Scalable Graph Algorithms on Distributed UpDown Accelerators

1st Brian Wheatman
University of Chicago
USA
0009-0005-2018-4577

2nd Andrew A. Chien
University of Chicago and
Argonne National Laboratory
aachien@uchicago.edu

***Abstract*—Graphs are a fundamental tool for representing and analyzing relationships in data, with applications spanning social networks, the web, and the biological sciences. As the volume of data continues to grow, so too do the size and complexity of the graphs we must process. Extracting insights from these massive graphs requires scalable, high-performance algorithms.**

While many graph algorithms exhibit substantial parallelism, it is often extremely fine-grained—just a few instructions—making it difficult to exploit efficiently on conventional architectures. In this work, we demonstrate how to restructure graph algorithms to run efficiently on UpDown, a scalable, programmable accelerator designed specifically for fine-grained parallelism. UpDown enables scaling graph processing algorithms to systems with millions of concurrent hardware threads.

Through detailed simulation of UpDown systems with up to one million cores, we show that our approach achieves over 100,000× parallel speedup and absolute performance that outperforms the best reported distributed systems in terms of power efficiency. At the same time, our load-balanced approach retains strong performance at smaller scales, outperforming optimized multicore implementations under similar power budgets—all using a single, unified implementation.

***Index Terms*—Graph Algorithms, Parallel Computing**

I. Introduction

Due to their importance in gaining understanding from data, graph processing has been an active area of research for many years [1]–[3]. The specific data distributions vary with field and use case, but there are some properties that are widely applicable. These are that the graphs have small diameter and large skewness in degree distribution [4]. And in many cases, the algorithms running on the graphs can be broken down into phases of functional maps over the edges [2].

Graph processing is fundamentally a sparse computing problem, characterized by irregular memory access patterns and extreme load imbalance. The skew in vertex degrees leads to uneven computation across cores, and the dominance of indirect memory accesses reduces spatial and temporal locality. These challenges make it difficult to efficiently harness modern parallel hardware [5]–[7].

Over the years, graph processing frameworks have diverged into shared-memory, multicore and scale-out, distributed systems. Distributed solutions scale to massive graphs but often spend a disproportionate amount of time on communication rather than computation. In contrast, multicore solutions can be far more efficient, but are limited by the capacity and bandwidth of a single node. This raises the central question:

Can we design a system that combines the efficiency of multicore approaches with the scalability of distributed architectures?

In this work, we explore the proposition that multicore graph programming abstractions can be scaled to hardware with million-fold physical parallelism. If successful, this would unlock new levels of performance, at levels of programming effort similar today. We design a system for the proposed UpDown architecture [8]–[11], a scalable fine-grained graph computing system developed in IARPA's AGILE program [12] For more details on UpDown see Section II.

We find that the critical challenge for extreme scaling is excellent load-balance, to manage the irregularity found in real-world graphs. Multicore runtime scheduling approaches do not scale to millions of cores. So, our primary focus is addressing efficient, balanced work distribution to millions of cores to enable scalable performance on skewed graphs. We present a general self load balancing edge map algorithm that exploits UpDown's efficient hardware mechanisms. We implement this algorithm for three different fundamental graph algorithms of Breadth First Search, PageRank and Connected Components and find that we achieve both good scalability and absolute performance. These algorithms were chosen as classic well studied problems that have been used in benchmarking of many graph systems in the past [3], [13]–[17].

Our algorithm achieves high performance and good scaling across a wide range of graphs. while also being more power efficient. It is able to achieve consistent high performance across system scales using the same code.

II. UpDown System and Architecture

The UpDown system is a co-designed scalable graph supercomputer [8], [10], [11], [18]. The UpDown project also includes development of programming tools [19], [20] that have been used to demonstrate excellent graph computing performance on a range of applications [21], [22] Key aspects of the system's capabilities include: global address space and high communication performance (efficient 64-byte messages,

Research supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082.

low-latency of 0.5 microseconds) across 16,384 computing nodes. The system employs the Polarstar topology a diameter-3 direct network based on expander graphs [23], providing a 4.4 terabytes/second injection per node, and 32 petabytes bisection bandwidth. The machine achieves high memory bandwidth (153 petabytes/second system, 9.4 terabytes/second node), and high parallelism (2,048 lanes/node, 33M lanes overall).These capabilities enable UpDown to directly exploit vertex and edge parallelism in algorithms [20], [22], and to write programs that assume a nearly flat memory space (performance). The global system view is depicted in Figure 2. UpDown has been designed as part of IARPA AGILE program, and ambitious effort to create new building blocks with breakthrough capability for graph computing [12].

A. Node-level Design and Fine-grained Parallelism

Each UpDown node has 1 CPU, 8 HBM2e DRAM stacks and 32 UpDown accelerators. We focus on the accelerators because they provide the performance on the benchmarks studied. An UpDown accelerator has 64 MIMD lanes, each running at 2 Ghz. Each lane has 128 hardware threads, with only a single thread executing at a time (see Figure 1). There are no data caches, but each lane has a 64KB scratchpad that can be accessed in a single cycle. Each of these lanes can directly access global system DRAM (16K nodes, 512GB each) with a local node latency of 150 ns and a remote memory latency of 1.1 microseconds.

