncore components like LLC and memory controller) during the process' execution. We also collected the energy consumed by the cores using the `energy-cores` RAPL counter. To account for variability between the runs, we executed all the kernels with all input graphs ten times and report the average values. The average of standard deviations over these ten runs was 4.29% for all the performance counters.

3.3. System Configurations

We report and analyze the observations for four different system configurations: *single-socket multithreaded* (*SS-MT*), *single-socket simultaneous multithreaded* (*SS-SMT*), *dual-socket multithreaded* (*DS-MT*), and *dual-socket simultaneous multithreaded* (*DS-SMT*) executions.

Multithreading is often used to improve the performance of sequential code. Increasing the number of threads, ideally, reduces the execution time. We capture these effects of scalability and parallelism exhibited by the kernels for various input graphs in our *SS-MT* configuration. We pinned each thread to a specific core (as discussed in Section 3.2); hence, the number of cores in a processor limits the thread count. The Intel Xeon E5-2687W processor consists of 8 physical cores allowing for a maximum of 8 threads in *SS-MT* configuration.

There are two options available to further increase the number of threads: using simultaneous multithreading on the same processor or using an additional processor. We capture these two scenarios and their trade-offs in the *SS-SMT* and *DS-MT* configurations, respectively. Simultaneous multithreading improves performance by increasing CPU utilization without any extra hardware, while spreading the execution across two processors increases memory parallelism (due to the two sockets) at the cost of additional hardware and communication. The Intel Hyper-Threading technology allows for one additional thread per core during simultaneous multithreading. In the *DS-MT* configuration, we used multithreading on two processors. Thus, for the two configurations (*SS-SMT* and *DS-MT*), a maximum of 16 threads are supported.

The *DS-SMT* configuration employs simultaneous multithreading on two processors, and potentially shows the benefits due to both features. The *DS-SMT* configuration has the maximum number of possible threads (32) among all the configurations.

4. Experimental Results

We discuss our experimental observations in this section. To make the information presented herein more tractable for the reader, Section 4.1 starts by summarizing the key insights derived from our analysis. Thereafter, Sections 4.2, 4.3, and 4.4 examine the impacts of various graph kernels, input graphs, and system configurations, respectively, in greater detail.

4.1. Key insights

The key observations of our study are summarized as follows:

1. The performance improvements and energy savings due to increasing thread counts are highly correlated. However, the rate of execution time and energy savings improvements is not linear (and shows diminishing returns) with an increasing number of threads. The graph kernels do not

Table 3: Execution Time and Energy values for single-threaded execution of all kernels with all input graphs

Graph	Edges	Execution Time (s)						Energy (J)					
	(millions)	*bc*	*bfs*	*cc*	*pr*	*sssp*	*tc*	*bc*	*bfs*	*cc*	*pr*	*sssp*	*tc*
kron-g20-k4	4.09	0.34	0.10	0.12	0.48	0.52	3.26	22.73	5.89	8.08	39.39	31.03	274.13
kron-g20-k8	8.04	0.48	0.08	0.12	0.62	0.74	10.35	38.01	6.70	7.54	53.52	42.82	772.30
kron-g20-k16	15.70	0.82	0.13	0.17	1.09	1.38	14.73	58.34	10.26	11.11	75.35	80.84	1062.78
kron-g22-k4	16.49	1.44	0.26	0.38	2.56	1.97	19.64	116.96	18.80	28.58	208.89	138.13	1372.02
kron-g22-k8	32.62	2.64	0.34	0.49	4.08	3.44	66.03	207.33	29.03	35.82	338.70	245.28	4441.05
kron-g22-k16	64.16	4.59	0.48	0.70	6.33	6.24	90.03	304.54	41.60	59.76	511.96	413.61	6003.41
kron-g24-k4	66.36	7.00	1.00	1.58	14.01	8.22	119.55	519.80	64.84	109.90	1067.91	546.25	8003.55
kron-g24-k8	131.72	13.10	1.35	2.08	22.55	14.88	422.22	947.27	88.86	172.38	1700.02	932.30	28034.60
kron-g24-k16	260.38	19.82	1.86	2.74	36.48	27.64	543.22	1506.80	128.66	192.22	2668.67	1600.50	36013.42
road	57.71	4.79	2.08	1.62	7.19	144.93	–	425.03	168.84	136.48	562.31	12292.35	–
twitter	1468.36	85.01	12.18	14.95	344.22	133.19	–	5674.89	888.41	1038.81	24112.63	7610.49	–
web	1930.29	47.04	17.33	14.95	68.17	169.04	–	2967.42	1224.74	1051.98	5063.77	9397.29	–

scale linearly due to load imbalance, limited parallelism exposed by the kernels and input graphs, or underlying hardware bottlenecks, as observed in [17].

