# CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores

Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, Omer Khan
{masab.ahmad, farrukh.hijaz, qingchuan.shi, khan}@uconn.edu

*University of Connecticut, Storrs, CT USA*

***Abstract*—Algorithms operating on a graph setting are known to be highly irregular and unstructured. This leads to workload imbalance and data locality challenge when these algorithms are parallelized and executed on the evolving multicore processors. Previous parallel benchmark suites for shared memory multicores have focused on various workload domains, such as scientific, graphics, vision, financial and media processing. However, these suites lack graph applications that must be evaluated in the context of architectural design space exploration for futuristic multicores.**

**This paper presents *CRONO*, a benchmark suite composed of multi-threaded graph algorithms for shared memory multicore processors. We analyze and characterize these benchmarks using a multicore simulator, as well as a real multicore machine setup. *CRONO* uses both synthetic and real world graphs. Our characterization shows that graph benchmarks are diverse and challenging in the context of scaling efficiency. They exhibit low locality due to unstructured memory access patterns, and incur fine-grain communication between threads. Energy overheads also occur due to nondeterministic memory and synchronization patterns on network connections. Our characterization reveals that these challenges remain in state-of-the-art graph algorithms, and in this context *CRONO* can be used to identify, analyze and develop novel architectural methods to mitigate their efficiency bottlenecks in futuristic multicore processors.**

## I. Introduction

Graph workloads ubiquitously arise in a variety of emerging application domains, such as cognitive computing [1], self-driving cars [2], and data analytics [3]. Recent explosion of data has further accelerated their usage, as well as challenges associated with their computational efficiency. Larger and more complex graphs, such as road networks [4] and social networks [5] now arise even in the personal computing space [6]. However, graph workloads are highly unstructured and irregular, and their analysis remains a bane for most computational architectures. Limited memory bandwidth and on-chip data access latency inhibits high performance [7]. Fine grain communication leads to low concurrency that further exacerbates this problem. Even parallel algorithms suffer from network congestion and traffic [8]. Their irregular data access patterns inhibit various architectural methods, such as out-of-order cores and on-chip caches, from performing well. To counter these constraints, prior research focuses on faster sequential machines and more recently specialized accelerator architectures.

Since single thread efficiency has slowed down due to the power wall, GPU based architectures have gained momentum to exploit performance for graph workloads [9]. GPUs offer large degree of hardware thread-level concurrency, which can be used to improve efficiency using workload parallelization strategies. Researchers have proposed GPU benchmark suites, such as Rodinia [10] and Pannotia [8], that optimize memory access patterns, load imbalance, and cache effects for various graph workloads. They also describe fundamental program properties for graph workloads with respect to their GPU implementations.

In the context of traditional multicore processors, the industry has already integrated multiple cores (4–16) on a single die [11]. Due to lack of high core count on a single chip, researchers are exploring processor clusters to perform parallelization studies for graph algorithms. Satish *et. al.* [12] discuss and analyze how various graph frameworks, such as Galios [3], GraphChi [6], GraphLab [13], and Combinatorial BLAS [14], scale and perform in distributed computing and supercomputing setups, such as the Graph500 [15]. Other researchers have evaluated primitive algorithms, such as Breadth First Search (BFS) on a variety of graphs, and shown that graph analytics exhibit significant nondeterminism [11][16]. However, these implementations suffer from a number of limitations, such as a lack of architectural characterization and standardization. They are not adequate enough to evaluate future large-scale single-chip multicore processors.

As conventional multicore processors get highly parallel, with notable commercial examples of Xeon Phi [17] and Tile64 [18], graph algorithms must scale well on such systems. Due to fine grain communication constraints, graph analytics has traditionally avoided parallel processor clusters. However, future multicores are expected to integrate hundreds of cores on a chip that are interconnected using on-chip networks and large distributed caches. Therefore, novel methods to address the concurrency bottlenecks in graph workloads must be studied for such futuristic multicores. Popular multicore benchmark suites, such as SPLASH [19] and PARSEC [20], do not contain many graph workloads. It is imperative to develop and characterize state-of-the-art graph benchmarks to explore the scalability of futuristic multicore processors.

We create an open source graph benchmark suite for shared memory multicores, called *CRONO*. It contains important and well known graph workloads, along with both synthetic and real world graph inputs. We present a thorough characterization of *CRONO* using both multicore simulator and a real multicore machine setup. Most of the graph benchmarks scale

to high thread counts, have diverse memory access patterns, and exhibit a variety of scalability challenges that must be studied for multicore architecture design space exploration. Our simulation based evaluation shows a diverse design space and fine grain view of architectural bottlenecks, while real machine results validate simulator trends. We make the following contributions:

- We present *CRONO*, a graph benchmark suite for shared memory multicores that uses state-of-the-art parallelization strategies. It encompasses the following benchmarks:
  - **Path Planning** : Single Source Shortest Path [8], All Pairs Shortest Path [21], and Betweenness Centrality [22].
  - **Search** : Breadth First Search [23], Depth First Search [24], and the Traveling Salesman Problem [24].
  - **Graph Processing** : Connected Components [25], Triangle Counting [12], PageRank [12], and Community Detection [26].
- We quantify the communication and computation tradeoffs stemming from synchronization and data sharing in graph analytics.
- We identify high degree of data sharing and network traffic as the key scalability bottlenecks in graph analytics. Based on our findings, we discuss inefficiencies that reside in today's architectures, and what future multicore processors must address to ensure scalability and high performance.

## II. Motivation

### A. Requirements for a Graph Analytics Benchmark Suite

**Multithreaded Shared Memory Workloads:** Trends in computing show that shared memory architectures will remain pervasive. Shared memory programming model allows ease of programming, where data sharing and inter-core communication is managed by the underlying cache coherence and consistency protocols. Novel architectural optimizations, such as the locality-aware coherence [27] and hardware consistency protocols [28], allow researchers to optimize parallel algorithms under the shared memory paradigm.

**Emerging Parallel Algorithms and Workloads:** Due to the progressive evolution of computing paradigms, algorithms change rapidly. Newer algorithms rise to fame, an example of which is Triangle Counting [12]. Moreover, contemporary algorithms, such as Depth First and Breadth First searches, are finding more usage in today's applications. State-of-the-art parallel graph algorithms can potentially deliver high performance when executed on concurrent hardware. However, the massive data and fine grain communication in these algorithms present their own set of challenges for future multicore processors. A multithreaded graph benchmark suite must contain algorithms that remain in use for time to come, as well as algorithms that have demonstrated potential.

**Diverse and Scalable Workloads:** Given the vast variety of available algorithms, any graph benchmark suite must contain parallel algorithms that are representative of graph analytics. Famous algorithms such as Breadth First search and Dijkstra's algorithm for shortest path computations fit well in this context.

Scalability also remains a major issue in parallel computing, and it is already known that graph workloads do not scale regularly [29]. However, some scalability, even if it is not linear, must be shown so future architectures may improve and build upon the existing challenges and enable performance improvements.

### B. Challenges in Existing Graph Analytics

Existing graph analytics research faces variations in algorithms, input data sets, and parallelization strategies. These variations cause differences in performance and energy characterization. They also cause various architecture implementations to exhibit different computation and communication behaviors. We outline some of these challenges.

**Irregular Data Access and Synchronization:** Graph workloads are highly irregular, and therefore exhibit different cache and synchronization effects. Irregular behavior also arises due to dynamic data dependencies within graph algorithms, leading to fine grain communication between threads and low locality for data accesses [8]. However, it still is not evident what methods must be deployed in futuristic multicore architectures to address the data locality and communication challenges.