The event queuing and scheduling hardware, combined with multiple thread contexts, enables UpDown lanes to switch between threads with low-cost (1-cycle). This, and numerous other features enables UpDown lanes to execute fine-grained parallelism, events with 10-100 instructions, efficiently. This allows direct exploitation of edge and vertex parallelism.

B. UpDown from a Software Point-of-View

The UpDown system is entirely event-driven with FIFO scheduling at each lane, this enables efficient zero-cycle event scheduling. Events can come from other lanes, memory requests, or the TOP (CPU). For example, a DRAM memory request becomes an event like anything else, and will return by invoking the calling thread with the given call back [10]. This allows for extremely high levels of memory parallelism since a single thread can issue many memory requests while waiting on the first one and process them as they are received.

UpDown provides a global address space with flexible address mapping [11], but UpDown hardware does not provide any global scheduling or balancing of events. This leads to several challenges for scalable performance, particularly on irregular graphs. The major one is without a scheduler the work must be explicitly mapped onto and scheduled on the different lanes and any uneven distribution limits the scalability of the entire program. Directly, scalable performance requires load balance across 33 million lanes! The second is that the number of threads in each lane is directly limited by the hardware (128) and thus cannot vary with the dataset.

C. Example: Normalizing a vector on UpDown

To help the explanation of the algorithms in the later section we describe how to program UpDown for the task of summing up a simple array of integers. In this task we will want to use many worker threads to help parallelize the sum.

We start by assigning a root worker which is responsible for summing up the entire vector and has many lanes to process the data. This root worker can then divide the work into a set of tasks which then must be assigned explicitly to individual lanes to complete. This process continues recursively until either we are out of lanes to assign work to, or we are out of data to process. Then those tasks at the bottom actual compute the sum and return the results back up the computation tree.

In this case Suppose we have 512 nodes for a total of 1,048,576 lanes. Lane 1 is the root and responsible for summing up the entire vector. We can then assign the first half of the vector to a different thread on lane 1 and the second half to lane 524,288. This splitting takes only a few cycles, but sending work to another lane on a different node takes about 0.5 μs. If a lane does not have any other lanes to send work to it can start multiple threads on itself to actually perform the DRAM reads. Each of these reads also takes 1 μs (round-trip). Once these reads are done each thread sums up its values and returns the values to its parent. When each node in the tree gets both children's results, it can also return to its own parent until we get back to the root.

The design of the system allows for each lane to have many outstanding read requests. This means while the latency is about 1 μs, the throughput is much higher. Each lane can have over 100 threads running. Also, because the memory requests are split transaction even a single thread can send many memory requests and then start processing them only when they come back.

We must ensure that we do not ever schedule more than 128 threads on any individual lane, since it causes a software fault. However, for performance reasons we do want to schedule many different threads on any lane so that threads can make progress while waiting for child tasks or memory requests.

All together this means we can normalize a large vector across over 1 million lanes spending less than 30 microseconds on coordination on UpDown. This could be done even more efficiently by using a wider tree, or organizing the children lanes to only go across nodes when required. All together UpDown is a flexible programmable system in which we can build scalable graph algorithms on top of.

III. Load balanced EdgeMap algorithm

In this section we present a scalable EdgeMap algorithm for graph processing on the Updown system. EdgeMap is a foundational primitive used in many graph algorithms, responsible for applying a function across the outgoing edges of a subset of vertices. Prior work (e.g., [1]) has shown that a wide range of graph algorithms, including PageRank, Connected Components, and BFS, can be expressed using EdgeMap, and that for low-diameter graphs, the majority of the runtime is often spent in the dense variant.

