UpDown: A Supercomputer Co-designed for Scalable Graph Processing

Andrew A. Chien*†, Jose M Monsalve-Diaz†, Charles Colley*, Jianru Ding*, Alexander Fell*, David F. Gleich§, Henry Hoffmann*, Moubarak Jeje‡, Rajat Khandelwal*, Yanjing Li*, Marziyeh Nourian*, Andronicus Rajasukumar*, Jiya Su*, Tianshuo Su*, Yuqing (Ivy) Wang*, Wenyi Wang*, Ruiqi Xu*, Tianchi Zhang*
*University of Chicago, †Argonne National Laboratory, ‡Tactical Computing Laboratories, ‡Purdue University
aachien@uchicago.edu

***Abstract*—Traditional supercomputers have focused on dense computation performance as exemplified by HPL. Graph processing applications differ with extreme irregularity (10^9 imbalance in skewed, real-world graphs) that produces unpredictable work, parallelism, memory access, and communication. Together, these make scalable performance and programming difficult.**

We describe the UpDown system architecture, co-designed for irregular graph computations. UpDown provides efficient fine-grained thread invocations ($\sim$ 10 instructions), direct messaging (no network interface card) for scalable local and global messaging, and split-transaction memory operations that enable extremely high memory bandwidth. Combined with architectural support for global addressing and an aggressive network design, these UpDown features enable direct exploitation of edge and vertex parallelism, using it to deliver breakthrough graph processing performance and programmability.

We evaluate the performance of the UpDown system using a challenging suite of graph applications (Pagerank, Breadth-first Search, Triangle Counting, Partial Match, etc). For a single-node, results show 100-fold performance advantage over multicore CPUs. Compared to today's fastest scalable parallel computers UpDown achieves 1000-fold performance increases. UpDown delivers these levels of performance with high-level programmability, these programs directly express vertex-edge parallelism which UpDown exploits directly in hardware.

***Index Terms*—parallelism, high-performance computing, graph analytics, graph algorithms, scalable, supercomputer**

I. Introduction

The success of scalable computing systems in scientific computing [1], [2] and cloud computing [3]–[5] has delivered dramatic progress. The application scale achievable has grown from teraflop/second rates in the 1990's to petaflop and exaflop rates in the 2020's. Perhaps just as important, the physical memory capacities of machines have grown equally fast, from terabytes to petabytes over the same period. Together, these capabilities enable dramatically large scientific and commercial applications of many types.

At the center of this progress is system architecture design driven by dense computations (MODSIM in scientific computing, exemplified by HPL [6] and the Top 500 [7])[1]. While laudable, this has left large classes of irregular, sparse applications experiencing much lower performance. Because of their importance, the community has created new computation benchmarking efforts focused on irregular applications such as high-performance conjugate gradient (HPCG) have arisen. To date, these benchmarks showcase how poorly current systems support irregular applications, delivering performance at only 1-2% of peak [9].

Exploiting the popular codesign paradigm, we pursue the research question – **what performance could be achieved with a machine co-designed for scalable graph processing?** That is, what levels of both efficiency and performance could be achieved if the system architecture was designed to support applications that are irregular in computation and parallelism, as well as communication.

Using a set of challenging graph benchmarks and challenging irregular, real-world graphs, we co-designed the UpDown system architecture. For irregular compute, UpDown supports fine-grained thread invocations to maximize parallelism and facilitate load imbalance. For irregular parallelism, UpDown provides inexpensive dynamic parallelism with direct messaging. These are a contrast to traditional systems and runtimes with much more expensive thread management . For irregular memory references, UpDown provides ample memory bandwidth with mechanisms to generate high memory level parallelism and eschews caches. For scalability to large graphs, UpDown provides an extremely low-latency, high-bandwidth network. Together, these features support scalability to 10,000's of nodes and extreme high performance for graph processing applications.

We describe the design of the UpDown system. First, we describe its mechanisms for fine-grained tasks, event-driven scheduling, and efficient communication amongst UpDown accelerators within a node. Second, we describe its global characteristics, notably, support for global addressing, latency tolerance, and memory parallelism, as well as the capabilities of its global system network. Next, we use a challenging set of benchmarks for evaluation, showing UpDown's achieved performance – first on a single node, then at scale with 512 nodes (1 million lanes). We drill down in several specific cases to give insight into how UpDown's novel architecture mechanisms are effective. Finally, we close with a discussion of how UpDown enables vertex-edge programmability, and how the balance of UpDown compares to conventional super-

[1]At a smaller scale, machine learning training has similar computational properties [8]

computers, notably per-node communication bandwidth, and how power is expended in the system.

Specific contributions of the paper include:

- Design of UpDown accelerator architecture whose fine-grained parallelism support enables efficient 10 instruction invocations with 1-cycle event-driven scheduling, and thread-management.
- Direct messaging mechanisms (no network interfaces), achieving message rates of 1 giga-msgs/s per lane and nearly 1 tera-msgs/s per node (1000x existing systems), enabling uniform communication for parallelism within and across nodes.
- Illustrate UpDown's explicit split-transaction DRAM access for high memory parallelism and latency-tolerant global memory access under software control.
- Evaluation of a 1-node UpDown that shows 100-fold speedup vs a multicore CPU (20 cores, GraphZero).
- Evaluation of a 512-node UpDown system (1 million lanes) showing it delivers 500-10,000-fold speedups vs conventional supercomputers on challenging graph applications using RMAT and other difficult skewed graphs.
- Programming examples showing UpDown's mechanisms are flexible and general, and UpDown's promising for a broader class of applications.

The rest of the paper is organized as follows. In Section II, we cover key background concepts and research. Next, in Section III, we describe the key challenges in scalable graph processing and our approach. After that, in Section IV, we introduce the node and system architecture for UpDown. Evaluation of this architecture on a variety of kernels and full-scale applications is presented in Section VI. In Section VIII, we discuss key related work. Finally, in Section IX, we summarize and discuss directions for future work.

II. Background

A. Conventional Parallel Architecture and Software Models

For three decades, scalable high-performance systems have been constructed from high-volume compute elements – CPUs and recently GPUs. This approach leverages low-cost high-performance components [7], [10]–[12]. These components depend on tightly-coupled, memory hierarchies and data-reuse to achieve efficient, high performance. Their assembly into "massively-parallel" systems (MPP), using the Message Passing Interface (MPI) and distributed memory computing delivered the dramatic increases in computing performance from teraflops to peta/exa flops.

The MPP approach has important performance implications. Local memory hierarchies require large chunks of computation with high data-reuse for efficiency as well documented by McCalpin [13]. To match this, applications are designed for coarse-grained parallelism (and typically weak-scaling). System networks are connected on the far side of the memory hierarchies (at the input/output bus, designed for block operations), relegating communication performance to high overhead, coarse-grained, and often high latency. In recent systems, system interconnect bandwidths are 10-100x lower than local memory bandwidths [12], [14]. These characteristics make exploiting fine-grained parallelism inefficient, and produces poor strong scaling. This is because much of the natural fine-grained parallelism in graph computations inaccessible, particularly for irregular graphs. To highlight this point, our performance studies emphasize strong scaling to show UpDown's flexible computing power for all sorts of graph problems.

B. Graph Applications and Requirements

Graph processing includes a rich class of algorithms from analytics to pattern mining and graph learning. These algorithms span a diversity of computations including but not limited to PageRank [15], Triangle Counting [16], Jaccard Similarity Coefficients [17], etc. Software frameworks have been designed for large scale graph processing in [4] [18] [19] that adopt vertex-centric or edge-centric iterative parallel paradigms or both [20] to describe parallelism. Because the amount of computation per vertex or edge is usually quite small, this is very fine-grained parallelism.