2. The *whole-graph* (*pr* and *tc*) kernels and *bc* show better energy savings with increased thread counts than the *single-source traversal-centric* kernels for all the configurations. The *whole-graph* kernels are designed to exhibit a large amount of fine-grained parallelism (since they operate on entire input graphs in parallel, without a specific starting node), and their high concurrency positively impacts energy-saving. The *bc* kernel operates only on a smaller sub-graph, showing energy savings characteristics similar to the *whole-graph* kernels. The *traversal-centric* kernels also do not have significant computations to mask the latency, hence, they do not scale well. These observations highlight how algorithmic research can impact energy efficiency, as some graph algorithms are naturally sequential and need approximation or relaxation to operate in parallel.
3. The performance and energy of the graph kernels depend significantly on the graph characteristics, and are more sensitive to the number of edges in the input graphs than the graph scale. Smaller graphs show better energy savings than larger graphs, as the smaller graphs result in fewer costly off-chip memory accesses, improving both the performance and energy savings across the system configurations. Moreover, the smaller degree graphs scale better than the larger degree graphs since they are less connected.
4. *Simultaneous multithreading* is energy-efficient only for fewer core counts. When the maximum (8) cores are used, *simultaneous multithreading*, in fact, consumes more energy than *multithreaded* execution, in both the single- and dual-socket configurations for most cases. As highlighted by previous studies [17], *simultaneous multithreading* does not significantly increase performance improvement, and does not offset the power consumption increase due to increased CPU utilization.
5. *Dual-socket simultaneous multithreading* improves the performance compared to *single-socket multithreading* due to higher thread counts, separate memories per processor, and data partitioning. However, the energy savings are negligible due to the power cost of additional hardware and memory transactions between the two processors.

The impacts of various input kernels, graphs, and system configurations are discussed in detail in the following subsections.

4.2. Impact of Graph Kernels

Table 3 reports the execution time and energy values for the kernels' single-threaded execution with

different input graphs. The *tc* kernel implementation in the GAP benchmark suite only works with undirected graphs, so the analysis for *tc* does not include results for the directed input graphs (*road*, *twitter*, and *web*).

The graph algorithms highly influence performance and energy values. The primarily *traversal-centric* kernels: *bfs* and *cc*, show the lowest execution times and energy values. The *bfs* and *cc* kernels execute only an average of 7.49B and 5.37B instructions, respectively, with an average 3.07B and 2.66B L1d cache accesses, considering single-threaded execution on all input graphs. The *tc* kernel exhibits the highest execution times and energy values. Even without executing the largest *real-world* graphs, the *tc* kernel showed the highest average instructions count (767.49B) and L1d cache accesses (88.80B), since it loads multiple neighbors and neighbor's neighbor lists for triangle counting. The *pr* kernel traverses the input graph multiple times for converging on the page rank score for each vertex, and hence shows a high average instruction count (49.42B) and L1d cache accesses (19.95B). Despite being a *single-source* kernel, the *bc* kernel operates on a smaller sub-graph and calculates multiple shortest paths to approximate the betweenness centrality score. Hence, it shows characteristics similar to the *whole-graph* kernels.

The *traversal-centric* kernels are highly *memory-bound*, showing high average L1d cache accesses per thousand instructions: *sssp* (518.16), *bfs* (497.31), and *cc* (466.79), compared to the *compute-centric* kernels *tc* (117.53), *bc* (359.40), and *pr* (397.59). The average energy consumed by cores (`energy-cores`) compared to the total energy (`energy-pkg`) for the *compute-centric* kernels: *tc* (77.99%), *pr* (77.06%), and *bc* (71.13%) is also higher compared to the *traversal-centric sssp* (58.52%), *bfs* (67.87%), and *cc* (69.06%) kernels across all the configurations.

We use the single-threaded execution values (in Table 3) to normalize all the system configurations' execution time and energy values for the analysis presented in Section 4.4.

4.3. Impact of Graph Characteristics

Table 3 also shows the effects of graph characteristics on the performance and energy for different kernels. The synthetic graphs are grouped together based on the scale (i.e., number of nodes) similar to Table 2. Within the same graph scale, the execution time and energy values scale by an average of 1.67× and 1.65×, respectively, for a 2× increase in the number of edges. Smaller graphs show better energy savings than larger graphs, as the smaller graphs result in fewer costly off-chip memory accesses. For example, the average DRAM accesses for the *small-scale kron-g20-** graphs are 7.89M compared to 575.63M observed for the *large-scale kron-g24-** graphs across all the kernels. When the number of edges are approximately equal with the scale increasing by 4× (e.g., *kron-g20-k16* to *kron-g22-k4*, and *kron-g22-k16* to *kron-g24-k4*), the execution time and energy values scale only by an average 1.83× and 1.84×, respectively. This observation suggests that the graph kernels are more sensitive to an increase in the number of edges than the number of nodes in the input graphs, and it makes sense since a higher number of edges per node (i.e., degree) makes the graph more connected and impacts the algorithms and graph traversals.

4.4. Impact of System Configurations

Tables 4, 5, and 6 summarize the results for the four system configurations that we have considered in our experiments. Tables 4 and 5 report the average normalized execution time and energy values, respectively, of the graph kernels for all input graphs. Table 6 reports the average normalized energy values for different graph categories. Since the corresponding single-threaded execution data normalize the values, lower values suggest better performance and energy savings. In Tables 4, 5, and 6, the columns denoting equal thread counts across four configurations are highlighted with the same gray shade to distinguish them visually. We discuss the results and compare the configurations in the following subsections.