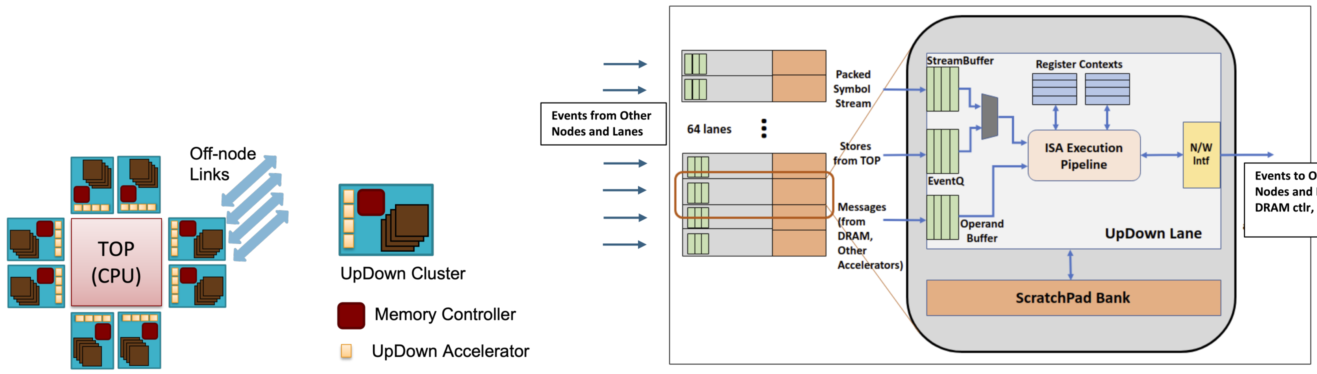

Fig. 1: Updown Node, Accelerators (each 64 lanes) Architecture

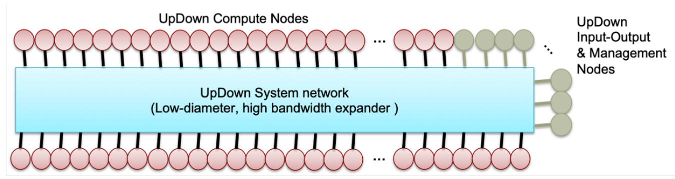
UpDown Compute Nodes
UpDown Input-Output & Management Nodes
UpDown System network
(Low-diameter, high bandwidth expander)

Fig. 2: Updown System: Nodes, Network

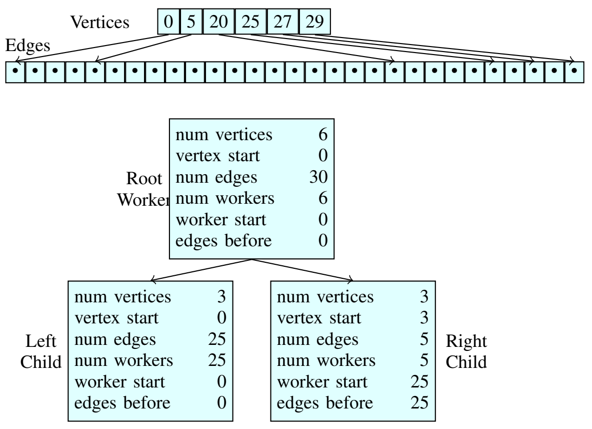

Fig. 3: A small graph and how the load balancing algorithm breaks it up into two components with proportional numbers of workers.

Our primary challenge is to implement a **scalable**, **load-balanced** version of EdgeMap suitable for UpDown's architecture, which lacks any global runtime or scheduler. Instead, all work must be explicitly mapped and scheduled across millions of hardware lanes. Our goal is to ensure that each lane receives a proportional share of the total work, enabling effective scaling even on highly skewed graphs. Any lane with an uneven distribution of work delays the entire computation.

This differs from many existing graph processing frameworks which rely on dynamic load balancing mechanisms such as work stealing [24], [25] . These techniques work well at small to moderate core counts, but have never been shown to scale to the million-fold parallelism required for UpDown.

To achieve this equal partitioning of any graph, we introduce a recursive EdgeMap implementation that partitions both the vertex set and the available lanes hierarchically, assigning subproblems in proportion to the amount of edge work. At the base level, each worker processes its assigned range directly.

Recursion: The recursive decomposition is shown in Figure 3 and works as follows. We begin with the entire graph stored in compressed sparse row (CSR) format and a single root worker responsible for processing the whole graph. As long as multiple workers and multiple vertices remain, we recursively divide the vertex set and assign subsets to child workers.

At each step of the recursion, we are given a range of vertices, the associated edge array, and a set of workers (lanes) to perform the computation. To determine how to split the work, we select a midpoint in the vertex array and compute its offset into the edge array, i.e., the number of edges belonging to the first half of the vertex range. This provides a proxy for the amount of work associated with each half.

For example, in Figure 3, the midpoint splits the edge array at offset 25 out of 30 total edges. This means the left half contains 5/6 of the edge work, so we assign it 5/6 of the available workers. We then recursively invoke the algorithm on each subrange, passing the appropriate subset of vertices and workers to each child. To maintain correctness and proportionality deeper in the recursion tree, we track two quantities: the number of edges before the current subproblem (to compute edge offsets correctly), and the index of the first worker assigned to this subproblem (to map output to global lane IDs correctly).

Base Cases The recursion terminates two different ways.

Single Worker Remaining: If only one worker is assigned to a subproblem, it processes the vertex and edge range directly. It iterates over the vertices in its range, and for each, iterates over the outgoing edges and applies the update function.

Single Vertex Remaining: If there is only one vertex to process but multiple workers available, we partition that vertex's edge list evenly among the workers. Since all edges are assumed to require similar work, this division can be done uniformly. This edge-level division is also performed recursively, although no proportional worker allocation is needed in this case.

Practical Details for an Efficient Implementation

While the recursive partitioning strategy described above provides a scalable and load-balanced theoretical foundation, achieving high performance in practice requires several implementation refinements. These optimizations address real-world overheads associated with memory access, tree depth, and coordination, and their performance impact can vary depending on the algorithm, input graph, and system size.

Wider Splitting High in the Work Tree: The theoretical algorithm splits the vertex range in half at each level, resulting in a binary tree of subproblems. However, this approach creates a deep tree with significant overhead, especially at the top levels where each step requires both a remote memory read and a remote task dispatch. To reduce tree height and associated overhead, we implement wider splits. Specifically, we split into eight children instead of two, which reduces the work-tree height three-fold (and cost of splitting) by three. Wider splits slightly increase the cost of each split operation, but this is outweighed by the reduction in depth. Thus we transition back to splitting into two children (within a node).