When real-world (eg highly-skewed) graphs are considered, the parallelism is both fine-grained and irregular, posing challenges to load balance and demanding irregular data movement in a parallel machine (eg. each edge potentially connects to a vertex "across" the machine). As a result, the parallelism in formulation, however, often doesn't translate to increased performance on traditional multicore CPUs, requiring extensive data structure design to get limited performance scaling to tens of cores [21]. These challenges have also stimulated interest in specialized hardware for graph processing [22].

Another consequence of the complex performance dynamics has been the emergence of a rich algorithmic and data structure literature for graphs [21] which can be exploited in scalable shared memory machines, but due to programming difficulty, have been largely unexploited in scalable parallel machines where MPI and distributed memory are the dominant programming environment [23].

C. AGILE Program and the UpDown System

The UpDown system design is one of six research efforts launched by the US Government under the Advanced Graphic Intelligence Logical Computing Environment program (AGILE) in September 2022 [24]. The ambition of the program is to create innovative, energy-efficient, and reliable computer architectures for scalable data analysis. AGILE focuses on sparse, time-varying, real-world graphs addressing fundamental data movement challenges at system-scale.

Within AGILE, the UpDown system architecture, and a full hardware design has been developed. Elements completed include a full system specification, instruction set architecture – aka software interface, full hardware system model, and scalable simulation tools.

III. Problem and Approach

A. Problem

Today's scalable parallel machines excel at dense regular computations [1], [2], [6] by exploiting their structured work and scalable data reuse. This makes hardware caches an

essential feature, and these machines depend on the ability increase grain size to amortize communication and synchronization overheads. This is a direct consequence of exploiting volume computing components. Unfortunately, for irregular computations on skewed graphs with low data-reuse, caches do not work well, and worse can inhibit performance [25].

Irregular applications on skewed graphs exhibit poor data reuse, as well as irregular computation structure, memory reference, and parallelism. The irregular compute structure produces poor load balance (see Figure 1 that illustrates skew of 13,700x), and irregular memory reference produces poor cache performance. But there is plentiful fine-grained (100 instruction) parallelism. These present system architecture challenges for scalability and efficient hardware utilization.

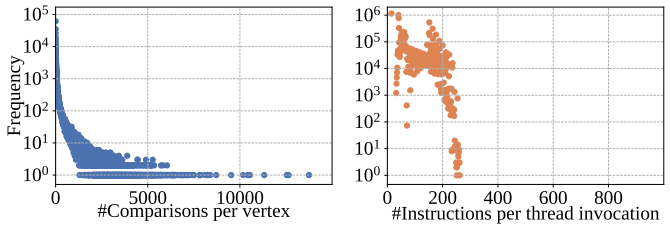


Fig. 1: Graph computing has per-vertex parallelism with skew of 13.7k-fold (left: comparisons per vertex – TC on Yo dataset). UpDown decomposes the work into smaller thread invocations with lower-skew (20-fold) (right: instructions per invocation – TC on Yo dataset)

While graph applications on conventional hardware can do little to overcome inefficiencies due to poor data reuse, they attempt to mitigate compute irregularity by aggregating plentiful vertex and edge parallelism [20], to increase execution efficiency. Larger grain sizes (millions of instructions) enable amortization of high scheduling, communication, and synchronization costs. However, these larger chunks sacrifice 1000-fold parallelism, limiting scalability [26]

Results show the best software frameworks, after trading away massive-parallelism, deliver poor scaling — 50% efficiency (Triangle counting) and even worse 25% efficiency (BFS) at 80 threads with hyperthreading (Figure 2).

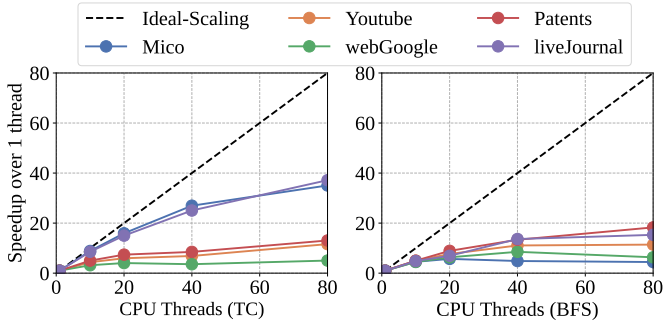


Fig. 2: Speedup vs Threads on Intel Xeon (Skylake) (Dual Socket, hyperthreaded) @ 2GHz on Triangle Counting [27] (left) and BFS [18] (right)

Thus the research problem is **how to design of an architecture to exploit the vertex/edge fine-grained, irregular parallelism directly?** And to do so for applications with low data reuse. Such a system could exploit much higher levels of parallelism for strong scaling, and it could be programmed naturally for graphs, with direct expression of vertex/edge parallelism and sophisticated algorithms [28]–[30]. To meet this challenge, the design must efficiently:

- Exploit fine-grained tasks efficiently to maximize parallelism and reduce load imbalance
- Exploit fine-grained communication to tolerate memory reference irregularity and latency
- Schedule tasks efficiently, in response to varied computation and memory events
- Scale to million-fold parallelism and enable programming a vertex-edge model at reasonable effort

These ambitious properties are the co-design challenge for scalable graph computing.

B. Approach

To meet these challenges, the UpDown system is an integrated system that combines novel compute accelerators for fine-grained task parallelism and memory latency tolerance. The system is designed for a global address space and scalable memory latency tolerance to 10,000's of nodes. Each node also includes conventional CPU cores for generality. Specific elements of the architectural approach include:

1) 1-cycle event-driven scheduling for efficient fine-grained parallelism (10 instruction thread invocations), specifically hardware mechanisms for thread creation, scheduling, and management
2) High hardware parallelism and memory latency tolerance (thousand to million-fold parallelism and latency of thousands of cycles)
3) 1-instruction communication and synchronization for up to 8-word messages and scaling up with computation capability
4) Extreme global communication (4.4 TB/s/node) and memory bandwidth (10 TB/s/node), both $>$ 100x greater than current machines
5) For programmability, general mechanisms that are wholly software controlled, enabling adaptation to application parallelism and data structure

Efficient fine-grained parallel computation, communication, and synchronization enable direct exploitation of parallelism at the level of single vertex or edge. This maximizes potential application parallelism and reduces programming effort. It also mitigates the impact of irregular computation, parallelism, and communication.

The architecture support for this scalable parallelism is lightweight threads with little explicit (registers) or implicit state (no caches). UpDown provides high memory and communication bandwidth to manage low data-reuse, scaling bandwidth with computing capability, enabled by HBM and in-package optics. Finally, all of the architectural mechanisms are generally, not tied to graph algorithms or data structures, ensuring broad applicability.

IV. UpDown System Architecture

The UpDown system is a scalable parallel computer, consisting of many nodes connected by a scalable direct network (see Figure 3). We describe the system "inside-out", beginning

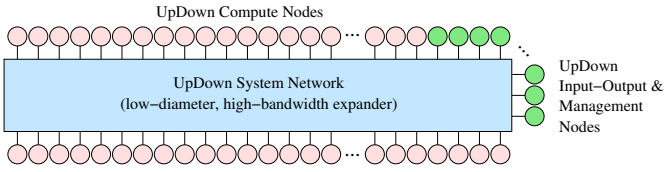


Fig. 3: The UpDown System consists of compute, input/output and management nodes. All are connected by the high performance system network.