**Lack of Parallelism and Load Balance:** Linear speedups are not fully observed in most graph workloads. Some work efficient algorithms, such as efficient heuristics for path planning, are not parallelizable, and even observe slowdowns in multithreaded setups. The lack of parallelization is observed because of synchronization and irregular data access behaviors. Due to this unstructured behavior the execution also observes load imbalance. Prior research on graph frameworks attempts to solve the load imbalance problem by using efficient scheduling methods [3]. However, these frameworks do not show scalability to high thread counts, or architectural characterization to identify where the bottlenecks arise.

**Lack of a Graph Benchmark Suite for Multicores:** Benchmark suites unify test workloads so that different researchers can properly compare their architecture methods against others, and justify why their scheme is better. The PARSEC and SPLASH suites are available for this purpose. However, they do not contain state-of-the-art graph workloads, and have scalability problems at high thread counts for various graph instances. The Graph500 suite [15] also contains several benchmarks, such as BFS, and is tailored for supercomputing setup with clusters of several nodes connected using a high bandwidth interconnect. However, it is not tailored for single-chip shared memory multicores, and it primarily relies on the Message Passing (MPI) paradigm for communication. Therefore, there exists a need for a graph benchmark suite for multicores, which is the primary focus of this paper.

## III. Overview of CRONO

In this section we discuss *CRONO* graph benchmarks and their parallelization strategies. These benchmarks are

parametrized to vary across thread count and input instances.

*1) **Single Source Shortest Path (SSSP) - Dijkstra**:* Dijkstra's algorithm computes shortest paths for graphs with non-negative edge weights, and is a popular benchmark used in various applications, such as self-driving cars [2]. The algorithm starts from a user defined vertex, and hops over all the vertices in the graph, updating neighboring vertices with lowest path costs from the starting vertex. A distance array is updated with these lowest path costs, while another data structure contains information about which vertices are already checked and which remain to be checked.

The algorithm consists of two main loops, an outer loop that visits all the vertices, and the inner loop which visits all the neighboring vertices of a given vertex, each of which can be parallelized. The outer loop can be parallelized in a controlled manner, where pareto fronts of vertices are opened and computed upon [8] [30]. Vertex path costs are updated using atomic locks, as threads may share vertices with common neighbors. On the other hand, the inner loop can be parallelized statically, dividing neighboring vertices amongst working threads. Real world graphs, such as road networks, are known to have a small numbers of neighboring vertices, and hence the outer loop parallelization works well in these cases.

*2) **All Pairs Shortest Path (APSP) - Floyd Warshall**:* The APSP benchmark is similar to SSSP. However, in this case a SSSP kernel is run for each vertex pair in the graph. We use the highly parallelizable Floyd Warshall algorithm [24], in which each thread 'captures' a vertex, and starts computing the shortest path to the destination vertex using Dijkstra's algorithm. Once a thread finishes working on its vertex, it captures another vertex to work on. Vertex capture is thus done via atomic locks, as two threads must not pick the same vertex. Each thread creates and maintains its own distance arrays and other data structures for shortest path computations. This is a similar parallel implementation as in [21].

*3) **Betweenness Centrality**:* The Betweenness Centrality benchmark identifies important vertices in a graph. This is done by computing all the shortest paths in a graph between all the pairs of vertices, and then identifying the number of shortest paths passing through each given vertex. Initially the algorithm computes APSPs using the highly parallel Floyd-Warshall algorithm [24], after which another loop runs over all the vertices and computes the number of shortest paths passing through each vertex. Some additional data structures are required in the APSP function executed earlier for this purpose. In our parallel version, APSP is executed as described in Section III-2, then a barrier is applied, and finally a loop executes to compute the centralities of each vertex. The final loop is statically divided amongst threads, with each thread reading shortest path values and updating the centralities via atomic locks. This is similar to the parallel implementation in [22].

*4) **Breadth First Search (BFS)**:* BFS is a highly popular algorithm in most graph applications [24]. The algorithm searches for a target vertex in a given graph, while doing a neighbors first type search. More parallelism is exploitable in BFS, and prior research has shown significant speedups [23]. Parallelism can be exploited via vertex capture, in which each thread picks a vertex and searches its neighbors. This is done in the inner loop where the neighbors are statically divided amongst threads. Vertex capture is done via atomic locks, while a barrier is required in inner loop based parallelism to hop to the next vertex in each iteration.

*5) **Depth First Search (DFS)**:* Like BFS, DFS is also a popular benchmark in various graph applications [24]. This algorithm also searches for a target vertex in a graph, with it performing a first-come first-served type search involving branches. Branches are connected components of a graph that extend outward like branches in a tree from a source vertex, in contrast to a BFS where the neighboring vertices are considered first. These branches can be searched in parallel, depending on the density and connectivity of the graph. In the case of DFS, only branch level parallelism is available, and hence more parallelism can only be exploited if the branches are long enough to offset computation versus communication ratios. A long branch implies that a thread has to spend more time in compute rather than communicate more often to acquire subsequent branches.

*6) **Travelling Salesman Problem (TSP)**:* TSP is a NP-hard problem that has been studied in detail for quite some time now [24]. It involves computations that determine a shortest path that can traverse all cities, where cities are in conjunction with vertices, given as an input. Parallel versions involve branch and bound implementations [31], with each thread working on a branch of a possible shortest path. A global bound is kept as the shortest path cost found in each iteration, and threads whose costs become greater than this bound stop working on their branch and move on to the next branch. Intuitively, branches are designated at static time, while the global bound is maintained dynamically via an atomic lock.

*7) **Connected Components**:* Connected Components is a formidable graph workload used primarily to measure connected regions in images. Clustering applications may also employ this workload [5]. A global data structure is maintained by this algorithm, and contains labels for each vertex in the graph. In the initialization phase, if the vertices are within a connected region then their labels are set to that connected region. A loop then runs over all the vertices in the graph, maintaining and updating labels iteratively. Label updates are done on the basis of connectivity and edge weights. Vertices with common labels count towards a connected component of the graph. In a parallel implementation, this loop is statically divided amongst threads. Barriers separate functions that set and update these labels. Our implementation of connected components is a modified version of prior work [25].

*8) **Triangle Counting**:* Triangle counting is an important graph workload to measure graph statistics regarding vertex connections and sharing. Triangles are formed in a graph when three vertices are connected to each other. The algorithm statically divides the graph into threads, after which each thread starts finding triangles in its allocated section. A global

data structure is maintained for each vertex, which stores the connections between vertices. The loop then runs over all vertices inside each thread, and updates to the global data structure are done via atomic locks. Then a barrier is applied, after which another loop runs, which is also statically divided amongst threads that compute the number of triangles for each vertex. A vertex may be connected to many other vertices, and thus may form multiple triangles. We use the exact version of the algorithm, and explanation from [12] to construct our implementation.

*9)* ***PageRank****:* PageRank is a well known algorithm used by web services, social networks, and search algorithms. It uses probability distributions to compute ranking of pages in a given graph. The rankings themselves are also probabilities which specify the likelihood that a person on the internet will visit a page. From many different PageRank implementations, we base our per iteration implementation on [12], with no approximations. The algorithm initializes the probability of each vertex to the inverse of the total number of vertices, and then uses the formula in Equation 1 to compute the page ranks of each vertex.

$$PR^{t+1}(i) = r + (1-r) * \sum_{j,neighbors} \frac{PR^{t}(j)}{degree(j)} \quad (1)$$

*PR* represents the page rank of vertex *i* at each iteration, *r* denotes the probability of a random page visit by a user, and *degree* is the number of neighboring vertices of vertex *i*. In our parallel implementation, we have a floating point array for the probability distribution of the input graph, and another array for the page ranks of the vertices. The graph is statically divided amongst threads, with updates for page ranks done via atomic locks, as threads may converge on common neighbors from their given vertices.