Work Tree Reuse Across Iterations: Many graph algorithms perform multiple rounds of edge processing across iterations. Rather than rebuild the recursive work tree each time, we cache and reuse it across iterations. This avoids a round trip to a remote memory location for each level of the tree. While the savings are small when the total work is large relative to the number of lanes, the savings become substantial as tree height grows with larger datasets and machines.

Early Exit Within Edge Processing: Some algorithms can short-circuit edge traversal once a condition is met. For instance, when searching for any unvisited neighbor, it is wasteful to scan the entire adjacency list after one such neighbor is found. We exploit this by allowing early exits during edge iteration.

Dynamic Pruning of the Work Tree: In many graph algorithms, vertices can become "inactive" once they have converged or completed their role. When this occurs, the corresponding leaf in the work tree can prune itself, notifying its parent that it no longer needs processing. Internal nodes with only a single remaining child can likewise be removed, passing their child up to the next level. This pruning mechanism reduces both vertex work and tree height in subsequent iterations. Figure 4 illustrates how pruning subtrees can simplify the work structure and reduce overhead. This dynamic shrinking of the work tree ensures continued efficiency even when the active vertex set becomes small-unlike many pull-based algorithms that must still scan the entire graph.

Pruning the work tree works together with reusing the work tree so that later iterations can iterate over increasingly small portions of the graph. This optimization can only be done when it can be known that a vertex is done being processed and will not need to be written to again.

Overlapping Latency via Multiple Workers per Lane: Each UpDown lane provides 128 hardware threads, but many operations, particularly memory access, incur latency due to remote fetches. To mask this latency and improve utilization, we schedule multiple lightweight workers per thread (typically up to 32), allowing one worker to compute while others are waiting on memory. This strategy effectively pipelines work and leverages the split-transaction memory system of UpDown to achieve greater parallelism and throughput.

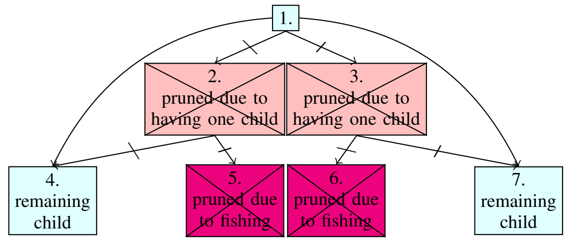

Fig. 4: This figure shows the benefits of pruning. In this example two nodes (5 and 6) finished processing. This allowed both them and their parents to be pruned with their siblings passed up to their grandparents which ends up reducing the height of the tree which can substantially improve the performance of later iterations.

IV. Example Algorithms

We describe several representative graph algorithms and describe their implementation using our load-balanced EdgeMap framework. For each, we highlight adaptations required to work within the UpDown model and examine how our various implementation optimizations—such as pruning, early exit, and work tree reuse—affect performance and scalability.

A. PageRank (PR)

PageRank is a classical algorithm for measuring the relative importance of vertices in a graph. Originally designed for ranking web pages, it operates by simulating a random walk: in each iteration, a node distributes its value evenly across its outgoing edges, accumulating incoming contributions to update its score. Over time, nodes that are highly connected or frequently reached tend to accumulate larger scores.

In the version we implement, each vertex starts with an initial score of $1/|V|$. In each iteration, every vertex computes a new score as the sum of contributions from its in-neighbors, where each neighbor contributes its current score divided by its out-degree. The algorithm continues either for a fixed number of iterations or until convergence—defined here as no vertex changing its score by more than $1/|V|$.

This algorithm maps naturally to our load-balanced EdgeMap. In each iteration, EdgeMap applies a function over all active vertices, distributing their values across outgoing edges. Because all vertices remain active until convergence, pruning is not used in PageRank. PageRank benefits from work-tree reuse across iterations and uses a reduction over the tree to track the global maximum score change, enabling early termination.

Implementation Details: One subtlety arises from the fact that score updates are performed asynchronously—there is no global sequencing between when different vertices read and write scores. To ensure correctness, each vertex maintains both its current and previous score, and the iteration number the values were written in. When a vertex reads a neighbor's score, it examines the iteration metadata to determine which score value is valid for the current update.

B. Connected Components (CC)

Connected Components is a fundamental graph algorithm that identifies the distinct connected subgraphs within an undirected graph. The algorithm assigns a label to each vertex such that two vertices share the same label if and only if there is a path between them.

The algorithm first assigns each vertex a unique label equal to its own vertex ID. In each iteration, every vertex examines its neighbor's labels and updates its own label to the minimum label found. Over successive iterations, labels propagate through the graph, and all vertices in a connected component eventually adopt the same minimum label.

We implement Connected Components using our load-balanced EdgeMap framework, where each vertex reduces labels over its neighborhood. Additionally, as the vertices compute their new label, we apply a min-reduction over these labels written, tracking the global minimum value propagated via the work tree. If a vertex is assigned this global minimum, it can never change again, since no smaller label exists. This allows us to prune such vertices from the work tree in future iterations. Moreover, during edge traversal, if a vertex encounters the minimum label in a neighbor, it can early exit the scan of it's adjacency list since no smaller value could be found.