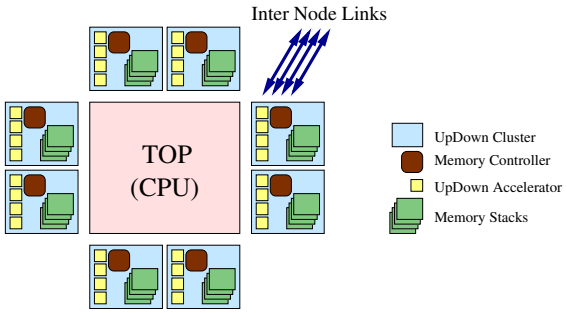


Fig. 4: Each Updown node includes a 96-core TOP CPU, a set of UpDown accelerators, HBM DRAM memory stacks, and system network connections.

with the UpDown node and accelerator (Section IV-B) that enables fine-grained parallelism and efficient communication. Then we proceed to the UpDown system network that provides scalable communication and bisection (Section IV-C).

We conclude with a discussion of the UpDown's global address space and high memory bandwidth (see Section IV-D), that have been described extensively elsewhere [31], [32].

A. *UpDown Node Design*

The key element of an UpDown node is a set of 32 UpDown accelerators, each consisting of 64 lanes. A 96-core CPU with cache hierarchy is used for computations with high data reuse, coherence is only within the node. The node also includes 8 HBM memory stacks (HBM3e in current design), and connection to the high speed system network. These parts are illustrated in Figure 4.

The UpDown accelerators are the key architectural innovation. With no caches, they provide high-performance on low-reuse computations (streaming for graph and sparse) by employing simple event-driven, in-order execution that also provide high power efficiency and computation rate. Between scratchpad memory and DRAM (globally across nodes), data movement and consistency are software managed.

Each UpDown node is a single package, similar to a CPU or GPU chip. It's about one-eighth the size and power of node in Argonne's new Aurora supercomputer which has two CPUs and 6 GPUs [12].

B. *Fine-grained Parallelism: the UpDown Accelerator*

Each UpDown accelerator consists of 64 UpDown lanes, designed for efficient fine-grained parallelism. Each lane executes independently (the accelerator is MIMD), and includes mechanisms for efficient response to hardware asynchrony and variable latency (eg memory access, interlane and cross-machine communication, etc.). Each lane has 128-way hardware multithreading and a local scratchpad (64KB), which

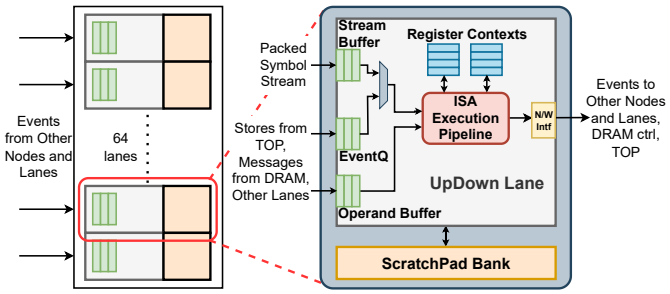


Fig. 5: Fine-grained parallelism is supported by UpDown *Lanes*, with 1-cycle, event-driven scheduling, 128-way multithreading, and ISA split-transaction DRAM access. 10-instruction thread invocations are efficient. Events are seamlessly routed inter- and intra-node (Figure 3 and 4).

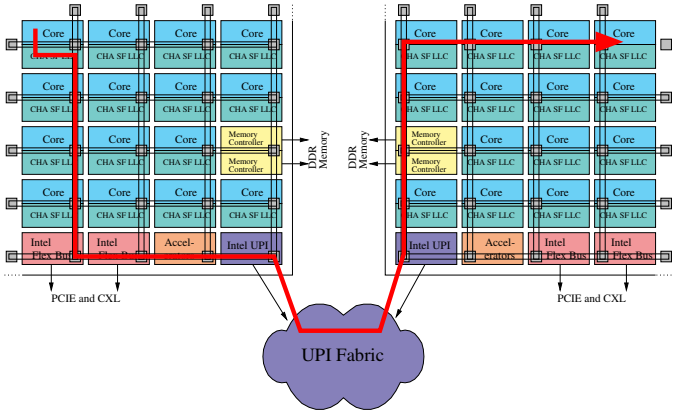


Fig. 6: Inter-thread events exercise a complex memory hierarchy and NUMA interchip coherence (x86 2-socket multicore). Result: 0.5µs latency.

can be pooled amongst lanes if desirable for software. The UpDown accelerator is depicted in Figure 5.

UpDown Lanes differ significantly from conventional RISC cores [33], [34]. For example, each event in a UpDown system creates or schedules a thread context when it arrives at a lane. Summarizing the novel capabilities and architecture features:

- **Low Latency Events:** Hardware event queues, and hardware event-driven thread scheduling. Enables 1-cycle latency.
- **Effective Short Thread Activations:** Direct access to event operands, hardware-supported thread create;schedule;terminate. Direct ISA messaging for 1-8 words. Enables 10-instruction invocations to achieve high core utilization.
- **Direct ISA split-transaction DRAM access**: One to eight-word load/store operations to DRAM, asynchronous return enables flexible, *software* memory-latency tolerance (1000 cycles).
- **General Acceleration:** Programmable Events and Flexible Threads.

Overall, UpDown's mechanisms enable it to act as a low-latency event processor, servicing events that trigger fine-grained parallelism. Fast thread creation/scheduling combined with the efficient access to operands and messaging, makes remarkably short events practical. Examples such as a ping-pong counter, stack, value filter, atomic access, queueing lock, etc. can be implemented with a few instructions.

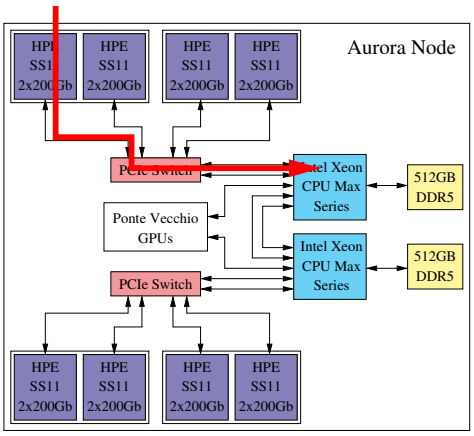


Fig. 7: Communication events cross PCIe, bridges, and notify a core with interrupts or polling. (Aurora Supercomputer) Result: 3µs latency.

1) Low-Latency Events: Traditional HPC systems notify CPU cores of communication events with interrupts, and interact over PCIe and bridges producing 3µs latencies in Perlmutter (see Figure 7 [14]); about 10,000 processor cycles. Intra-node event notifications in efficient software threads (e.g., an OpenMP runtime) synchronize through cache-coherent operations spanning multiple CPU chips and UPI/HT interconnects. With L3 latencies of 30-50 cycles, an ownership transaction can take 0.5µs, nearly 1000 processor cycles (see Figure 6). In contrast, UpDown handles events with 1-cycle latency, creating or scheduling threads in response.