*10)* ***Community Detection****:* Community Detection is a highly irregular and inherently sequential algorithm to detect communities with similar characteristics. The Louvain algorithm is the most efficient method in this domain, and uses heuristics to relax the inherently sequential inter-vertex community dependencies. It optimizes modularity, a measure of connectivity in a graph, which is later used to detect communities. We use a parallel version of the Louvain algorithm [26], and use a bounded heuristic to relax modularity approximations. This improves scalability at high thread counts, while propagating a loss of modularity accuracy with the increase in parallelism. The graph is statically divided amongst threads, with each thread recursively placing vertices in communities of other vertices that maximize the overall modularity. The algorithm terminates when the modularity can not be increased any further.

## IV. Methods

### A. Summary of Benchmarks and their Parallelizations

All graph algorithms follow a generic structure that consists of an outer loop and an inner loop [8]. The *outer loop* traverses the graph vertices, while the *inner loop* traverses the neighboring vertices only for a given vertex. For most cases, more outer loop parallelization is available as graph vertices (outer loop iterations) always outnumber inner loop iterations. *Graph division* [12] is an outer loop parallelization technique in which the input graph is statically or dynamically divided amongst threads. Threads use locks and barrier to synchronize shared variables or parts of the graph. *Vertex capture* [32] is also an outer loop parallelization technique in which threads compete for vertices (compete for work) dynamically. For workloads having many combinations of branches, such as in *TSP*, *branch and bound* strategies are applied [31]. These employ threads traversing different branches, and stop working on branches whose cost outweighs a global cost. Global costs are updated via locks at threshold based iterations, where thresholds are defined by heuristics. Table I provides a summary of *CRONO* benchmarks and their parallelization strategies.

TABLE I
Benchmarks and Parallelizations used for Evaluation.

| Benchmark Identifiers | Parellelizations |
|---|---|
| SSSP_DIJK | Graph Division [8] |
| APSP | Vertex Capture [32] |
| BETW_CENT | Vertex Capture & Outer Loop [22] |
| BFS | Graph Division [23] |
| DFS | Branch and Bound [31] |
| TSP | Branch and Bound [31] |
| CONN_COMP | Graph Division [8] |
| TRI_CNT | Vertex Capture & Graph Division [12] |
| PageRank | Vertex Capture & Graph Division [12] |
| COMM | Vertex Capture & Graph Division [26] |

All benchmarks are parallelized using the popular `pthreads` API. We execute each benchmark compiled using `-O2` and `-O3` flags, and then report speedups based on completion times for both real machine and simulator setups.

### B. Many-core Simulator Setup

We evaluate *CRONO* using the Graphite simulator that models futuristic shared memory multicores [33]. We consider a 256 core NoC-based multicore organization with a two-level private L1, shared L2 cache hierarchy per core. The upper limit of 256 for the core count is chosen as it shows all of the effects of scalability. The architectural parameters used for evaluation are shown in Table II. We use the Graphite simulator because many-core chips with hundreds of cores do not exist yet, whereas Graphite can simulate up to a thousand cores. Multicore configurations with both in-order and out-of-order cores are simulated for architectural exploration. To mitigate simulation slowdowns at large core counts, Graphite relaxes cycle accuracy and uses multithreading for increased performance.

### C. Many-core Real Machine Setup

We also evaluate our results on a real multicore machine to validate result trends observed with the simulator. We use an Intel i7-4790 machine clocked at $3.6GHz$ with 4 out-of-order two-way hyperthreaded cores, an $8MB$ shared L3 cache, and a $256KB$ per-core private L2 cache [35].

TABLE II
GRAPHITE ARCHITECTURAL PARAMETERS FOR EVALUATION.

| Architectural Parameter | Value |
|---|---|
| Number of Cores | 256 @ 1 GHz |
| In-Order Core Setup | |
| Compute Pipeline per core | Single-Issue Core |
| Out-of-Order Core Setup | |
| Compute Pipeline per core | Single-Issue Core<br>Out-of-Order Memory |
| Reorder Buffer Size | 168 |
| Load/Store Queue Size | 64/48 |
| Memory Subsystem | |
| L1-I Cache per core | 32 KB, 4-way Assoc., 1 cycle |
| L1-D Cache per core | 32 KB, 4-way Assoc., 1 cycle |
| L2 Cache per core | 256 KB, 8-way Assoc., 8 cycle<br>Inclusive, NUCA |
| Cache Line Size | 64 bytes |
| Directory Protocol | Invalidation-based MESI<br>ACKWise$_4$ directory [34] |
| Num. of Memory Controllers | 8 |
| DRAM Bandwidth | 5 GBps per controller |
| DRAM Latency | 100 ns |
| Electrical 2-D Mesh with XY Routing | |
| Hop Latency | 2 cycles (1-router, 1-link) |
| Contention Model | Only link contention<br>(Infinite input buffers) |
| Flit Width | 64 bits |

### D. Benchmark Characterization

For each simulation run, we measure the *Completion Time*, i.e., the time in *parallel* region of the benchmark; this includes the compute latency, the memory access latency, and the synchronization latency. The memory access latency is further broken down into four components. (1) **L1Cache-L2Cache** latency is the time spent by the L1 cache miss request to the L2 cache and the corresponding reply from the L2 cache including time spent in the network and the first access to the L2 cache. (2) **L2Home-Waiting** time is the queueing delay incurred because requests to the same cache line must be serialized to ensure memory consistency. (3) **L2Cache-Sharers** latency is the roundtrip time needed to invalidate private sharers and receive their acknowledgments. This also includes time spent requesting and receiving synchronous write-backs. (4) **L2Home-OffChip** memory latency is the time spent accessing memory including the time spent communicating with the memory controller and the queueing delay incurred due to finite off-chip bandwidth.

For on-chip private L1 cache misses, we analyze the following types of misses: (1) **Cold misses** are cache misses that occur to a cache line that has never been previously brought into the cache, (2) **Capacity misses** are cache misses to a cache line that was brought in previously but later evicted to make room for another cache line, and (3) **Sharing misses** are cache misses to a cache line that was brought in previously but was invalidated or downgraded due to a read/write request by another core.

To evaluate the overall on-chip cache effects, we measure the **cache hierarchy miss rate**, which is the number of L2 cache misses divided by the total number of L1 cache accesses.

We also measure the dynamic **energy consumption** for the memory system including on-chip and off-chip data accesses, and the on-chip network. For energy evaluations of on-chip electrical network routers and links, we use the DSENT [36] tool. Energy estimates for the L1-I, L1-D and L2 (with integrated directory) caches are obtained using McPAT [37]. The evaluation is performed at the $11nm$ technology node to account for future technology trends.