This pruning step proves valuable in disconnected graphs. When the disconnected portion of the graph, with the smallest id has been entirely labeled it can be detected and all those vertices and be pruned from the work tree.

Implementation Details: As in PageRank, updates are asynchronous: a vertex reading a neighbor's label may see either the current or previous iteration's value. However, unlike PageRank, this nondeterminism is harmless in Connected Components. If a vertex reads a newer value early, it effectively skips an iteration. This only improves performance and does not affect correctness, as the algorithm still converges to the same final labeling.

C. Breadth First Search (BFS)

Breadth-First Search (BFS) is a classic graph traversal algorithm that computes the minimum distance from a given source vertex to all reachable vertices. BFS proceeds in rounds: the source vertex is initialized with distance 0, while all other vertices start with distance ∞. In each round r, all vertices at distance r propagate updates to their neighbors, setting them to distance $r + 1$ if their current distance is ∞.

There are two standard approaches for parallel BFS [26]:

Push-based: Track the active frontier (vertices at distance r), and in the next round, have these vertices push updates to their neighbors. While this is efficient for small frontiers, it requires atomic updates to avoid race conditions when multiple vertices attempt to update the same neighbor concurrently.

Pull-based: Iterate over all vertices, and for each unvisited vertex (distance ∞), check whether any of its neighbors has its distance set. This approach avoids atomic writes—since each vertex only updates itself—but becomes inefficient when the active set is small, as it requires scanning the entire graph.

In our implementation, we begin in push mode, which is more efficient when the active set is small (e.g., early in the traversal), and transition to pull mode when the frontier becomes large. Pull mode remains efficient in later rounds due to pruning, which avoids redundant computation as the active set shrinks.

Pull Mode: Pull mode directly maps to our load-balanced EdgeMap. Each iteration, we process all vertices whose distance is still ∞, scanning their neighbors for any vertex whose distance has been set. If such a neighbor is found, we set the vertex's distance to the next level and prune it from future iterations. Since we only need to find one matching neighbor, we enable both early exit and pruning within the work tree.

a) Push Mode: Push mode is more complex due to the need for atomic updates to neighbors. Additionally, the EdgeMap in this case operates only over a subset of the graph (the active frontier), requiring us to compute and track the outgoing edge counts (and the prefix sum) of the next active set to enable load-balanced scheduling in the next round.

We handle this using a three-phase strategy, inserting synchronization points between phases but maintaining the same work tree structure throughout a single iteration. We also prune the tree between phases to eliminate excess work.

Phase 1 (Reservation): Each vertex in the active set traverses its outgoing edges and attempts to reserve its neighbors by writing its own ID into a reservation buffer. Because multiple neighbors may try to reserve the same vertex, the final reservation value will be the last written vertex ID — exactly one writer wins per target vertex.

Phase 2 (Prefix Sum and Filtering): Each vertex in the active set re-traverses its neighbors to check if it successfully made any reservations. If it reserved none, it can be pruned. Otherwise, it computes two prefix sums: one over the number of successful reservations (determining the size of the next active set) and one over its total outgoing degree (used to schedule work for the next iteration). At the end of the second phase we use the counts to allocate the next active set.

Phase 3 (Output Construction): The third pass uses the results from Phase 2 to write out the new frontier and edge metadata. The prefix sums allow parallel writes into preallocated arrays, enabling efficient construction of the next active set and its edge list for the following iteration.

This phased design preserves determinism and avoids atomic operations, while still benefiting from load-balanced execution across millions of hardware lanes.

V. Evaluation

We evaluate the performance and scalability of our load-balanced EdgeMap implementation on the UpDown architec-

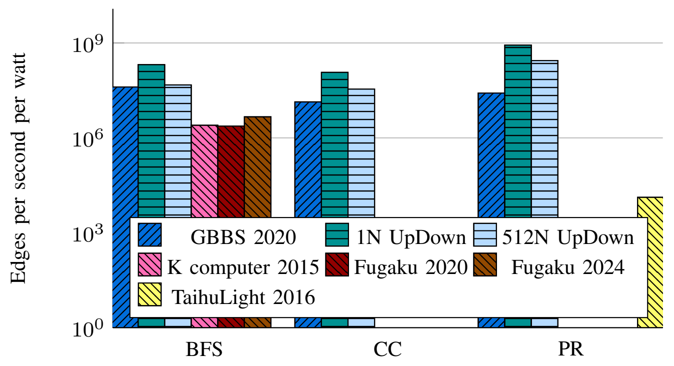

Fig. 5: ISO-power performance: The Load-balanced solution on Updown substantially outperforms other systems (normalized for power usage).