UpDown achieves 1-cycle latencies by sending events directly to lanes, and using simple FIFO queuing. Once there, computation is triggered using a hardware EventQ and Operand Buffer integrated into the lane (as shown in Figure 5). The EventQ-head's value selects the thread state by indexing the register contexts. [2]

2) Efficient Short Thread Activations: UpDown's radically lower event latency (1-cycle) exposes the need to make the triggered computation (a thread activation) efficient. Ideally, a few instructions would be able to perform a useful task. A key UpDown design is **direct-access** to event operands. Register names can be used by the currrent thread to denote operands in the current event. In UpDown, 8 register names are used for message operand values. As a result the program can access the values directly, no copying (load instructions) is required. enabling efficient, short threads. As shown in Figure 8, UpDown's direct-access design can reduce instructions for a simple filter operation by two-thirds (67%). In prior aggressive hardware support for messaging (left), programs copied data from a hardware message buffer into registers before processing. In Section IV-B4, another example shows UpDown needs can do address calculation for DRAM access in just 5 instructions because of this direct operand (op0) access. All of these examples use **direct messaging**, UpDown instructions that allow an event/message to be sent with a single instruction.

3) Explicit Split-transaction DRAM access: UpDown programs issue DRAM accesses as in Figure 11, and then

[2]We call this a *transition*, but it is just a dynamic branch to event program.

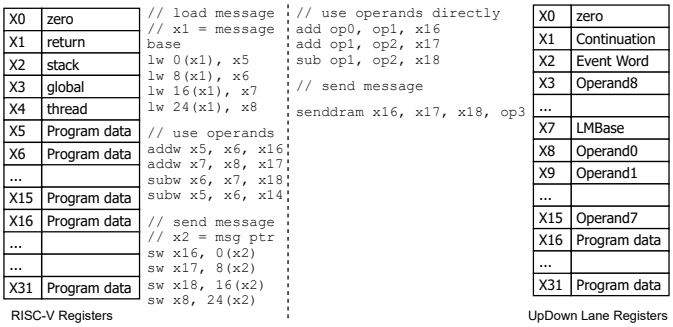


Fig. 8: UpDown's direct access to operand buffer shortens latency and increases short thread effectiveness. *send* from registers is also crucial.

explicitly handle responses. The memory responses come back as events, tagged with a memory address. This **explicit split-transaction** enables applications to tolerate memory-latency, yielding the execution pipeline which the hardware fills with work from other fine-grained threads. In contrast, conventional CPU/GPU systems rely on latency avoidance through memory hierarchies; software handles binding loads. In UpDown, a single thread can issue 100s of memory requests (using senddram instruction), and yield. These requests can access memory globally, anywhere in the UpDown system. This essentially unlimited number of outstanding requests enables extremely high memory parallelism and tolerance of cross-machine memory latencies of 1000's of cycles [31]. While akin to aggressive hardware approaches to latency tolerance [35], [36], UpDown's approach is novel in how its split-transaction operations use memory addresses as an unlimited synchronization namespace.

UpDown also allows memory operations to control their read size from 1 to 8 64-bit words, providing more efficient data movement (fewer instructions and avoiding overfetching). The full set of features in the UpDown memory access architecture is described and studied extensively in [31].

4) General Acceleration: Programmable Events and Threads: While we often use event and thread synonymously, more precisely, each thread is a set of activations, each started by an event. For programmers, event-driven programming is difficult, akin to hardware state machine design. To bridge the gap, UpDown supports a flexible threading model in hardware that allows programmers to think about threads which string together a collection of events, but have a continuous view of state. The key to this is hardware support of the threading model, providing efficient implementation of state continuity.

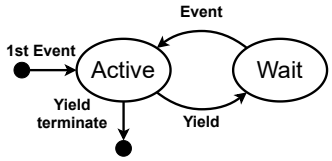


Fig. 9: UpDown Thread Life Cycle. Each time a thread runs (i.e., in active state) is called an activation.

UpDown threads are lightweight, each executes in a hardware-managed context of 16 registers with the remaining register namespace used to address the operand buffer, event queue, and special registers as illustrated in Figure 8. Separate

register contexts are implemented in hardware to keep the thread states of each alive thread. UpDown threads are created by events, and threads may yield and be subsequently restarted by events as in Figure 9. We call the initial creation, and each restarts an *activation*. When a thread *yield*s, it clears the event (i.e. event word and associated operands) and relinquishes the lane for the next activation. When a thread executes *yield-terminate*, it additionally relinquishes its register context. Only one thread runs at a time on a lane. The programmer is responsible for managing parallelism to keep the number of threads under the hardware limit. These mechanisms are directly supported by hardware-managed register contexts, enabling efficient hardware thread scheduling and management.

5) Additional UpDown Fine-grained Features: For efficient interaction, the TOP cores can write events into UpDown queues, data into scratchpads and of course DRAM. Messages are sent to the TOP using the pre-defined address regions in the shared scratchpad, and setting flags polled by the TOP cores. The flag value can be used to indicate which lane that has notified the TOP. And the message data is also memory-mapped and can be read directly by the TOP.

The UpDown lanes can access all of DRAM (split-transaction, message). In accelerators, SRAM is critical for highest performance. To ensure best use, the UpDown system, allows each lane to address all the local scratchpad memories (64 lanes, via load-store, total 4MB) at 1-cycle latency (local bank), 2-cycles (other 63 banks). Not a full shared memory, but rather efficient pooling of the precious resource of fast.

The UpDown approach builds on innovative machine designs for event-driven scheduling and messaging [37]–[42], beyond them has two unique UpDown features: 1) the ability to send message to threads – dynamic instances of parallelism, providing a mechanism for flexible programming and 2) unifying software events with memory operations (and internode messaging). The use of register namespace for message values (operands) reduces the names available for local scratch registers to 16, which appears to be sufficient for fine-grained computation (see Section VI-A). One benefit of reducing the # of registers by one-half reduces the cost for UpDown's 128-way multithreading in each lane.

Collectively, the UpDown's fine-grained features for 1-cycle events, efficient short threads, direct messaging, explicit split-transactions are critical enablers for performance on irregular computation. Efficient events and threads break computation into smaller pieces, allowing for better load balance and increased parallelism. Even irregular parallelism can be broken up and flexibly load balanced, contributing to performance scaling. Direct messaging and explicit split-transaction memory access mitigate irregular communication and unpredictable memory access by breaking them into small pieces that can progress independently, minimizing the impact of long latencies. Together, UpDown's mechanisms enable performance scaling over irregularity in computation, parallelism, memory reference, and communication caused by skewed graphs well.

C. Scalable Global Communication and System Bandwidth

1) Scalable Fine-grained Messaging with NIC-less, Low-overhead Messaging: UpDown accelerators and TOP CPUs are able to send messages directly to the network, as illustrated in the node diagram (Figure 4), and receive messages from the system network. Because the system and intranode packet formats are standard, no special network interfaces are required. UpDown communication is "NIC-less", as in no network interface. Packets flow directly from lanes on one UpDown node to another. The UpDown network adds only speed matching and flow control to manage the longer internode and interrouter link delays (and of course the transition from electrical to the longer optical internode links). Thus, direct messaging can send a message within a node, to any other node, and to access any of the global DRAM in the UpDown system.

2) Scalable Global Bandwidth with a Diameter-3, Direct Network: The UpDown system network is a low-latency direct network, provisioned for high bandwidth to meet the target applications that exhibit low data reuse. The system network employs an expander graph-based topology, Polarstar, and high-speed links to create scalable global bandwidth (see Figure 10). The network is sized so that the node injection bandwidth is a significant fraction of UpDown node memory bandwidth (see Table VII). The diameter-3 topology provides a nearly flat network locality structure. Polarstar [43], is descended from the Polarfly topology, in turn evolved from the Dragonfly topologies used in many of today's HPE/Cray supercomputers such as Argonne's Aurora [12], [44], [45].