TABLE III
INPUT GRAPHS FOR EVALUATION.

| Dataset | # Vertices | # Edges |
|---|---|---|
| **Road Networks [4]** | | |
| Texas | 1,379,917 | 1,921,660 |
| Pennsylvania | 1,088,092 | 1,541,898 |
| California | 1,965,206 | 2,766,607 |
| **Social Networks [5]** | | |
| Facebook | 2,937,612 | 41,919,708 |
| **Synthetic [38]** | | |
| Sparse | 1,048,576 | 16,777,216 |
| Cities for TSP | 32 Cities | |

### E. Load Imbalance

Load imbalance is a primary cause of performance degradation in many-core processors [7]. Instruction counts are generally used as a metric in this case, because they incorporate communication as well as compute instructions. We define the load imbalance metric in equation (2). *Max(thread inst.)* is the instruction count of the thread that executes most number of instructions, while *Min(thread inst.)* is the thread with minimum instruction count. *Variability* defines how much imbalance a thread has with respect to other threads in the worst case. We use the range of instructions because it envisions the loads across the multicore, from which it quantifies the extreme data points.

$$Variability = \frac{Max(thread\,inst.) - Min(thread\,inst.)}{Max(thread\,inst.)} \quad (2)$$

### F. Input Graphs and Data Structures

*CRONO* uses several synthetic and real world input graph types. Real world graphs are taken from the SNAP dataset directory, which contains several graph types such as road networks [4], citation networks, and social networks [5]. We also generate random synthetic graphs using a modified version of the GTgraph generator [38]. Compared with prior works, *CRONO*'s graph generators are included within the programs, and can sustain significantly large datasets, from several kilobytes to several gigabyte or higher sizes, based on the user's specification. Generated graphs are converted to an adjacency list representation, which contains a data structure for vertex connections and another structure for edge weights. All data structures are cache line aligned to ensure optimal performance. With the exception of *TSP*, which uses 32 cities as

an input, all benchmarks are able to use road networks, social networks, and synthetic graphs. *APSP* and *BETW_CENT* uses an adjacency matrix representation containing 16,384 vertices. To allow subtle comparisons between algorithms in terms of characterization, we use sparse synthetic graphs as inputs. The synthetic graph generators are bundled with the workloads, using which testing can be done for small test graphs. An overview of these graphs is listed in Table III.

## V. Characterization

In this section we discuss characterization results for each *CRONO* benchmark. Analysis is done on the basis of parallelism, work efficiency, and speedups. Architectural analysis is done using detailed execution time breakdowns, containing information on where the time is spent in the architecture for each workload. We also identify bottlenecks generalized across all graph workloads as well as for individual benchmarks. The evaluation uses in-order cores and synthetic sparse graphs as default, unless otherwise stated.

### A. Computation, Communication, and Scalability

Figure 1 shows the detailed completion time breakdowns for all benchmarks. All benchmarks, except *DFS*, *TSP* and *COMM*, scale up to 256 threads. Scalability, however is not fully linear in all cases. *SSSP_DIJK* scales to 256 threads and gives a maximum speedup of 4.45×. Graph division based parallelism, which divides the graph into threads to exploit vertex-level parallelism, is used in *BFS*, *TRI_CNT*, and *PageRank*, and therefore these benchmarks show similar trends. Benchmarks containing data dependent memory accesses, such as *SSSP_DIJK* and *PageRank*, scale less than benchmarks containing less dependencies, such as *BFS*, which scales to 8.26×.

Synchronization and coherence overheads (L2Home-Waiting and L2Home-Sharing times) are the primary factors leading to diminished scalability. These arise from threads working on shared data and/or waiting at the atomic locks or barriers. L2Home-Waiting times are higher at intermediate thread counts, as threads wait more on locked data that is being computed on by other threads. However, with increase in thread counts, each thread is able to allocate it's graph chunk within its private L1 cache, and thus the waiting time at the shared cache decreases. Synchronization time increases as L1 traffic needs to be orchestrated for shared vertices, and it remains a major problem even when graph chunks local to a thread fit in the private caches.

*COMM*, an inherently sequential benchmark parallelized using heuristics, fails to scale to 256 threads due to growing synchronization caused by algorithmic constraints. A more approximate heuristic might allow further scaling, but will also degrade accuracy. Communication due to dependencies also plays a major role in the scalability of *CONN_COMP*. It scales less than *APSP* and *BETW_CENT*, and is mainly bottlenecked by synchronization and L2Home-Waiting times. Dependencies stall threads from accessing data already being used by another thread, hence threads have to wait longer to access a cache line, contributing further to on-chip communication latency.

L2Home-Sharing is the time spent ensuring cache coherence for shared cache lines containing vertices and other shared global variables. For example, the *DFS* and *TSP* plots show significant L2Home-Sharing times in Figure 1. This stems from vertices that are shared in branches that are allocated to different threads. This is evident from the *branch and bound* parallelization strategy. However, *DFS* scales less than *TSP* since *DFS* is an inherently sequential algorithm, and also because it has more vertex level dependencies, as evident from its higher L2Home-Sharers times. *TSP*, on the other hand, has more combinations to exploit parallelism, with only partial vertex level dependencies that serialize on a bound check. Therefore, *TSP* scales to 128 threads and attains a speedup of 10.7×, which is more than twice of what is achieved for *DFS*.

Compute times are observed to be highly scalable, as evident from parallelization strategies. The L1Cache-L2Home times decrease for all benchmarks since available L1 cache capacity increases at higher thread counts. In case of highly scalable benchmarks, such as *APSP* and *CONN_COMP*, benefits are gained from reduction in the Compute and L1Cache-L2Home times. The communication component, such as Synchronization, is much smaller compared to the Compute component in these benchmarks.

Off-chip memory access times, shown by the L2Home-OffChip component, are higher in percentage at lower thread counts. However, this component gets distributed as more threads exploit the available memory level parallelism. We observe that at the best thread count, L2Home-OffChip does not contribute to the lack of scalability. Hence, memory bandwidth is not a prominent issue in our graph benchmarks.

**Scalability Summary:** Most graph benchmarks exhibit weak scaling. The primary reason from our characterization is that on-chip communication, such as synchronization and cache coherence traffic contributes mainly to the lack of scalability.

### B. Load Imbalance

Load imbalance remains an undesirable characteristic in graph workloads. Benchmarks such as *SSSP_DIJK* and *PageRank*, which use the Graph division parallelization strategy, exhibit much lower imbalance. This is because most graph workloads are 'vertex centric', meaning that most of the compute and communication is done because of the vertices, and not because of the edges or other variables. Once a graph is divided, statically or dynamically, load balancing strategies come into play. Our workloads use various methods to reduce load imbalance, such as cache line alignment of data structures to mitigate redundant cache line sharing and synchronization.

In the case of *SSSP_DIJK*, dynamic load balancing of threads by allocating 'pareto fronts' of vertices improves performance. In all benchmarks, load imbalance increases at higher thread count (see Variability component in Figure 1). This is because work is distributed amongst threads in a fine grain manner, which results in a few threads doing variable