System	BFS	CC	PR	System TDP (Watts)
GBBS (2020)	40.5	13.7	26.1	660
1 Node UpDown	206.6	117.4	857.9	500
512 Node UpDown	47.0	34.9	277.9	0.29 MW
K computer (2015)	2.50	–	–	12.6 MW
Fugaku (2020)	2.37	–	–	30 MW
Fugaku (2024)	4.63	–	–	30 MW
TaihuLight (2016)	–	–	0.013	15 MW

TABLE I: ISO-power performance (Millions of edges per second per watt). The Load-balanced solution on UpDown substantially outperforms other systems (normalized for power).

ture, using a single copy of the graph in UpDown's global memory. Our results demonstrate strong scaling across graph types and system sizes, while delivering high energy efficiency. We compare against both state-of-the-art multicore (in-memory) solutions and distributed systems, highlighting how our method combines both efficiency and scalability.

Methodology

All results are generated using the Fastsim simulator—a cycle-accurate instruction-level simulator for UpDown. Fastsim emulates instruction execution across all hardware lanes and includes bandwidth and latency models for local DRAM accesses, remote DRAM accesses, and inter-node messaging. It was validated against a much slower GEM5 simulation model of 1-4 node UpDown configurations with full HBM (dramsim3) and interconnect simulations. [8]. We use Fastsim because its simulation rate of 80 million UpDown instructions per second and scalability to 512 nodes (1 million lanes) enables large-scale experiments.

All experiments are run across 5 different graphs real world and synthetic graphs [3], [27]. For comparisons we normalize all results using edges per second. That is to say how many edges does the graph have divided by the amount of time it took to run the algorithms. we make **ISO-power** comparisons, that is we normalize performance by estimated system TDP power measured in watts. This allows us to compare different types of systems with a wide range of scale. In all cases we make conservative comparisons, using the the fastest published results from any compared approach.

Graph	Vertices	Edges
RMAT s28 (RMAT28)	268M	4.2B
Erdos Renyi s28 (er)	268M	9.4B
soc-liveJournal (LJ)	4.8M	43.1M
com-orkut (orkut)	3.5M	117M
Twitter (twitter)	61.6M	1.2B

TABLE II: Graph Statistics

Normalized Power Efficiency (ISO-power)

Figure 5 shows the power-normalized throughput, measured in edges processed per second per watt, across PageRank, BFS, and Connected Components. Overall our approach is about 5x more power efficient than exsisting solutions. We compare our UpDown implementation to: GBBS [28], high-performance software designed for in-memory graph processing on multicore systems. K Computer [29] and Fugaku [30], [31], two of the most powerful distributed graph processing systems, both of which have topped the Graph500 benchmark. We also compare to the ShenTu results for PageRank on the TaihuLight system [32]. Table I include the detailed numerical results.

Despite its simplicity and energy efficiency, the GBBS framework does not scale beyond a single shared-memory machine (multicore). In contrast, distributed systems like Fugaku can scale out but suffer inefficiencies and complex programming.

Our load-balanced EdgeMap on UpDown achieves a compelling middle ground: matching or exceeding the performance of these systems while using far less power and maintaining clean programmability. Load-balanced EdgeMap on a single UpDown node outperforms GBBS on multicore by 5-34x (ISO-power), and while parallel scaling is not perfect, power-efficiency still exceeds the GBBS-multicore system at 512 nodes. Naturally performance is more than 500x higher at that point.

On BFS, for example, we match or exceed Fugaku's 2024 numbers at a fraction of the power budget. In fact, load-balanced EdgeMap on UpDown scales so well that at 512-nodes its power efficiency (and absolute performance) exceeds that of the other systems.

PageRank (PR): Figure 6a shows the scaling performance of PageRank. Our implementation shows nearly linear scaling across graph types up to 1 million lanes. At 2,048 lanes, we transition from a single-node to multi-node system, increasing the proportion of remote memory accesses. Despite this, we continue to scale—reaching a peak of over 16,000 GTEPS on large graphs. We outperform the GBBS baseline around 1,000 lanes (half a node), using approximately 2x less power.

We can also use PR to evaluate the benefit of the tree reuse optimization. In Table III we see the savings range from over 2x on the smaller graphs to about 15% on the largest graph.

Connected Components (CC): Figure 6b shows scaling results for Connected Components. As with PageRank, we