4.4.1. Power cost for system hardware

Table 4 shows that as system resources increases (i.e., we increase the number of cores, activate simultaneous multithreading, and/or increase the number of processors), the performance improves (excution time reduces). However, the power consumption

Table 4: Average execution time values (normalized to single-threaded execution) for all input graphs in four system configurations. The same gray shade signifies an equal number of threads.

	SS-MT			*SS-SMT*			*DS-MT*			*DS-SMT*		
Cores per socket	1	4	8	1	4	8	1	4	8	1	4	8
Threads	1	4	8	2	8	16	2	8	16	4	16	32
bc	1.00	0.82	0.46	0.75	0.77	0.57	0.61	0.24	0.18	0.37	0.18	0.18
bfs	1.00	0.67	0.59	0.78	0.64	0.61	0.81	0.62	0.63	0.69	0.61	0.67
cc	1.00	0.58	0.48	0.73	0.54	0.48	0.81	0.52	0.50	0.64	0.49	0.49
pr	1.00	0.33	0.20	0.57	0.22	0.20	0.63	0.22	0.16	0.36	0.16	0.15
sssp*	1.00	0.79	0.72	0.84	0.76	0.72	0.85	0.70	0.67	0.78	0.66	0.67
sssp-road	1.00	1.07	1.27	1.02	1.26	2.36	1.28	1.66	2.59	1.38	2.57	9.71
tc	1.00	0.28	0.14	0.52	0.14	0.10	0.51	0.15	0.08	0.27	0.08	0.06
Average	1.00	0.59	0.44	0.70	0.53	0.46	0.71	0.42	0.38	0.53	0.37	0.38

Table 5: Average energy values (normalized to single-threaded execution) for all input graphs in four system configurations. The same gray shade signifies an equal number of threads.

	SS-MT			*SS-SMT*			*DS-MT*			*DS-SMT*		
Cores per socket	1	4	8	1	4	8	1	4	8	1	4	8
Threads	1	4	8	2	8	16	2	8	16	4	16	32
bc	1.00	0.79	0.45	0.58	0.73	0.53	0.80	0.38	0.31	0.52	0.31	0.36
bfs	1.00	0.67	0.64	0.72	0.71	0.67	0.85	0.68	0.72	0.70	0.69	0.95
cc	1.00	0.62	0.56	0.68	0.63	0.56	0.88	0.61	0.60	0.69	0.59	0.65
pr	1.00	0.42	0.32	0.56	0.33	0.33	0.84	0.38	0.32	0.55	0.33	0.34
sssp*	1.00	0.77	0.78	0.76	0.78	0.77	0.80	0.77	0.76	0.77	0.72	0.75
sssp-road	1.00	1.32	1.72	1.16	1.77	3.65	1.49	2.69	5.37	1.83	5.51	19.35
tc	1.00	0.37	0.26	0.54	0.26	0.21	0.73	0.30	0.22	0.44	0.22	0.18
Average	1.00	0.61	0.51	0.64	0.58	0.52	0.82	0.53	0.50	0.62	0.49	0.55

Table 6: Average energy values (normalized to single-threaded execution) for different input graph characteristics in four system configurations. The same gray shade signifies an equal number of threads.

	SS-MT			*SS-SMT*			*DS-MT*			*DS-SMT*		
Cores per socket	1	4	8	1	4	8	1	4	8	1	4	8
Threads	1	4	8	2	8	16	2	8	16	4	16	32
Small-degree Synthetic	1.00	0.57	0.46	0.61	0.57	0.46	0.77	0.47	0.44	0.56	0.41	0.53
Medium-degree Synthetic	1.00	0.58	0.50	0.63	0.54	0.52	0.81	0.52	0.50	0.60	0.48	0.56
Large-degree Synthetic	1.00	0.62	0.52	0.66	0.58	0.53	0.82	0.54	0.50	0.63	0.50	0.50
Real-world	1.00	0.71	0.57	0.69	0.67	0.59	0.89	0.59	0.56	0.69	0.56	0.62
Small-scale Synthetic	1.00	0.62	0.55	0.67	0.64	0.58	0.77	0.56	0.55	0.63	0.53	0.72
Medium-scale Synthetic	1.00	0.58	0.47	0.59	0.53	0.49	0.78	0.49	0.45	0.56	0.45	0.50
Large-scale Synthetic	1.00	0.57	0.46	0.64	0.51	0.44	0.85	0.48	0.44	0.60	0.48	0.38

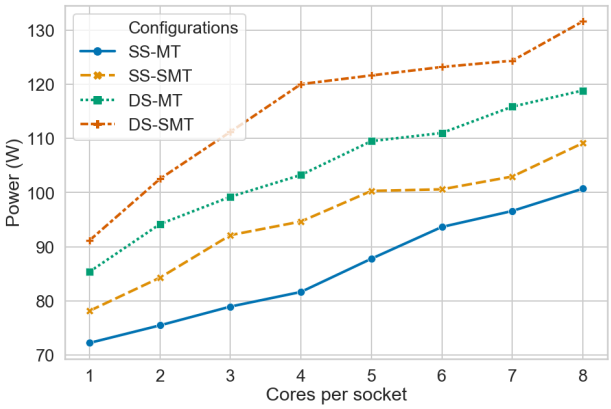

Figure 1: Average power consumption for all kernels and input graphs in four system configurations.

should also increase due to the increased CPU utilization or the extra hardware. Figure 1 depicts the impact of these system changes on the power cost, averaged for all input kernels executing all the input graphs. The power consumption increased fairly linearly as the number of cores increased due to multithreading, at an average rate of 4.76 W per core for all configurations. *Simultaneous multithreading* improves the performance by increasing CPU utilization; however, it costs an additional 10.23 W on an average for different core counts. Using an additional processor (i.e., *dual-socket execution*) increases the power by 19.60 W on average.