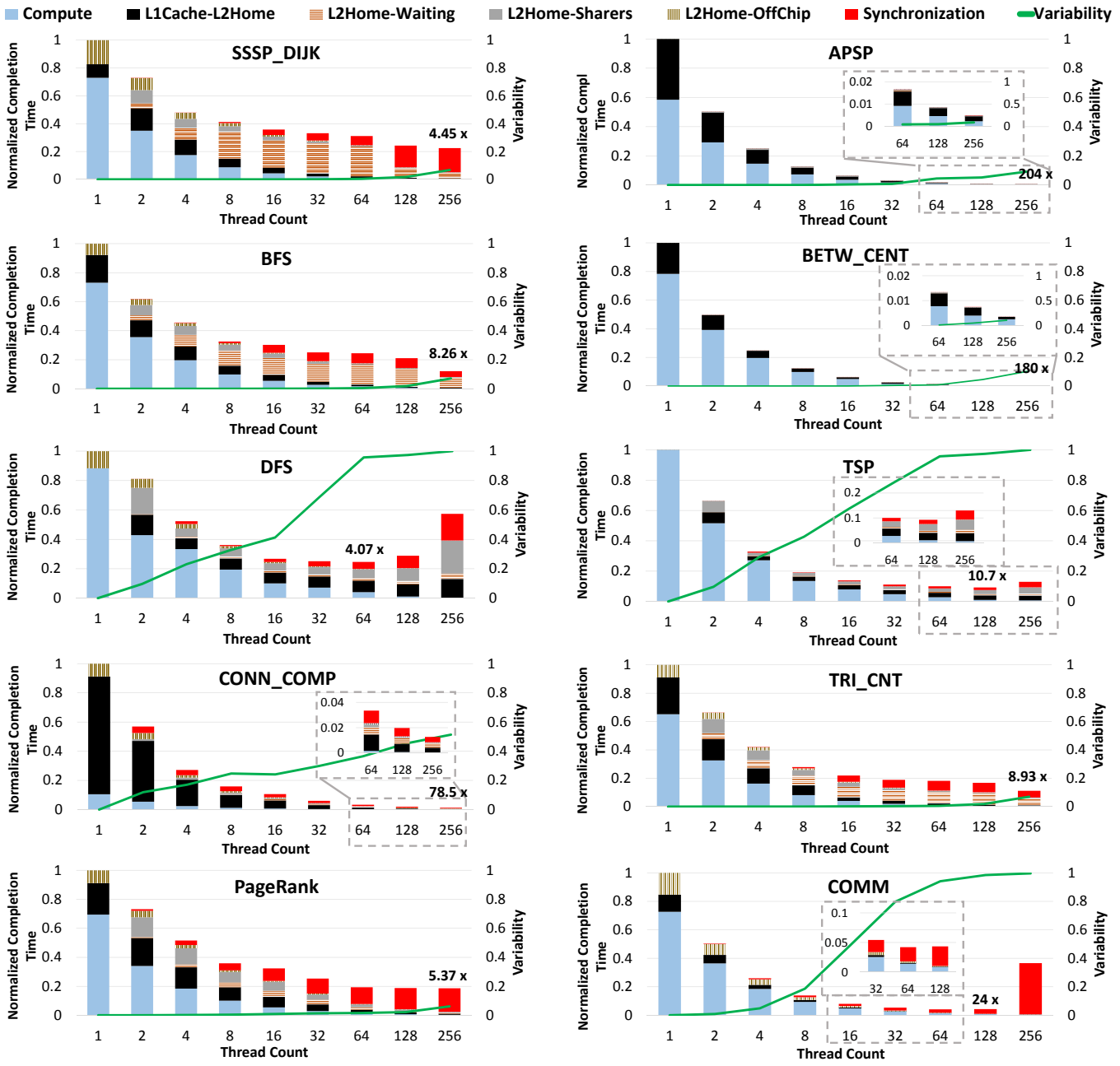


Fig. 1. Normalized completion time breakdowns for *CRONO* benchmarks. Speedup over the sequential version is shown above the best thread count.

work either due to data sharing effects, or because those threads simply got a chunk of vertices that require less work. However, in other benchmarks such as *DFS* and *TSP*, load imbalance remains high, mainly because of threads working less due to bound constraints.

### C. Parallelism and Graph Access Patterns

Graph algorithms need to sustain parallelism to ensure scalability without bottlenecks. We conduct a study of active vertices at best thread counts for the *CRONO* benchmarks. Active vertices per unit time define how much exploitable parallelism is available, as well as some visualization of the program's memory access pattern. A larger number of vertices active during a course of work for a benchmark show that the benchmark requires a larger memory at that unit time. This may also be used as an indicator of how many threads are active at any time.

Figure 2 shows how active vertices change during the course of execution for our benchmarks with sparse inputs. Active vertices and completion time both have been normalized to allow for a subtle comparative analysis. The shortest path benchmarks (*SSSP_DIJK*, *APSP*, and *BETW_CENT*) dynamically open pareto fronts that steadily increase parallel work. After some course of action, the pareto fronts dwindle as less vertices remain to be worked on, and thus parallelism decreases.

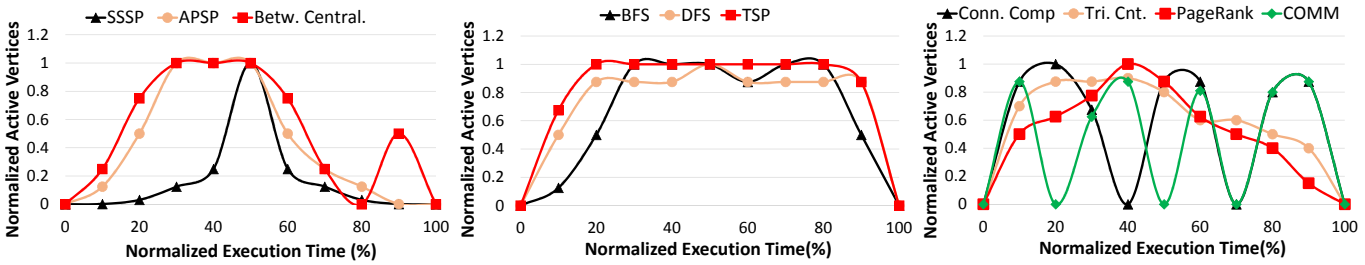


Fig. 2. Active vertices study for *CRONO* benchmarks. Completion time shown as normalized (0%-100%). Active Vertices also shown as normalized with respect to the maximum vertex count (0-1).

For *BETW_CENT*, we observe a spike at the end, during which the algorithm runs some iterations to determine centralities of vertices after it had executed an instance of *APSP*. In the case of *BFS, DFS* and *TSP* benchmarks, active vertices remain consistently high due to the available parallelism during execution. However, synchronization plays a major factor in these algorithms, as seen earlier. For *CONN_COMP* and *COMM*, the access pattern is sinusoidal, as the algorithm does three main parallel function calls, with each call separated by a barrier instance. *PageRank* and *TRI_CNT* have similar active vertex patterns, as they use a similar graph division parallelization structure.

### D. Data Locality and Cache Effects

This section discusses the private L1 and shared L2 cache effects for the *CRONO* benchmarks. Figure 3 shows the L1 cache miss rate breakdowns. The *APSP, BETW_CENT*, and *CONN_COMP* benchmarks have high capacity miss rates. This occurs due to low locality in these workloads. *APSP* and *BETW_CENT* both use the Floyd-Warshall algorithm, which initializes large input graph data structures for each thread, leading to L1 cache thrashing. *CONN_COMP* shows a high capacity miss rate because of its low locality, even though it is seen to be somewhat scalable in Figure 1. The primary reason for this is that the benchmark requires larger data structures to maintain graph connectivity labels, and hence a larger private cache capacity is required. For the remaining benchmarks encompassing both graph division and branch and bound parallelization strategies, high sharing miss rates are observed. These occur due to ping-pong of shared global variables between threads, and also due to invalidation messages arising from shared vertices. In Section VII we discuss how these sharing misses can be mitigated using intelligent caching schemes for workloads such as *PageRank* and *SSSP_DIJK* .