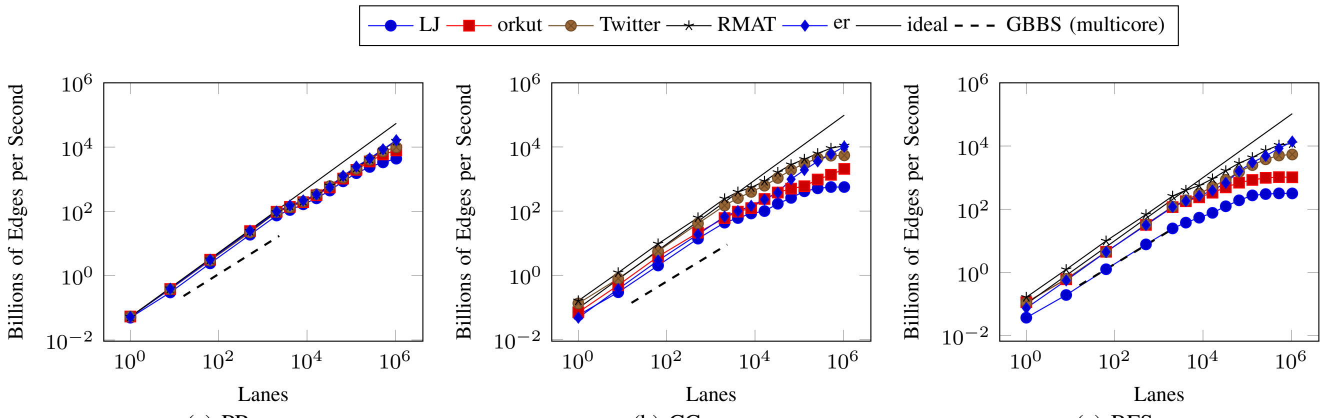

Fig. 6: Load-balanced EdgeMap algorithm on Updown. A high-performance multicore solution (GBBS) is included and aligned based on ISO-power. The studies are limited by simulation capabilities to Billions of edges and 512 UpDown nodes. With larger graphs we expect better scalability up to the full 16K node system.

TABLE III: Benefit of tree reuse in PR on 1 million lanes. On smaller graphs a larger portion of the work is coordination which tree reuse allows us to reduce.

Iteration	LJ	orkut	twitter	RMAT28	er
1st iteration cost (μs)	35	45	290	320	620
2nd iteration cost (μs)	15	25	230	255	525

TABLE IV: Benefit of tree pruning in CC on 1 million lanes.

Iteration	LJ	orkut	twitter	RMAT28	er
1st iteration cost (μs)	35	40	265	240	530
2nd iteration cost (μs)	15	20	55	75	365
3rd iteration cost (μs)	15	20	15	20	10

observe near-ideal scaling up to 512 nodes (1M lanes), achieving over 10,000 GTEPS on large inputs. The performance surpasses GBBS's absolute performance around 1000 lanes, with a 2X power advantage.

Connected Components particularly benefits from work tree pruning and early exit (Table IV). As vertices converge to their final labels, they are pruned from subsequent iterations.

Breadth First Search (BFS): Figure 6c shows BFS performance across graphs and scales. As described in Section IV-C, BFS uses a hybrid push/pull strategy. In early iterations, we use push mode for small frontiers; later, we switch to pull mode and leverage pruning to maintain efficiency.

We achieve up to 13,839 GTEPS on large graphs and observe continued scaling past the multi-node threshold. Our push-phase implementation avoids atomics via a reservation protocol, preserving determinism and reducing synchronization overhead. Compared to distributed BFS on K Computer and Fugaku, our solution is 10X more energy efficient (Table I). At the lower levels of performance reachable by the multicore solution, our is 5x faster at ISO-power.

VI. Summary and Future Work

Our scalable, load-balanced framework for graph algorithms on the UpDown architecture uses recursive work-partitioning strategy to achieve efficient parallel edge traversal, using 1 million parallel lanes. Additional practical optimizations including work tree reuse, dynamic pruning, and hybrid push/pull traversal, we achieve both high performance and energy efficiency.

Our results demonstrate both high-efficiency and strong scalability, outperformaning state-of-the-art multicore and distributed systems when normalized for power. Our results show that UpDown can bridge the gap between the simplicity of in-memory multicore systems and the scalability of large-scale distributed architectures. Implementation of PageRank, Connected Components, and BFS illustrates how a single, load-balanced EdgeMap abstraction can generalize across diverse algorithms and graph structures. We believe this approach provides a compelling path forward for future graph processing systems and accelerators.

While we focused on straightforward implementations of classic algorithms, future work could integrate more advanced techniques. For example, asynchronous Gauss–Seidel methods for PageRank have been shown to accelerate convergence compared to the classical method [33], and recent work on Fugaku demonstrated that tree-pruning strategies can significantly improve BFS performance [31]. Using such optimizations with our load-balanced EdgeMap framework could further enhance efficiency on UpDown.

Acknowledgement

We would like to thank the entire UpDown team, for their support and continued development of the UpDown architecture. In particular, we acknowledge Alexander Fell for help in understanding the UpDown hardware model and the supporting simulation and programming tools. His insights and support were instrumental throughout this work.

References

[1] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[2] B. Wheatman, X. Dong, Z. Shen, L. Dhulipala, J. Łacki, P. Pandey, and H. Xu, "Byo: A unified framework for benchmarking large-scale graph containers," *arXiv preprint arXiv:2405.11671*, 2024.

[3] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[4] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 1372–1385.

[5] Argonne, "The aurora exascale supercomputer," Argonne National Laboratory, https://www.alcf.anl.gov/aurora.

[6] ORNL, "The summit exascale supercomputer," Oak Ridge National Laboratory, https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.

[7] "5th Generation AMD EPYC™ Processors." [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/9005-series.html

[8] Andrew A Chien, et. al, "Updown: A supercomputer co-designed for scalable graph processing," *under review*, 2024, available from http://people.cs.uchicago.edu/ aachien/lssg/research/10x10/UpDown_System_Paper___TPDS_submittedv2.pdf.