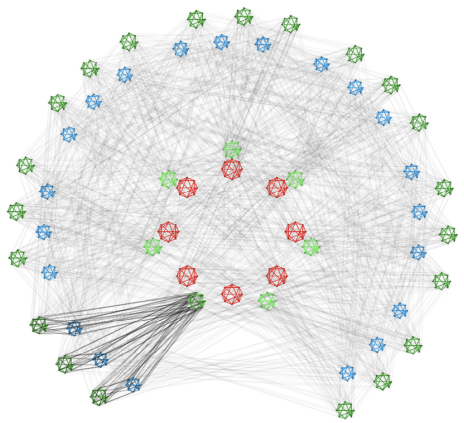


Fig. 10: The UpDown system network is a scalable Polarstar network (Example Polarstar11 with an ER7 structure graph from [43]). Highlighted bundles of links and supernodes are a supernode cluster.

UpDown uses a radix-22 PolarStar [43] network with an ER13 structure graph and an IQ8 supernode, comprising 3,294 switches. . Each UpDown node uses two ports, providing a total of 4 TB/s bidirectional link throughput to the switch. Overall, UpDown system network has 3,294 switches with 32,940 external ports for a total bidirectional bandwidth of 64 PB/s, where 32,768 ports connecting to 16,384 UpDown nodes, and 172 ports to other IO nodes.

This network capability is made possible by recent technology advances of in-package optics projected to enable 51Tbps (6.4TB/s) bidirectional bandwidth in-package as a volume technology by 2027 [46]–[48].

TABLE I: UpDown Instruction Set Architecture Highlights

Category	Execution Mechanisms and Instructions
Lightweight Threading	Thread spawn/invoke: Msg arrival event, send msg; Thread yield/destroy: (`yield, yieldt`)
Event-driven Scheduling	Event and Operand Queues; hardware management w/o any instructions
Ultra-Short Threads	Register naming for Msg Operands, use in add-class, send, and load-store (scratchpad memory) operations
Split-transaction DRAM	DRAM send instructions (`sendm, sendmops, sendmr`); DRAM responses come to software-defined event label
Messaging (incl DRAM)	Single instruction sends of 1-8 words: `send, sendr, sendops`, `sendm` (DRAM), `sendmops` (DRAM)
Traditional Instructions	add-class arith and logic (`add, sub, and, or`) and control flow (`beq, ble, bgt`)
Scratchpad Memory	load-store (`movlr, movrl`), copy block (`bcpy`), streaming compare-n-copy (`cstr`)
Synch (scratchpad)	compare-and-swap (`cswp`)

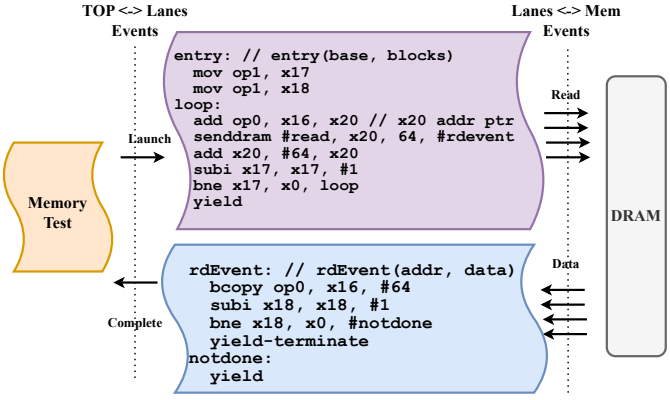


Fig. 11: UpDown split-transaction access to DRAM memory, uses *addr* values (memory addresses) as a scalable namespace to manage memory concurrency and tolerate memory latency.

D. Global Address Space and High Memory Bandwidth

1) Global Address Space: UpDown's system design provides a **single global address space** for all data stored in the global DRAM (spans all machine nodes). Global naming enables program expression independent of data layout. The address space is implemented with a segment-based translation system that provides efficient scalable translation (and application isolation) to petabyte memories and also provides flexible application locality control. The UpDown scheme involves two-levels of translation, separating protection from data layout, and thereby allowing fast unprivileged control over data locality (see [32] for a full description).

Flexible global data placement enables applications to exploit memory bandwidth, capacity, and locality to increase performance. UpDown goes beyond PGAS (partitioned global address space) software models [49]. On the UpDown system, global address translation has no software cost, and application data can be mapped (and remapped) flexibly.

2) High Memory Parallelism: UpDown's design enables high memory-parallelism and memory-latency tolerance with using explicit split-transaction operations (as in Figure 11). With this mechanism, a single thread can create 100's of outstanding requests (*senddram*), then *yield*, allowing the UpDown lane to be used by other threads. UpDown memory operations can specify 1 to 8 64-bit words reducing the number of instructions required for data movement. UpDown programs use the memory addresses (names) as the essentially unlimited synchronization namespace (more details in [31] and Section IV-B3). These mechanisms enable an application to customize access patterns to tolerate irregular latencies that arise in graph and sparse computations; a stark contrast to cache-based cores whose fixed hardware prefetchers and MSHRs.

E. Key Mechanisms and Instruction Set Architecture (ISA)

The UpDown accelerator's generality and flexibility come from its novel Instruction-Set Architecture (ISA) and architectural mechanisms that support the features listed in Section IV-B as described in Table I. Thread control instructions for fast thread suspension, termination (`yield, yieldt`) and creation (`ev, evi, evii`) enable low latency hardware thread switching. The event and operands are mapped into the register namespace, and can be used directly by most instructions (arithmetic, logical, load-store, send). This makes short thread invocations efficient. Split-transaction DRAM read/write is done with `sendm, sendmr, sendmops` instructions. Events can be sent to other lanes with `send, sendr, sendops` instructions. These send-class instructions format short messages (up to 8 words), providing flexible, variable-length data movement. For scratchpad, in addition to load/store, UpDown adds block-copy (`bcpy`), and streaming compare-n-copy (`cstr`) instructions. Basic synchronization support includes compare-and-swap (`cswp`) instructions on scratchpad memory. The full ISA is available here [50].

F. UpDown System Statistical Summary

The system's key local and aggregate properties are summarized described in Table II. First, UpDown has massive MIMD parallelism and instruction processing bandwidth provided by the 33.5 millions UpDown lanes (4 Tera-instructions/second per node, and over 64 peta-instructions/second system-wide). Second, it has extremely high memory bandwidth of 8.8 terabytes/second per node and 153 petabytes/second system-wide. The system network is low-latency and supports communication at nearly one-half the memory bandwidth (vs 1/10th or 1/100th in many modern supercomputers). The bandwidth per instruction is 2 bytes, a high number in modern times. The properties reflect codesign for irregular graph computations, and enable the performance shown in Section VI.

Updown nodes can be densely packed into 19 inch racks, enabling a the entire system to fit into 44 racks. The limits of this density are largely power-density (cooling) for which major advances are growing in commercial high-volume deployment [51]–[53]. A comparison of UpDown to conventional supercomputers can be found in Section VII.