To view the overall on-chip cache effects, we plot the cache hierarchy miss rate for all *CRONO* benchmarks in Figure 4. The cache hierarchy miss rate is low, with the exception of *CONN_COMP*, implying that most benchmarks do not put exorbitant pressure on the memory bandwidth. *CONN_COMP* has higher L1 capacity misses (as discussed earlier) that creep into the overall cache hierarchy effects. Benchmarks working on smaller graph sizes, such as *APSP*, *BETW_CENT*, and *TSP* have negligible pressure on the on-chip cache hierarchy.

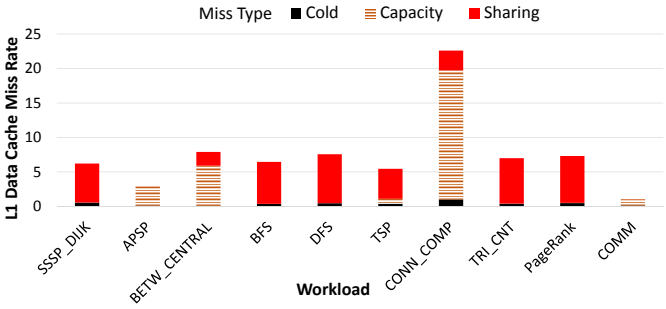


Fig. 3. Private L1 cache miss rates at thread counts that give the highest speedup

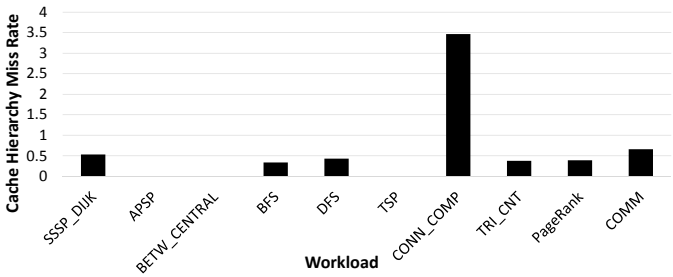


Fig. 4. Cache hierarchy miss rates at thread counts that give the highest speedup

Overall, with the exception of *CONN_COMP*, all benchmarks exhibit high cache hit rates, thus exhibiting effective opportunities for efficient parallelization outcome.

### E. Graph Dependence and Vertex Scalability

Various input graphs exhibit different scalability that must be considered in graph workloads. Therefore, we take several real world graphs, along with synthetic graphs (sparse), to evaluate *CRONO* benchmarks. Sparse graphs are regularly used in most applications, and have a small number of neighboring vertices connected to any vertex. Table IV shows best speedups for *CRONO* benchmarks on various graphs described in Section IV-F. Similar trends are seen for synthetic sparse, road network, and social network, graphs, mainly due to similar sparsity. The Facebook graph provides larger speedups as it is sparse and has more vertices, which exploits more outer loop scalability.

Input scalability also is an important issue in graph workloads. Graph sizes vary with the numbers of vertices, for

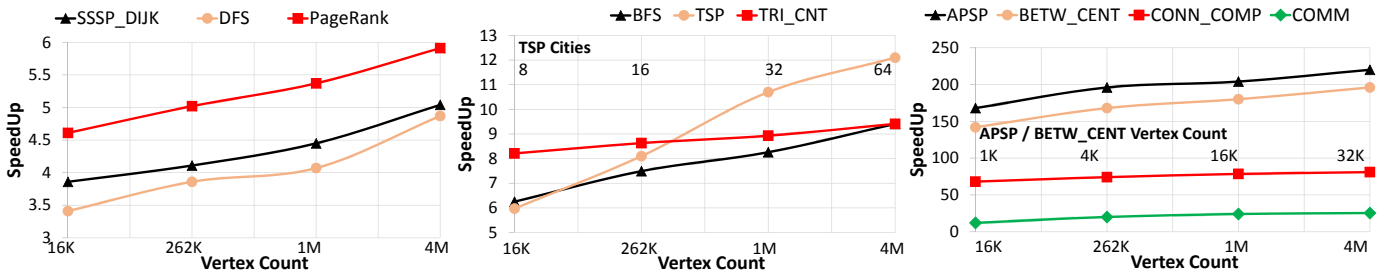


Fig. 5. Vertex scalability study for *CRONO* benchmarks. Speedups are shown for the best thread counts. Vertex/City counts for *APSP, BETW_CENT* and *TSP* are shown separately.

example the road network of a city will have less vertices than that of a state or a country. Therefore, graph algorithms must show vertex scalability to be considered scalable. We scale vertex counts for sparse graphs from $16K$ to $4M$ vertices. For *APSP* and *BETW_CENT* we scale from $1K$ to $32K$ vertices, and for *TSP* we scale from 4 to 32 cities. Figure 5 shows the results for different vertex counts. All benchmarks scale as expected.

TABLE IV
GRAPH VARIATION RESULTS FOR *CRONO* BENCHMARKS. ALL SPEEDUPS ARE RELATIVE TO THE SEQUENTIAL VERSIONS (BEST SPEEDUPS).

| **Algorithm** | **Sparse** | **TX** | **PN** | **CA** | **FB.** |
|---|---|---|---|---|---|
| | Synth. | Road.Net. | | | Social.Net. |
| SSSP_DIJK | 4.45 | 4.1 | 4.31 | 4.24 | 6.62 |
| APSP | 204 | - | - | - | - |
| BETW_CENT | 180 | - | - | - | - |
| BFS | 8.26 | 8.14 | 7.82 | 8.21 | 8.81 |
| DFS | 3.57 | 3.14 | 3.37 | 3.26 | 3.62 |
| TSP(32 Cities) | 10.7 | - | - | - | - |
| CONN_COMP | 78.5 | 65.1 | 66.1 | 66.4 | 82.1 |
| TRI_CNT | 8.93 | 8.12 | 8.21 | 8.19 | 9.53 |
| PageRank | 5.37 | 4.91 | 5.22 | 5.14 | 5.66 |
| COMM | 24 | 21.1 | 21.8 | 21.5 | 22.3 |

## F. Energy Analysis

Dynamic energy consumption is an important metric in modern architectures that are constrained by the power wall. Specifically, we characterize the energy consumption of the memory system components. Figure 6 shows normalized energy breakdowns for all *CRONO* benchmarks using sparse synthetic graph inputs. Sparse graphs ($1M$ vertices with 16 edges per vertex) are used in this context, with the exception of *APSP* and *BETW_CENT*, which use a $16K$ vertex graph, and *TSP*, which uses 32 cities. All benchmarks are seen to dissipate a high portion of the energy (an average of 75%) in the network link and the network router, which shows that graph benchmarks stress the on-chip network. Benchmarks such as *APSP* and *TSP*, which have small working sets but reuse a lot of data with large programs, stress their L1-D and L1-I caches significantly. *COMM* shows a higher DRAM component due to the larger L2Home-OffChip component visible in *COMM*'s breakdown in Figure 1.

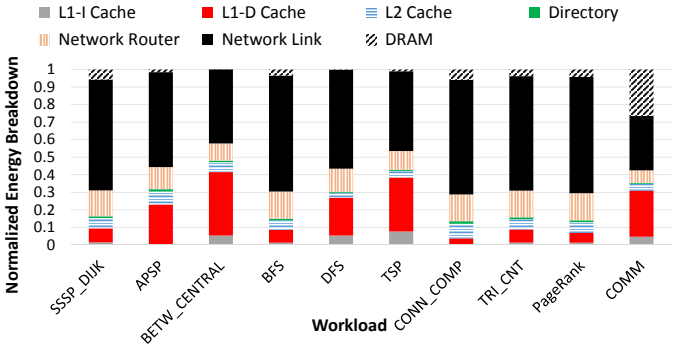


Fig. 6. Normalized dynamic energy breakdowns for *CRONO* benchmarks.