Despite the increase in power with the increase in resources, we also found that the performance improved enough to increase the energy savings. Table 5 depicts the energy values normalized to single-threaded execution in the different system configurations. Given that energy is the product of power and execution time, the linear increase in power consumption with increasing core counts results in the energy values scaling at a slower rate than the execution time (in Table 4). The energy savings are smaller than performance improvements for all cases.

4.4.2. Multithreading on a single socket (SS-MT)

Increasing the number of cores improves the performance and energy savings of the kernels. However, the energy does not decrease linearly, and saturates to a lower bound, as seen from Table 5. We observed the highest energy savings improvement when the thread count increases from 1 to 2, showing an average 22.09% improvement in energy savings. Across all the GAP benchmark kernels, we observed average energy savings of 49.14% for eight threads.

The *whole-graph* kernels—*tc* (73.72%) and *pr* (68.04%)—show the highest average energy savings for eight threads, while the *sssp* (21.94%) kernel shows the least. The *whole-graph* kernels are designed to exhibit more parallelism (since they operate on the entire input graph in parallel, without a specific starting node or vertex), which positively impacts the energy-saving. The *whole-graph* (*pr* and *tc*) and *bc* kernels scale better with increased thread counts than the *single-source* kernels for all the configurations. The *bc* kernel operates on a smaller sub-graph, showing better energy savings than other *single-source* kernels. The *whole-graph* kernels show significantly better average energy savings (60.84%) than for *single-source* kernels (38.11%) for eight threads. Thus, the highly scalable kernels also show the most energy savings.

The *sssp* kernel scales the worst among all the kernels, inherently due to its algorithm. The *sssp* kernel finds the shortest paths (i.e., the path with the minimum sum of weights) for all the nodes in the graph from the specific starting vertex. Parallelizing the *sssp* kernel requires additional synchronization to maintain and update the correct weights across the threads spawned. Moreover, when the *sssp* kernel operates on the *road* input graph, multithreading degrades the performance and results in more energy consumption with an increase in the number of threads, as seen in Tables 4 and 5. The *road* graph is big (nodes = 23.95M, degree = 2) and has a high diameter, causing the synchronous implementations to have longer runtimes. The results for *sssp–road* are significant enough to skew the average result values, and using them for analysis could give inaccurate conclusions. Hence, when reporting the average results for the *sssp* kernel, we did not consider the *sssp–road* observations.

The energy savings for *small-degree* synthetic graphs scale better than higher degree graphs, as observed in Table 6. Across all the kernels, the

small-degree input graphs show the best average energy savings (54.08%), higher than energy savings for the *medium-degree* (49.58%), and *large-scale* (48.16%) synthetic graphs at the maximum eight-core usage. The graph kernels are more sensitive to an increase in the number of edges per node. In the graphs categorized based on degree, all the three categories: *small-degree*, *medium-degree*, and *large-degree*, have the same average number of nodes, but the degrees increase from 4, 8, to 15, respectively. Higher edge degrees means the graph is more connected and has more number of edges. The *real-world* graphs are big as well as have higher degrees compared to the synthetic graphs. Hence, they show the worst scaling with 43.49% average energy savings for eight cores. Interestingly, when the synthetic graphs are compared based on the scale, the *large-scale* synthetic graphs scale much better than the smaller synthetic graphs as well as *real-world* graphs. The *smaller-scale* synthetic graphs do not provide enough parallelism to scale, while the much larger *real-world* graphs have more off-chip memory requests that degrade the performance and energy savings. This observation suggests that increasing the number of threads is beneficial for graphs up to a certain scale, depending on the underlying system architecture. Thus, unsurprisingly, the *small-degree large-scale* synthetic graphs *kron-g24-k4* and *kron-g22-g4* show the best average energy savings of 56.87% and 56.78%, respectively, while the *real-world* graphs *web* and *twitter* show the worst energy savings of 31.52% and 44.71%, respectively, for eight cores.

4.4.3. Simultaneous multithreading on a single socket (SS-SMT)

Simultaneous multithreading is only effective for smaller core counts. Compared to *SS-MT* execution, the energy savings gradually reduced with increasing core counts for most of the graphs, as seen in Table 5. Across all the kernels, *SS-SMT* shows the best average energy savings for single-core execution (35.61%). In contrast, the average energy savings is 1.32% less than *SS-MT* when the processor is fully used (i.e., eight cores), despite having 2× threads. These observations suggest that as the number of cores increases, the performance improvement cannot offset the increased power cost due to additional system usage, making *simultaneous multithreading* energy-inefficient for a higher number of cores. Previous studies [12, 17] have already shown that increasing the thread count and using simultaneous multithreading have limited benefits in modern processors due to load imbalance, limited parallelism exposed by the algorithms, and hardware bottlenecks like deep cache hierarchies and limited memory parallelism. The linearly increasing power consumption due to increased CPU utilization without sufficient performance improvements negatively impacts the energy in the configurations employing *simultaneous multithreading* (like *SS-SMT* and *DS-SMT*).