TABLE II: Key UpDown System Element Counts and Capacities

Element	Count	Attributes	
Compute Nodes	16K	TOP (16K), UD Accels (524K), Lanes (33.5M)	DRAM (8.4 PB, 153 PB/s), SRAM (2TB, 1,073 PB/s)
	32 Accelerators/node	4 TIPS (128 BIPS, 4MB Scratchpad each)	greater instruction throughput than TOP
	2,048 Lanes/node	4 TIPS (2 BIPS, 128 HW threads each lane)	64 peta-IPS system-wide
	8 HBM3 Stacks/node	8.8 TB/s (1.1 TB/s 64GB/stack (2030 design))	$>$ 2 Bytes/instruction, 153 PB/s system-wide
	96 core TOP CPU	(864 BIPS total, 9 BIPS each)	multi-issue, deeply pipelined with L1,L2,L3 cache hierarchy
System Network	3,072 switches (32x32)	32K networks ports (2 TBps/link, 4 TBps/node)	64 PB/s system network bisection, transpose memory in 0.13 seconds

V. Methodology

We evaluate the UpDown System with a variety of graph applications on varying machine sizes. The workloads, system elements, and configurations are documented below.

A. Workloads and Datasets

We challenging workloads and datasets for single-node and large-scale multi-node evaluations. These graphs and computatations vary in scale and properties.

TABLE III: Graphs for 1-node Studies

Graph-Datasets	#Vertices	#Edges	Size	#Max deg	#Avg deg
Youtube (Yo) [54]	1.1M	3M	72MB	28,754	5
Web-Google (Wg) [55]	875K	5.1M	87MB	6,332	3
Patents(Pa) [56]	3.8M	16.5M	340MB	793	9
LiveJournal (Lj) [54]	4M	34.7M	622MB	14,815	17
com-Orkut (Or) [54]	3.1M	117M	3.1GB	33,313	76
bio-CE-CX (Bcc) [57]	15.2K	246K	5.19MB	375	32
bio-grid-human (Bgh) [57]	9.4K	62.4K	340KB	616	13

1) Single Node: Graph mining applications: Triangle Counting (TC), Diamond Count (DIA), and Four Cycle Count (4CYC) and graph analytics computations: PageRank (PR), Breadth-First Search (BFS), and Jaccard Similarity Coefficients (JS). Computations are run on a variety of real-world graphs from the SNAP collection [58] (see Table III).

2) Multiple Nodes: Microbenchmarks to characterize the UpDown system's communication and memory bandwidth. Because we are interested in irregular structures, we next evaluate a Scalable Hash Table (SHT), used in many applications.

Applications include PageRank (PR), Breadth-First Search (BFS) and Triangle Counting (TC). We also use 2 streaming workloads - Graph Ingestion (GI) and Partial Match (PM) from the AGILE program [24]. For PR, BFS and TC, we use scalable synthetic RMAT graphs [59] as an exemplar for real-world graph skew. TC runs are on RMAT-s25, and others are up to RMAT-s29. These sizes are limited by simulation speed and memory capacity. Finally, for GI and PM, we use datasets provided by AGILE [24]. See Table IV for a summary.

B. System Modeling

Applications are executed directly on a variety of simulation models, selected for accuracy and scale. A Gem5 extension (**Gem5sim**) models the Updown accelerators, TOP cores, DRAM memories, node and system-level interconnects in full detail, and cycle-by-cycle. For example, detailed models of interconnects and each component are used.

TABLE IV: Graphs for 512-node (large-scale) Studies

Graph-Datasets	#Vertices	#Edges	Size	#Max deg	#Avg deg
RMAT-scale22 [59]	2M	122M	900MB	23K	60
RMAT-scale25 [59]	14M	1.01B	9.8GB	2M	71
RMAT-scale28 [59]	109M	8.3B	68GB	4.1M	76
RMAT-scale29 [59]	160M	8.4B	70GB	8.6M	52
ER-400x128k-p0.01 [24]	131K	524K	2.8MB	16	4
data200 [24]	19.4M	78.9M	4.2GB	1.8M	4

RMAT graphs generated with probabilities a=0.59, b=0.19, c=0.19

TABLE V: Execution Costs of UpDown Instructions [50]

UpDown ISA Operation	Cycles	Note
Simple (`add, sub, beq, bne, and, lsh, etc`)	1	
Scratchpad load/store (`movlr, movrl, mowlr, movwrl`)	1	
DRAM access (`sendm, sendmr, sendmops`)	1+	max scratchpad BW of 2 words / cycle
Messaging (`send, sendr, sendops`)	1+	max scratchpad BW 2 words / cycle
Pipelined (`fp_add, fp_mul, etc`)	1	1 issue, 2 cycles latency
Block move (scratchpad) (`bcpyol, bcopyoli, etc`)	1+	max scratchpad BW of 2 words / cycle
Event word creation (`ev, evi`)	1	
synchronization (`cwsp`)	1	compare & swap, scratchpad only
Transitions (`basic, flagged, majority, default, etc`)	1	
Complex (`fp_div, fp_mod, fp_sqrt, fp_exp, etc`)	8	Not pipelined

See the modeled performance of each UpDown instruction in Table V. Initial clock rate targets were set and initially validated with a variety of design studies [42], [60]. But modeled UpDown architecture performance and operation costs here are derived from a full hardware design being completed for the AGILE UpDown project.

The UpDown design includes a full hardware behavioral model and extensive RTL studies. Recently, we have completed RTL design studies for key eventQ, message sending, and multi-threading features, synthesized using SAED 16nm [61] and closed timing at 2GHz, using the Synopsys Design

compiler. These results match the architecture performance model in Table V. We expect the final design in program target TSMC N3 to exceed this performance target significantly.

The memory hierarchy consists of registers (0 cycles), scratchpad memory (1 cycle), and then on-node (200-300 cycles) and global DRAM. With the Polarstar network, global DRAM is 7x load latency, approximately 1400 cycles.

For larger scale studies, a faster UpDown simulator (**Fastsim**) models the UpDown accelerator with cycle accuracy, but employs simpler models for other elements, such as the TOP CPU, HBM stacks (latency, bandwidth), and the system network (latency). Fastsim is validated one to 4-node scale against Gem5 simulations, and for all of the applications in this paper, the simulations match within 5% runtime or are conservative. This multi-node Fastsim model does at-scale simulation of 512 UpDown nodes (over 1 million parallel lanes) and large application data sizes.

VI. Evaluation

A. Fine-grained Parallelism

UpDown achieves high-performance by exploiting fine-grained thread invocations (10 to a few hundred instructions), radically lower than the 1,000,000s of instructions per thread switch in a conventional CPU. The challenge is to do so efficiently; the UpDown architecture makes the creation, yielding and destruction of events and threads extremely cheap (single instructions). Threads are expected to yield after their computation to allow another event to execute. We measured several benchmark programs, that achieved high performance, and present basic event (aka invocation) statistics in Table VI. They exhibit the fine-grained computation structure – in threads, number of activations, and average number of instructions per activation. These results, combined with the excellent speedups for these applications reported in Sections VI-B and VI-E, show that UpDown mechanisms allow it to exploit fine-grained parallelism at 10s to 100's of instructions and achieve high overall compute efficiency.