## G. Core Type Analysis

In this section we analyze the effects of out-of-order (OOO) core type for the *CRONO* benchmarks. OOO cores have the potential to hide the memory subsystem latencies by overlapping computation with communication. The key question we seek to answer is whether the bottlenecks observed for the in-order cores can be mitigated if OOO cores are used? We use the OOO core type from Table II and plot the associated normalized completion time breakdowns in Figure 7. We stack the normalized completion time to show a better view of each component. Speedups at the best thread count over the sequential OOO core are plotted in Figure 8.

Overall, trends in scalability remain similar to the in-order cores from Figure 1. Compute bound benchmarks, such as *APSP* and *BETW_CENT* scale, and are expected to continue scaling as they consist of large compute and L1Cache-L2Home components. *CONN_COMP* and *COMM* still suffer from synchronization and communication bottlenecks. Benchmarks with data dependencies, such as *SSSP_DIJK*, *PageRank*, *TRI_CNT* and *COMM*, have subtle synchronization and coherence (L2Home-Waiting and L2Home-Sharers) components.

Branch and Bound benchmarks, such as *DFS* and *TSP*, show smaller speedups compared to their in-order counterparts. This happens because the sequential OOO core performs significantly better than a sequential in-order core. Furthermore, *DFS* and *TSP* suffer from a combination of completion time components such as L1Cache-L2Home, L2Home-Waiting, and L2Home-Sharers, showing that an OOO core is unable to improve on-chip communication in these benchmarks. Viewing these results for the *CRONO* benchmarks, we conclude

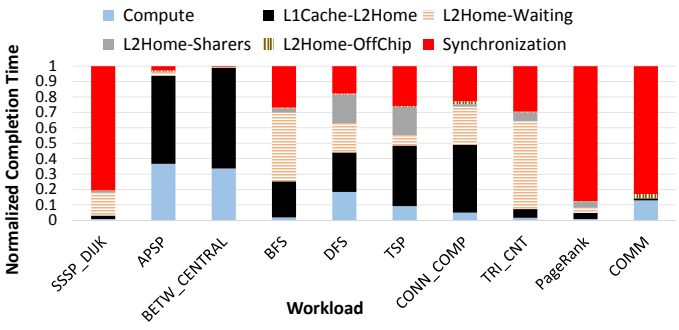


Fig. 7. Normalized completion time at the best thread count for an OOO core based simulated multicore.

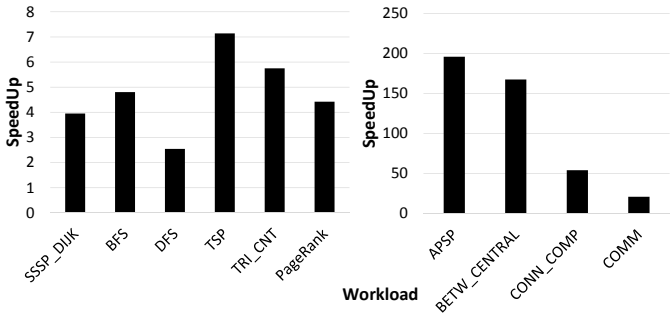


Fig. 8. Speedups at the best thread count over a sequential OOO core.

that OOO cores can not hide latencies associated with on-chip communication. Therefore, alternative research such as improving caches and on-chip networks must be undertaken to improve the bottlenecks in graph analytic workloads.

## VI. Real Machine Results

We also executed our benchmarks on a real machine setup to validate our simulator results. Figure 9 shows the speedup for each benchmark using a sparse input graph, relative to its sequential version. All results give similar speedups compared to the results obtained from the Graphite simulator in Figures 1 and 8. However, speedups are higher at lower thread counts for less scalable benchmarks, such as *SSSP_DIJK* and *PageRank*. Benchmarks such as *APSP* and *BETW_CENT* scale linearly for the real machine setup as well. Speedups reduce at 16 threads as the operating system begins to share 16 threads via context switching on 8 cores.

Further architectural optimizations are enabled in the real machine setup, such as a deeper cache hierarchy, data prefetching to reduce off-chip bandwidth limitations, and complex networks to reduce contention. However, the main limitation of the real machine setup is that it has a smaller number of cores. Future multicores are expected to integrate a large number of cores on a chip, and they may use *CRONO* as an evaluation suite for graph analytic workloads.

## VII. Discussion and Future Work

Based on our analysis and observations, we optimize our benchmarks for locality and scalability. However, some issues

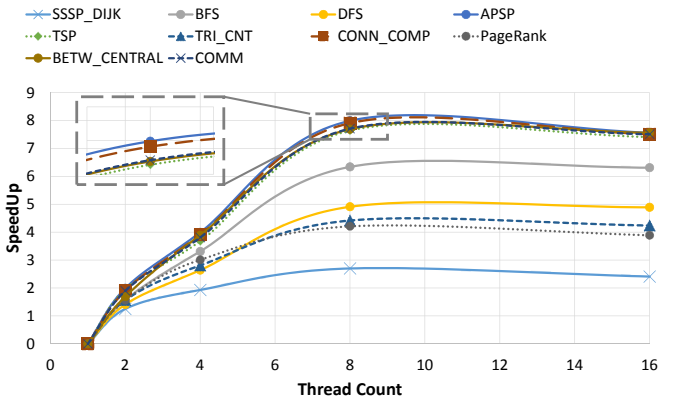


Fig. 9. Speedups for the *CRONO* benchmarks using the real machine setup from Section IV-C.

still remain, such as high cache and network latency and energy consumption. We provide some insights on how future architectures may improve to mitigate bottlenecks associated with graph analytic workloads.

### A. Scalability and Architectural Optimizations

We have shown that while most graph algorithms scale to high thread counts with the right type of parallelization, others scale to only an intermediate thread count. The ones that scale also are not expected to scale to very high thread counts, such as up to a thousand core using the current parallelization settings. Therefore, algorithms need to become *adaptive* in the sense of load balancing. Graph frameworks, such as Galios [3], do improve scheduling paradigms to mitigate load imbalance. However, these frameworks need to be extended for conventional parallel programs with architectural support to improve scalability.

Emerging data access mechanisms, such as the locality aware coherence protocol may be used to improve on-chip data locality [27]. This protocol allows private caching of cache lines that demonstrate high spatio-temporal locality at runtime, whereas low locality data is not allowed allocation in the private L1 caches. As a result, the private L1 caches do not thrash. Moreover, since shared data with low locality is never replicated in the L1 caches, the protocol results in significant reduction in on-chip traffic. In the context of graph analytics, global variables such as global bounds in branch and bound algorithms, and read-write shared data structures can be efficiently accessed using such architectural optimizations.

### B. Thread and Network Issues

The dynamic energy consumption analysis in Section V-F shows that all graph benchmarks spend a significant portion of their energy consumption in the on-chip network. This implies that graph workloads stress the network greatly, causing traffic and contention. Routing protocols, such as oblivious routing [39], may be able to reduce contention. Contention in graph workloads mostly arises from shared read-write data structures, such as distance arrays in the case of path planning algorithms, and website ranking in the case of *PageRank*.

Some of this contended data is more useful to some threads, which are slower due to locks residing in other threads. This behavior ultimately leads to high load imbalance. Routing protocols need to provide low contention routes for such data, and therefore classification of packets with respect to threads is desirable. Other ideas include speeding up master threads using out-of-order cores or even hardware accelerators, leading to a design space exploration of heterogeneous architectures [10] [40].