The *tc* and *pr* kernels show better than average energy savings of 79.18% and 66.69%, respectively, for eight cores. As seen from Table 5, the *traversal-centric* (*bfs* and *cc*) and *sssp* kernels show a relatively flat trend, suggesting that neither *simultaneous multithreading* nor an increase in core counts can improve the energy savings. Only the *tc* kernel shows better energy savings of 5.46% in the *SS-SMT* configuration than *SS-MT* for eight cores.

The average energy savings across all the kernels decrease from 54.08% for the *small-degree* synthetic graphs to 40.84% for the highest degree *real-world* graphs. The trends observed for *SS-MT* hold true for the *SS-SMT* configuration as well. However, the *SS-SMT* configuration has a slightly negative impact on energy savings compared to *SS-MT* for eight cores. The *large-scale* synthetic graphs *kron-g24-k4* and *kron-g24-k8* graphs show the best average energy savings of 60.62% and 58.65%, respectively, while the *web* graph shows the worst average energy savings (28.50%).

4.4.4. Multithreading on two sockets (DS-MT)

Dual-socket execution improves the performance of the graph kernels. The energy savings, however, were negligible. For eight cores per socket, the *DS-MT* shows performance improvements of 6.40% (with respect to *SS-MT*) and 7.99% (with respect to *SS-SMT*), as seen in Table 4, but only 1.19% and 2.51% more energy savings than *SS-MT* and *SS-SMT*, respectively, from Table 5. When the kernels execute

over two sockets, the input graphs also get partitioned across the NUMA domains—this data partitioning results in more off-core memory requests, which improves the memory parallelism offered by the system. As such, the additional power cost of using more processors results in the observed energy inefficiency. The average remote DRAM accesses increased by 705.86× when two sockets were used compared to single-socket usage across all the kernels and input graphs. However, for the largest *real-world* *twitter* and *web* graphs, the average remote DRAM accesses increase was only 1.40×, since the relevant accessed data occupies large memory compared to the synthetic *kron-* graphs. The average improvement in energy savings decreases with increasing core counts for most kernels. Across all the kernels, the highest gains in energy savings compared to the baseline *SS-MT* configuration (by 13.33% – 17.98%) are observed for up to three cores-per-socket usage, while the worst gains are observed for per-socket core counts higher than five (by 0.39% – 1.19%).

We can make a fair comparison of the *SS-SMT* and *DS-MT* configurations since these two configurations support the same maximum of 16 threads. The *DS-MT* configuration involves using an additional processor, while the *SS-SMT* simply increases CPU utilization without additional resources. The comparison between these two configurations describes the trade-off of increasing power costs due to CPU usage and using additional resources. The high power cost of adding a processor, while showing similar performance improvements for fewer cores-per-socket makes *DS-MT* energy inefficient compared to the *SS-SMT* configuration. However, when the number of cores used per-socket increases, the *DS-MT* shows comparable energy savings to *SS-SMT* due to better performance improvements and increased power cost for simultaneous multithreading. The *SS-SMT* configuration exhibits better average energy savings than the *DS-MT* configuration for one core per-socket (by 17.63%). For eight cores per-socket usage, the *DS-MT* shows slightly better average energy savings (2.51%).

As observed from Table 5, the *tc* kernel scales the best and exhibits the best average energy savings of 78.14% among all the kernels for eight cores per socket. In comparison, *sssp* and *bfs* show the worst average energy savings of 23.67% and 28.48%, respectively. The *bc* kernel benefits the most from *DS-MT* compared to *SS-SMT*, showing an increased performance improvement by 38.30% and energy savings by 22.00%. The improvement observed for the *whole-graph* kernels (*tc*, *pr*) and *bc* are the result of the higher memory parallelism offered by the use of two sockets. The *traversal-centric* kernels (*bfs* and *cc*) in fact, show reduced energy savings by 4.30% and 4.69%, respectively, in *DS-MT* configuration with respect to *SS-SMT* for eight cores-per-socket usage.

The *small-degree* synthetic graphs show the highest average energy savings of 56.40%, compared to the average energy savings of 49.72% and 50.06%, for the *medium-degree* and *large-degree* synthetic graphs for eight cores-per-socket. Across all the kernels, the *kron-g24-k4* and *kron-g22-k4* graphs show the best average energy savings (61.38% and 59.08%, respectively), while the *real-world* graphs—*web*, *road*, and *twitter*—show lower average energy savings of 39.76%, 43.71%, and 47.50%, respectively. The *DS-MT* configuration mostly benefits the largest *web* graph, showing 8.24% and 11.26% additional energy savings than in *SS-MT* and *SS-SMT* configurations, respectively. Unsurprisingly, due to the high diameter and synchronization limitations, when additional processors are used in the *DS-MT* configuration, the *road* graph exhibits reduced average energy savings compared to *SS-MT* (by 7.45%) and *SS-SMT* (by 13.22%) configurations.