[9] A. Rajasukumar, J. Su, Yuqing, Wang, T. Su, M. Nourian, J. M. M. Diaz, T. Zhang, J. Ding, W. Wang, Z. Zhang, M. Jeje, H. Hoffmann, Y. Li, and A. A. Chien, "Updown: Programmable fine-grained events for scalable performance on irregular applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.20773

[10] A. Rajasukumar, T. Zhang, R. Xu, and A. A. Chien, "Updown: A novel architecture for unlimited memory parallelism," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 61–77. [Online]. Available: https://doi.org/10.1145/3695794.3695801

[11] Y. Wang, S. Perarnau, and A. A. Chien, "Updown: Combining scalable address translation with locality control," in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1014–1024.

[12] "Advanced graphic intelligence logic computing environment," 2022, https://www.iarpa.gov/research-programs/agile.

[13] P. Dreher, C. Byun, C. Hill, V. Gadepally, B. Kuszmaul, and J. Kepner, "Pagerank pipeline benchmark: Proposal for a holistic system benchmark for big-data platforms," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2016, p. 929–937. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2016.89

[14] L. Dhulipala, J. Shi, T. Tseng, G. E. Blelloch, and J. Shun, "The graph based benchmark suite (gbbs)," in *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA'20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3398682.3399168

[15] B. Wheatman, R. Burns, and H. Xu, "Batch-parallel compressed sparse row: A locality-optimized dynamic-graph representation," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024, pp. 1–8.

[16] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7.

[17] B. Wheatman, R. Burns, A. Buluc, and H. Xu, "Cpma: An efficient batch-parallel compressed set without pointers," in *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2024, pp. 348–363.

[18] A. Rajasukumar, J. Su, Yuqing, Wang, T. Su, M. Nourian, J. M. M. Diaz, T. Zhang, J. Ding, W. Wang, Z. Zhang, M. Jeje, H. Hoffmann, Y. Li, and A. A. Chien, "Updown: Programmable fine-grained events for scalable performance on irregular applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.20773

[19] A. Fell, Y. Wang, T. Su, M. Nourian, W. Wang, J. M. Monsalve-Diaz, A. Rajasukumar, J. Su, R. K. Ruiqi Xu, D. F. G. Tianchi Zhang, Y. Li, H. Hoffmann, and A. A. Chien, "Kvmsr+udweave: Extreme-scaling with fine-grained parallelism on the updown graph supercomputer,," in *Proceedings of SC Parallel Applications Workshop, Alternatives To MPI+X*, November 2025.

[20] Yuqing Wang, Andronicus Rajasukumar, T. Su, M. Nourian, A. P. Jose M Monsalve Diaz, J. Ding, C. Colley, W. Wang, Y. Li, D. F. Gleich, H. Hoffmann, and A. A. Chien, "Efficiently exploiting irregular parallelism using keys at scale," in *Proceedings of Conference Workshop on Languages and Compilers for Parallel Computing*, Nov 2023.

[21] J. Su, A. Fell, D. F. Gleich, and A. A. Chien, "Scaling triangle counting and k-truss on the updown architecture," in *submitted for publication*, Jun. 2025.

[22] Y. Wang, C. Colley, B. Wheatman, J. Su, D. F. Gleich, and A. A. Chien, "How fast can graph computations go on fine-grained parallel architectures," 2025. [Online]. Available: https://arxiv.org/abs/2507.00949

[23] K. Lakhotia, L. Monroe, K. Isham, M. Besta, N. Blach, T. Hoefler, and F. Petrini, "Polarstar: Expanding the horizon of diameter-3 networks," in *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '24. ACM, Jun. 2024, p. 345–357. [Online]. Available: http://dx.doi.org/10.1145/3626183.3659975

[24] P. Nookala, P. Dinda, K. C. Hale, K. Chard, and I. Raicu, "Enabling extremely fine-grained parallelism via scalable concurrent queues on modern many-core architectures," in *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2021, pp. 1–8.

[25] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720–748, Sep. 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[26] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Scientific Programming*, vol. 21, no. 3-4, pp. 137–148, 2013.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[28] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," *ACM Transactions on Parallel Computing (TOPC)*, vol. 8, no. 1, pp. 1–70, 2021.

[29] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, "Efficient breadth-first search on massively parallel and distributed-memory machines," *Data Science and Engineering*, vol. 2, no. 1, pp. 22–35, 2017.

[30] M. Nakao, K. Ueno, K. Fujisawa, Y. Kodama, and M. Sato, "Performance of the supercomputer fugaku for breadth-first search in graph500 benchmark," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 372–390. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_20

[31] J. Arai, M. Nakao, Y. Inoue, K. Teranishi, K. Ueno, K. Yamamura, M. Sato, and K. Fujisawa, "Doubling graph traversal efficiency to 198 terateps on the supercomputer fugaku," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2024, pp. 1–14.

[32] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu *et al.*, "Shentu: processing multi-trillion edge graphs on millions of cores in seconds," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 706–716.

[33] D. Silvestre, J. Hespanha, and C. Silvestre, "A pagerank algorithm based on asynchronous gauss-seidel iterations," in *2018 Annual American Control Conference (ACC)*. IEEE, 2018, pp. 484–489.