TABLE VI: Thread and Activation Statistics on TC(RMAT-s25), PR(RMAT-s29), JS(ER-400x128k-p0.01), PM(data200)

Application	# of Threads	Average Regs	# of Actvtns	Insts / Actvtn	Lane Utilization
PR	17,125M	5	19,196M	16	72.9%
TC	1,535M	11	189,317M	74	73.5%
JS	2,123M	16	6,708M	28	68.9%
Partial Match	18,525M	11	55,288M	18	0.1%

First, the summary statistics for thread counts and activations for the RMAT graph [59] on TC (scale 25) and PR (scale 29), Erdos-Renyi [24] on JS and data200 [24] on PM (see Table VI), show that UpDown programs have large numbers of threads (1.5 - 18.5 billions), far more than a traditional parallel program. These threads use only a number of registers (conforming to the UpDown architecture limit of 16 per lane, but often quite a bit lower). The threads can involve interacting with memory, interlane synchronization work sharing, and of course computation. The threads are each executed in a large number of short invocations, averaging as few as 16 instructions, and with the highest average being only 74 instructions. This is far shorter than the millions of instructions executed between a typical Linux thread context switch [62]. Finally, we include average lane utilization – the fraction of instruction issue slots that are used by the program. For all three graph applications (TC, JS, PR), despite very fine-grained programs, the lane utilization is high – far higher than that of issue slots in a typical superscalar processor core! The one exception to this is Partial match, which intentionally idles hardware to achieve low-latency response (see Section VI-E5).

We drill down into the detailed fine-grained behavior of several applications, showing histograms statistics for thread activations in Figure 12. All of the histograms show the extremely fine-grained structure of UpDown programs and execution (dozens of instructions for most invocations).

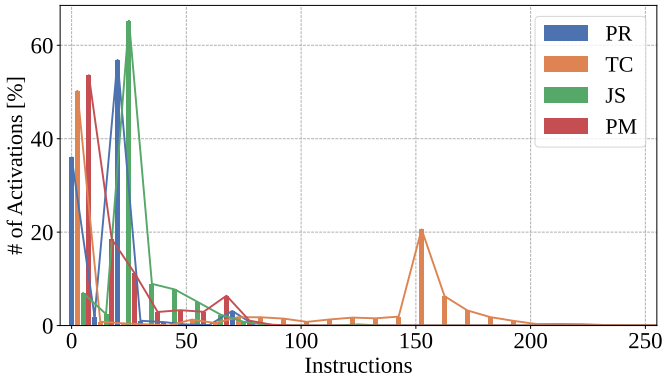


Fig. 12: Histograms of activation length (instruction) for PR, TC (RMAT), JS (ER), and PartialMatch (data200)

Among the applications, triangle Counting (TC) is the chunkiest, with a cluster of activations around 150-175 instructions, but with many less than 10 instructions. The remaining three – Jaccard Similarity (JS), Page Rank (PR), and Partial Match – have similar histograms with many invocations shorter than 10 instructions, and the vast majority less than 50 instructions. So in UpDown, threads manage the parallelism across UpDown accelerator lanes, and achieve high hardware efficiency by sharing lanes using very short invocations. These short invocations come together to achieve high hardware lane utilizations as high as 98% for good performance and scaling. This is dramatically higher than the $\approx$ 25% that graph software on large out-of-order cores in multicore CPUs achieve.

The ability to exploit fine-grained parallelism is critical to extreme scaling. In Figure 13, we compare the scalability of two implementations of the triangle counting algorithm on a Scale 25 RMAT graph. The results show that with vertex-level parallelism (coarse-grained), scaling falls off due to load imbalance around 16 nodes (32,768 lanes). By 64 nodes, the difference with the vertex-pair parallelism (fine-grained) is already 2.3-fold, and at 512 nodes (1,048,576 lanes), the difference is 18x. UpDown's unique architecture mechanisms enable it to exploit fine-grained parallelism and thereby achieve linear performance scaling all the way to

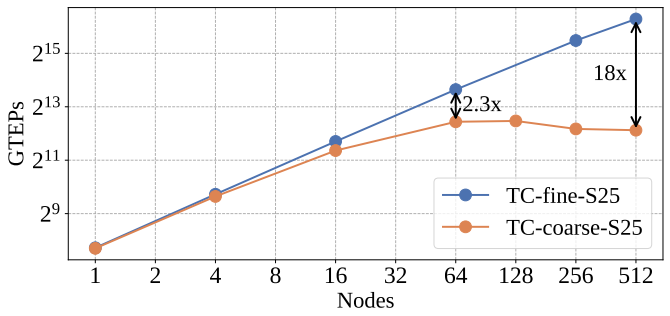


Fig. 13: Fine-grained parallelism (vertex pairs) produces improved scaling compared to coarse-grained parallelism (vertices), 1-512 nodes on scale-25 RMAT graph. The metric is giga traversed edges per second, see Sec. VI-E3.

512 nodes (1M lanes), and the remarkable 80 tera traversed edges/second on a scale-25 RMAT graph.

B. Single Node Performance

We assess performance of a single UpDown node, the building block for the UpDown system, to show how its mechanisms for fine-grained parallelism, efficient data movement, and other architecture features deliver scalable, high performance. Using real-world graphs (Table III), we first compare UpDown performance to hand-tuned single-threaded programs run on conventional CPU core. The speedups achieved for the varied graph workloads (TC, DIA, 4CYC, BFS, PR, and JS) are extreme. A single UpDown node delivers geomean speedups of 1,461-1,865x, exploiting its 32 UpDown accelerators (2,048 simple lanes) with high efficiency. Per workload, the best speedups are 2,868x (TC), 2,507x (DIA), 2,162x (4CYC) and 2,248x (BFS), 4,665x (PR), and 1,782x (JS). In all cases, a single fine-grained program expression of each application can be used for any number of lanes on the node (1-2,048). This programming uniformity also continues to systems of 100's of nodes and millions of UpDown lanes.

The impact of fine-grained parallelism as described in Section 1 is seen in the scaling achieved even on a single node. We present scaling results for TC, DIA and BFS in Figure 15. UpDown achieves linear speedup for TC and DIA, and near linear for BFS, a more challenging benchmark. Exploiting fine-grained parallelism enables performance scalability as high as 31x on 32 accelerators (2048 MIMD lanes) in a single node for a variety of workloads.

C. Scalable Fine-grained Communication and Global Memory Bandwidth

UpDown provides "network interface-less", low-overhead messaging, reducing the cost of sending a message – globally in the machine to a single cycle. In Figure 16, we show the message rates that a single lane can achieve, a single UpDown accelerator, a single UpDown node, and 8 UpDown nodes. For 64B messages (8 64-bit words), these rates exceed giga-messages per second for a single lane and scale up to over a trillion messages per second (10^{12}) for a single node. Compared to a single Perlmutter node with a state-of-the-art Slingshot-2 NIC, these rates are nearly a million times higher [14]. The UpDown message rates exceed those of any existing supercomputer. For UpDown, these message rates decrease as the message size increases because ultimately communication is bandwidth-limited by the interconnect.

A more typical communication performance measurement of delivered bandwidth vs message size is shown in Figure 17. The measurements show that UpDown's NIC-less design scales well and delivers high bandwidth, largely independent of message size. Compared to Perlmutter, the achieved bandwidth for 64B messages is over 100x greater, and only at 4KB messages can an entire Perlmutter node match a single UpDown lane's communication performance. A single UpDown accelerator is capable of 100-10,000 times higher bandwidth. And beyond that at node scale, the UpDown will be limited by the system network communication capabilities (see Section VIII-A).

The combination of UpDown's global address space, efficient memory access and efficient messaging enable high-speed global data movement. To illustrate this, we implemented the UpDown SHMEM Put/Get library (a flexible memory copying library for parallel machines [63]–[65]). This implementation shows that UpDown's high memory bandwidth can be leveraged directly by software to achieve scalable data movement. Note that this is different from all RMA/RDMA systems and DPUs that make use of special hardware data copy engines. UpDown's remarkable efficiency enables software to use sets of threads as "data-movers", as in Figure 18(a).