4.4.5. Simultaneous multithreading on two sockets (DS-SMT)

The *DS-SMT* configuration is possibly the best performing configuration; however, it has the worst energy-efficiency. The average energy savings across all kernels (for all the input graphs) for eight cores-per-socket usage for *DS-SMT* configuration is 44.86%, which is worse than the average energy savings observed for *SS-MT*, *SS-SMT*, and *DS-MT* by 4.28%, 2.96%, and 5.47%, respectively. *DS-SMT* performed worse than all the other configurations despite showing the best performance improvement. These results confirm that the performance improvements observed due to increased thread counts and mem-

ory parallelism do not offset the increased power consumption due to the additional processor and the increased CPU utilization.

The *tc*, *pr*, and *bc* kernels show the most average energy savings (81.63%, 66.19%, and 63.64%, respectively) for eight cores-per-socket, while the *bfs* kernel shows the least average energy savings of 4.81%. Compared to *DS-MT*, the use of simultaneous multithreading, reduces the average energy savings for the *bc*, *bfs*, *cc*, and *pr* kernels for the maximum core count, as seen in Table 5.

The *large-scale* synthetic graphs show the best average energy savings (62.35%) across all the kernels, which is 34.45% higher than the average energy values shown by the *small-scale* graphs for the maximum eight cores-per-socket. The *large-scale* synthetic graphs: *kron-g24-k4* (66.82%), *kron-g24-k8* (64.41%), and *kron-g24-k16* (55.81%) show the highest average energy savings, while the *road* and *web* input graph shows the lowest average energy savings of 18.07% and 42.54%, respectively.

5. Modeling Energy Characteristics

The data-dependent behavior and irregular memory accesses make it challenging to model the performance of graph kernels. Previous studies [24, 25, 26] have tried to model the system performance of a few graph kernels typically by performing static and dynamic code analysis, modeling for the amount of the expected memory communication volume, and the performance modeling of memory hierarchy components (like caches) based on miss rates or memory bandwidth utilization. Our extensive experiments have provided us with a lot of data points for six graph kernels' energy values, executing over 12 input graphs in four configurations for eight per-socket core counts. To illustrate how researchers might use the data provided herein[1], we experimented with fitting two different models for the data, detailed in Section 5.1. We evaluate the two models and present their comparison and analysis in Section 5.2. Our models try to fit the graph kernel's energy values for the following features: input graph characteristics (*number of nodes*, *number of edges*, *average degree*), and system characteristics (*number of cores*, *number of processors*, *number of threads per core*).

5.1. Model Selection

We experimented with two fundamentally diverse regression models: a simple *Multiple Linear Regression* [27] and an *ensemble-based Random Forest Regressor* [28]. Note that several other models can be explored, but these two models were selected to illustrate how the characterization and data presented herein can be used to model the energy characteristics of graph workloads and understand what type of model would be more useful for fitting the data. We used Python's `scikit-learn` library [29] to explore the two models. The presented models are data-driven, and hence are dependent on the evaluated system. However, our methodology is reproducible and can be used to derive energy models for a different system. A brief description of the two models is as follows:

- *Multiple Linear Regression* (*MLR*): The MLR model is one of the simplest models, and tries to fit a linear approximation model to predict the target values y based on independent input features (x_1, x_2, ..., x_n), according to:

  $$y = a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n$$

  In our model, the target y is the energy values, and the input variables are the various graph and system characteristics. The MLR model finds the coefficients (a_0, a_1, a_2, ..., a_n) of each input feature such that it minimizes the mean squared error between the predicted and actual target values. We used the `scikit-learn` library's builtin `sklearn.linear_model.LinearRegression` class to build our MLR model.

- *Random Forest Regressor* (*RFR*): The RFR model is an ensemble model that uses multiple decision trees and bagging technique to predict

[1]The data can be downloaded from `https://github.com/ankurlimaye/energy-characterization-of-graph-workloads-data`

the target values. *Random forest* builds multiple decision trees from randomly sampled input training data and merges their predictions together by averaging the trees' predictions. The RFR model reduces overfitting the data due to the bagging technique of combining the predictions of different trees generated for random data. The advantage of using an RFR model is that it is nonparametric, and thus does not depend on any specific data distribution and can contain collinear variables. We used the `scikit-learn` library's builtin `sklearn.ensemble.RandomForestRegressor` class to build our RFR model.

5.2. Model Validation

We performed 5-fold cross-validation for both models to reduce bias in input data. In the 5-fold validation, we first randomly split the input data into five sets. Both the models are then trained five times, each time with a different set used as the testing data and the remaining four sets as the training data. The 5-fold cross-validation ensures that each input data entry is a part of test data at least once. After training the models for five times, we have reported the average scores to determine the fit.

To evaluate the two models, we used two metrics: *R-Squared (R^2)* and *accuracy*. We briefly describe both metrics as follows:

- *R-squared (R^2)*: R-squared (R^2) is the default goodness-of-fit measure for regression models in the `scikit-learn` library. This statistic indicates the variance ratio explained by the model compared to the total variance in input data. It measures the strength of the relationship between the model and the test outputs. The best possible R^2 value is 1, and lower values suggest that the model does not fit the input data well.
- *Accuracy*: We define accuracy as how close the predicted values are to the original test values. It is calculated as the ratio of error (the difference between the input test values and predicted values) to the input test values. An accuracy of 100% denotes that the predicted value matches the original value.