## VIII. Related Work

Several benchmark suites exist containing graph workloads. Examples of these include MiBench [32], Parboil [41], PBBS [42], Rodinia [10], and Lonestar [43]. Other notable works include studies by Harish *et. al* [21], Burtscher *et. al* [29], [11] and [16]. Such suites contain famous graph workloads such as *BFS* and *SSSP_DIJK*. The Graph500 Suite [15] also has several workloads for different real world graphs running in distributed setups such as supercomputing applications. However, because these suites generalize workloads from different domains and across specific architectures, such as GPUs, the essence of these suites does not center on graph benchmarks. The workloads in these suites are also highly regularized, and do not perform studies such as input scalability and detailed architectural analysis. Harish *et. al* [21] performed a detailed study across different graph workloads on GPUs. Their implementations include several diverse benchmarks parallelized using a generic program skeleton. Pannotia [8] is the only benchmark suite directed towards graph analytics, with it also being implemented and evaluated for GPUs. However, these works have several shortcomings such as lack of input and architectural scalability studies. In the case of multicores, on-chip cache and network effects are more predominant, and need to be characterized effectively.

Graph frameworks remain the current state-of-the-art for graph analytics in multicore processors and distributed setups [44]. Pregel [45], Galios [3], GraphChi [6], and GraphLab [13], are some instances to name a few. Each framework focuses on a specific domain, such as Pregel focuses on web based graphs, Galios generalizes algorithms across road networks and random graphs, and GraphLab associated itself for machine learning big data paradigms. These frameworks also introduce various scheduling schemes optimized for their domains. They show performance studies for several graph workloads to validate their scheduling infrastructures. However, these graph frameworks require special programming models, and cannot be generalized for all multicore processor benchmarks, such as PARSEC [20] and SPLASH [19]. Parallelization paradigms such as Cilk [46] provide parallel programming language variants for C/C++ programs. These variants can be used to create schedulers that do automatic load balancing between threads to ensure scalability. In contrast to these frameworks and models, we provide a complete benchmark suite along with their parallelization characterization. We analyze across a variety of behaviors such as graph dependence, architectural scalability, and memory access patterns.

## IX. Conclusion

This paper presents and characterizes *CRONO*, a scalable and diverse graph benchmark suite for multicore processors. *CRONO* is implemented using conventional POSIX threads, and can run on any conventional machine. We focus on new widely used applications, such as *PageRank* and Path planning since graph analytics is becoming increasingly popular. We characterize *CRONO* across various parameters, such as energy, graph dependence, architectural bottlenecks, and data sharing. Our analysis shows the following deductions:

- Graph analytic workloads exhibit low locality, are highly irregular, and have diverse energy and memory access patterns.
- Most performance bottlenecks are seen to arise due to synchronization and data sharing, occurring because of data dependent memory access patterns.
- Energy consumption bottlenecks reside in the on-chip network that incurs high traffic and contention.

Optimizations for future multicore architectures for graph analytics, such as using novel cache protocols and network optimizations are also discussed. *CRONO* is parametrized to run across varying number of threads and input instances, and therefore is a rigorous way for researchers to instrument their architectural schemes.

## Acknowledgment

The authors thank Professor Krishna Pattipati, Dr. Chris J Michael, Dr. Jim Hansen, and the reviewers for their useful feedback and comments. This research was partially supported by the National Science Foundation under Grant No. CCF-1452327. Masab Ahmad was supported by the U.S. Department of Education GAANN Fellowship.

## References

[1] D. S. Modha, R. Ananthanarayanan, S. K. Esser, A. Ndirango, A. J. Sherbondy, and R. Singh, "Cognitive computing," *Commun. ACM*, vol. 54, no. 8, pp. 62–71, Aug. 2011.

[2] "Google Is Building Its Own Self-Driving Car Prototypes," http://spectrum.ieee.org/cars-that-think/transportation/self-driving/google-is-building-selfdriving-car-prototypes, May, 2014.

[3] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic galois: On-demand, portable and parameterless," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 499–512.

[4] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[5] J. Mcauley and J. Leskovec, "Discovering social circles in ego networks," *ACM Trans. Knowl. Discov. Data*, vol. 8, no. 1, pp. 4:1–4:28, Feb. 2014.

[6] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 31–46.

[7] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '10. New York, NY, USA: ACM, 2010, pp. 147–158.

[8] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 185–195.

[9] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured gpu applications," in *2014 IEEE International Symposium on Workload Characterization*, October 2014.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, Oct 2009, pp. 44–54.

[11] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 78–88.

[12] 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 *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, pp. 979–990.

[13] 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.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[14] A. Buluç and J. R. Gilbert, "The combinatorial blas: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011.

[15] T. G. L. Graph500, "http://www.graph500.org/," 2010.

[16] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[17] in *Intel Xeon Phi Coprocessor High Performance Programming*, J. J. Reinders, Ed. Boston: Morgan Kaufmann, 2013.

[18] S. Bell and e. a. Edwards, "Tile64 - processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb 2008, pp. 88–598.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22nd Annual Int. Symph. on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 24–36.

[20] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[21] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 197–208.

[22] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, NJ, USA, 2014, pp. 572–583.

[23] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[24] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[25] "Darpa uhpc program baa, https://www.fbo.gov/spg/oda/darpa/cmo/darpa-baa-10-37/listing.html," March, 2010. [Online]. Available: https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-37/listing.html

[26] H. Lu, M. Halappanavar, A. Kalyanaraman, and S. Choudhury, "Parallel heuristics for scalable community detection," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 1374–1385.

[27] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 523–534, Jun. 2013.

[28] B. A. Hechtman and D. J. Sorin, "Exploring memory consistency for massively-threaded throughput-oriented processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 201–212.

[29] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, Nov 2012, pp. 141–151.

[30] M. Ahmad, K. Lakshminarasimhan, and O. Khan, "Efficient parallelization of path planning workload on single-chip shared-memory multicores," in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, Sept 2015.

[31] B. Gendron and T. G. Crainic, "Parallel branch-and-branch algorithms: Survey and synthesis," *Operations Research*, vol. 42, no. 6, pp. 1042–1066, 1994.

[32] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[33] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, Jan 2010, pp. 1–12.

[34] G. Kurian, C. Sun, C.-H. O. Chen, J. E. Miller, J. Michel, L. Wei, D. A. Antoniadis, L.-S. Peh, L. Kimerling, V. Stojanovic, and A. Agarwal, "Cross-layer energy and performance evaluation of a nanophotonic manycore processor system using real application workloads," *Parallel and Distributed Processing Symposium*, vol. 0, pp. 1117–1130, 2012.

[35] Intel, "http://ark.intel.com/products/80806/intel-core-i7-4790-processor-8m-cache-up-to-4_00-ghz," 2014.

[36] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on*, May 2012, pp. 201–210.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[38] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," 2006.

[39] M. A. Kinsy, M. H. Cho, T. Wen, E. Suh, M. van Dijk, and S. Devadas, "Application-aware deadlock-free oblivious routing," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 208–219.

[40] F. Hijaz, B. Kahne, P. Wilson, and O. Khan, "Efficient parallel packet processing using a shared memory many-core processor with hardware support to accelerate communication," in *Networking, Architecture, and Storage (NAS), 2015 10th IEEE International Conference on*, Aug 2015.

[41] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures," in *Proceedings of the 13th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 257–268.

[42] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 181–192.

[43] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[44] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 22:1–22:12.

[45] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. NY, USA: ACM, pp. 135–146.

[46] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. of the ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.