Table 7: Performance of the energy models

Kernel	*MLR*		*RFR*	
	Accuracy	R^2	Accuracy	R^2
bc	-304.87%	0.748	77.37%	0.894
bfs	30.12%	0.982	93.01%	0.994
cc	16.20%	0.984	93.67%	0.997
pr	-1099.05%	0.643	88.53%	0.926
sssp	-15.49%	0.977	88.41%	0.988
tc	-254.50%	0.749	89.30%	0.961

Table 7 presents the accuracy and R^2 values of the two models explored for the different kernels. As observed in Table 7, the MLR model performs poorly compared to the RFR model for both metrics. The MLR model performed very poorly with R^2 values as low as 0.643. The R^2 values for *bfs*, *cc*, and *sssp* are high (0.977 – 0.984), suggesting that the model is a good fit and the input features highly explain the trends in predicted values. However, the accuracy of these kernels are relatively low (-15.49% – 30.12%). This observation suggests that even if the MLR model can predict the correct trend, it fails to predict the actual values. The *bc*, *pr*, *sssp*, and *tc* kernels showed negative accuracy suggesting that the predicted values' errors were greater than the actual test values. On the other hand, the RFR models achieve high R^2 values (0.894 – 0.997) with high accuracy ranging from 77.37% – 93.67% for the different kernels. Thus, the RFR model predicts the correct trend, and the predicted values are much closer to the actual values.

We investigated the RFR model further to understand why it performed better by identifying the most critical features that affected the predictions. We removed one input feature at a time, retrained the RFR models, and measured the drop in R^2 values compared to original RFR models. The input feature that results in the highest drop in the R^2 values is the critical feature. Interestingly, the critical input feature across all the cases was the number of per-socket cores used. This fact suggests that the model needs to correctly account for the non-linear scaling in energy values with increasing threads to improve the model's performance. The MLR model fails to

account for the non-linearity. Hence, the highly scalable *bc*, *pr*, and *tc* kernels show worse performance metrics for the MLR model in Table 7 than the poorly scalable *bfs*, *cc*, and *sssp* kernels.

Our modeling experiment's key takeaway is that the choice of model selection highly depends on the graph kernels' execution behaviors. Even though the RFR models showed high R^2 values, the prediction accuracy can still be improved. The models and predictions can be further improved by considering more input features, hyperparameters tuning, training with different input graphs, or even choosing different models (e.g., polynomial regression to factor non-linearity).

6. Conclusion

In this study, we performed the energy characterization of graph kernels from the GAP benchmark suite. We performed the experiments in four system configurations: *single-socket multithreaded*, *single-socket simultaneous multithreaded*, *dual-socket multithreaded*, and *dual-socket simultaneous multithreaded* executions. We analyzed the effects of graph shapes on the energy characteristics for different kernels.

The *whole-graph* GAP benchmark suite kernels showed better average energy savings than single-source kernels in all the system configurations. The *whole-graph* kernels are designed to exhibit fine-grained parallelism and high scalability. This impacts the energy-savings positively.

The smallest size (scale and degree) input graphs show the most energy savings. The smallest graphs result in low off-chip memory accesses, improving both the performance and energy savings across the system configurations. The real-world graphs, which have large scale and degree, show poor energy savings, and the *road* graph even shows increased energy consumption for higher core counts. The *small-degree* graphs scale better than the larger degree graph since they are less connected.

In the *single-socket multithreaded* execution, increasing the thread counts resulted in lower energy consumption for almost all the kernels; however, this decrease is not linear and saturates to a lower bound. Thus, we observed limited energy savings due to multithreading. *Simultaneous multithreading* on a single socket is only beneficial and shows high energy savings for lower core counts. However, when the kernel execution is spread across two sockets, the average energy savings reduces. Our study suggests that even though *simultaneous multithreading* does not significantly improve performance and energy savings, there are still significant opportunities to optimize the concurrency and single-core utilization for graph applications, since the communication across two sockets becomes more relevant from the energy efficiency perspective.

Our study advances the state-of-the-art in graph profiling research by providing valuable insights. Using our experimental data, we developed models that consider the graph and system characteristics as input features to predict the energy values for different graph kernels. This work provides a baseline for optimizing the energy, and the analysis discussed herein can also provide insights for designing next-generation energy-efficient systems for graph processing and improving graph models.

References

[1] T. Ham, L. Wu, N. Sundaram, N. Satish, M. Martonosi, Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics, in: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '16, 2016, pp. 1–13. doi:10.1109/MICRO.2016.7783759.

[2] W. Song, V. Gleyzer, A. Lomakin, J. Kepner, Novel Graph Processor Architecture, Prototype System, and Results, in: Proceedings of the IEEE High Performance Extreme Computing Conference, HPEC '16, 2016, pp. 1–7. doi:10.1109/HPEC.2016.7761635.

[3] J. Ahn, S. Hong, S. Yoo, O. Mutlu, K. Choi, A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing, SIGARCH Comput. Archit. News 43 (3) (2015) 105–117. doi:10.1145/2872887.2750386.

[4] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, H. Kim, GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks, in: Proceedings of the IEEE International Symposium on High Performance Computer Architecture, HPCA '17, 2017, pp. 457–468. doi:10.1109/HPCA.2017.54.

[5] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, X. Qian, GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition, in: Proceedings of the IEEE International Symposium on High Performance Computer Architecture, HPCA '18, 2018, pp. 544–557. doi:10.1109/HPCA.2018.00053.

[6] J. Feo, D. Harper, S. Kahan, P. Konecny, ELDORADO, in: Proceedings of the 2nd Conference on Computing Frontiers, CF '05, 2005, pp. 28–34. doi:10.1145/1062261.1062268.

[7] L. Kaplan, Cray XMT, in: D. Padua (Ed.), Encyclopedia of Parallel Computing, Boston, MA, USA, 2011, pp. 453–457. doi:10.1007/978-0-387-09766-4_307.

[8] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, S. Stein, Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture, in: Proceedings of the 6th Workshop on Irregular Applications: Architectures and Algorithms, IA^3 '16, 2016, pp. 2–9. doi:10.1109/IA3.2016.7.

[9] W. Shen, Hierarchical Identify Verify Exploit (HIVE) (2016).
URL https://www.darpa.mil/program/hierarchical-identify-verify-exploit

[10] D. Bader, G. Cong, J. Feo, On the architectural requirements for efficient execution of graph algorithms, in: Proceedings of the IEEE International Conference on Parallel Processing, ICPP '05, 2005, pp. 547–556. doi:10.1109/ICPP.2005.55.

[11] G. Cong, K. Makarychev, Optimizing Large-scale Graph Analysis on Multithreaded, Multicore Platforms, in: Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS '12, 2012, pp. 414–425. doi:10.1109/IPDPS.2012.46.

[12] S. Beamer, K. Asanovic, D. Patterson, Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server, in: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC '15, 2015, pp. 56–65. doi:10.1109/IISWC.2015.12.

[13] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, S. Katti, Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges, in: Proceedings of the 24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '16, 2016, pp. 49–58. doi:10.1109/MASCOTS.2016.66.

[14] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, P. Faraboschi, S. Katti, Parallel Graph Processing: Prejudice and State of the Art, in: Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering, ICPE '16, 2016, pp. 85–90. doi:10.1145/2851553.2851572.

[15] S. Pollard, B. Norris, A Comparison of Parallel Graph Processing Implementations, in: Proceedings of the IEEE International Conference on Cluster Computing, CLUSTER '17, 2017, pp. 657–658. doi:10.1109/CLUSTER.2017.56.

[16] L. Jiang, L. Chen, J. Qiu, Performance Characterization of Multi-threaded Graph Processing Applications on Many-Integrated-Core Architecture, in: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '18, 2018, pp. 199–208. doi:10.1109/ISPASS.2018.00033.

[17] S. Eyerman, W. Heirman, K. D. Bois, J. Fryman, I. Hur, Many-core Graph Workload Analysis, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, 2018, pp. 22:1–22:11.

[18] S. Beamer, K. Asanovic, D. Patterson, The GAP Benchmark Suite (2017) 1–16.
URL http://arxiv.org/abs/1508.03619v4

[19] U. Brandes, A faster algorithm for betweenness centrality, J. Mathematical Sociology 25 (2) (2001) 163–177. doi:10.1080/0022250X.2001.9990249.

[20] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication, in: A. Jorge, L. Torgo, P. Brazdil, R. Camacho, J. Gama (Eds.), Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases, PKDD '05, 2005, pp. 133–145. doi:10.1007/11564126_17.

[21] 9th DIMACS Implementation Challenge - Shortest Paths (2006).
URL www.dis.uniroma1.it/challenge9

[22] H. Kwak, C. Lee, H. Park, M. Sue, What is Twitter, a Social Network or a News Media?, in: Proceedings of the 19th International Conference on World Wide Web, WWW '10, 2010, pp. 591–600. doi:10.1145/1772690.1772751.

[23] T. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software 38 (1) (2011) 1:1–1:25. doi:10.1145/2049662.2049663.
URL http://doi.acm.org/10.1145/2049662.2049663

[24] S. Lee, J. S. Meredith, J. S. Vetter, COMPASS: A Framework for Automated Performance Modeling and Prediction, in: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 15, 2015, p. 405414. doi:10.1145/2751205.2751220.

[25] R. Friese, N. Tallent, A. Vishnu, D. Kerbyson, A. Hoisie, Generating Performance Models for Irregular Applications, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 317–326. doi:10.1109/IPDPS.2017.61.

[26] G. Zhu, G. Agrawal, A Performance Prediction Framework for Irregular Applications, in: Proceedings of the IEEE 25th International Conference on High Performance Computing (HiPC), 2018, pp. 304–313. doi:10.1109/HiPC.2018.00042.

[27] A. Sen, M. Srivastava, Multiple Regression, in: Regression Analysis, Springer, 1990, pp. 28–59.

[28] A. Liaw, M. Wiener, et al., Classification and regression by randomForest, R News 2 (3) (2002) 18–22.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.