As shown in Figure 18(b), the software-based UpDown SHMEM libraries significantly outperforms the published GPU performance (NVSHMEM on A100) by 100x. With thousands of threads per node, UpDown's efficient hardware support allows them to work together to provide remarkable performance (10 TB on 1GB transfers) and performance scales well with larger numbers of nodes and of course with large data blocks.

The capability to perform efficient data movement in software has several other advantages. First, because the SHMEM is a user-level library, so no complex translation mappings need be exported to hardware IOMMU's and no such hardware costs are involved. The standard software translation mechanisms suffice. Second, one can build higher-level data movement libraries that copy complex data structures (or even filter them on the fly). For example, high speed streaming performance can be easily realized for hashed access with filtering as shown in Figure 19. In a hardware-based RDMA system, employing user-defined data structures or hashing in the midst of data movement would be impossible.

D. Scalable Hashing with Filtering

Hashing is widely used to distribute irregular data and work; UpDown includes hardware hash instructions that combined with lightweight events and DRAM access are the basis of scalable hashtables. We use a benchmark with 10M keys/node (5000 per lane) and uniformly distributed keys to demonstrate weak scaling of hash + filter. Results show scaling to 2^{35} or 32 giga-hashes per second (see Figure 19), and in a full-sized machine, to 2^{44} or 16 tera-hashes per second. Each

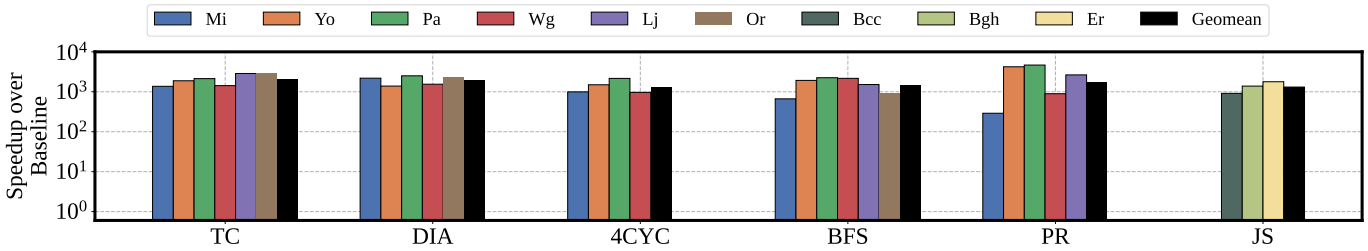


Fig. 14: UpDown speedup. Varied workloads and graphs. The rightmost bar for each cluster is the Geomean.

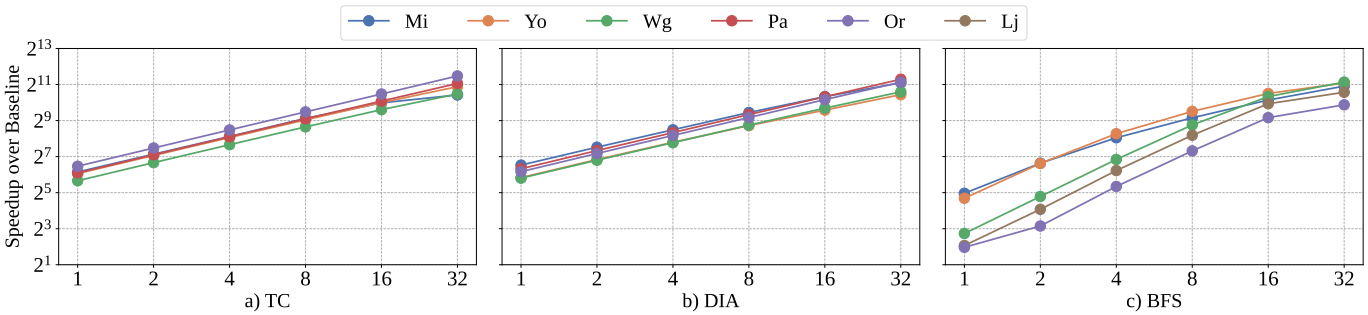


Fig. 15: UpDown speedup for TC, DIA, and BFS. Various datasets. 1-32 accelerators (64 - 2048 lanes)

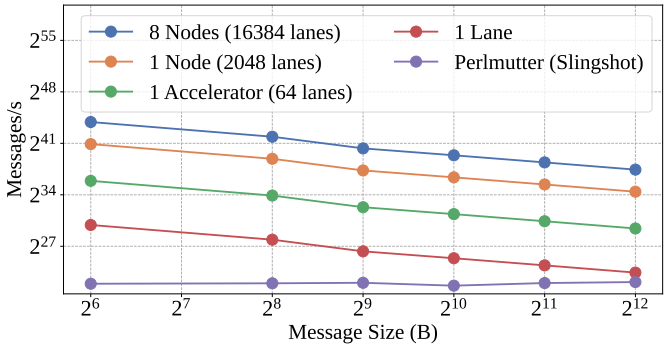
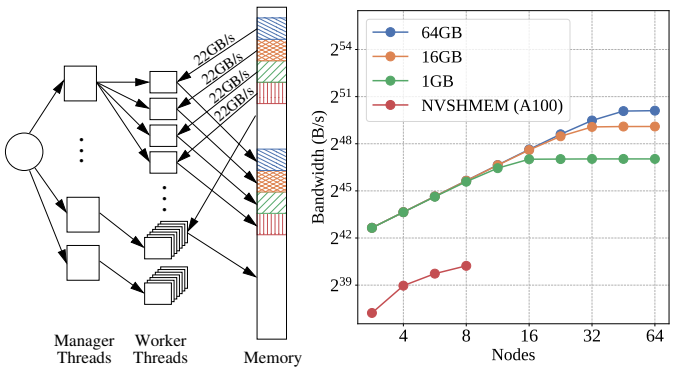


Fig. 16: UpDown achieves 1 Giga-messages/sec/lane, 64 GigaM/s/accelerator, 2 TeraM/s/node. Rates scale with nodes, so message rates reach 32 PetaM/s/system for a full-scale machine. Rate decreases slightly with size, due to data spooling, and ultimately global network limits.



(a) Threads Data Movers (b) UDShmem bandwidth

Fig. 18: UpDown Shmem library combines scalable messaging with high memory bandwidth to efficiently move data in the machine.

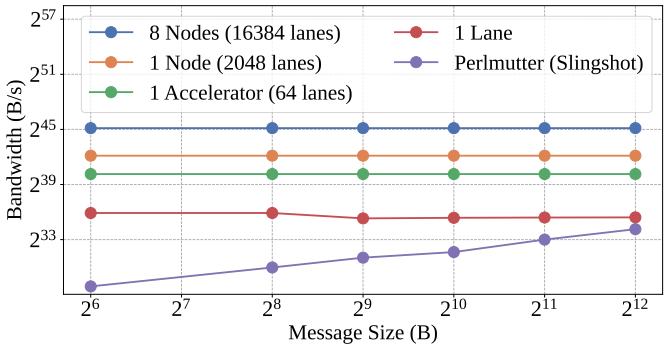


Fig. 17: UpDown lane delivers high bandwidth, even at tiny message sizes (64 bytes), achieving 60 GBs/sec/lane, 3.8 TB/s/accelerator, 122 TB/s/node. Bandwidth drops slightly from 256 to 1024-bytes, due to looping overhead. For a full machine, system bisection limits performance.

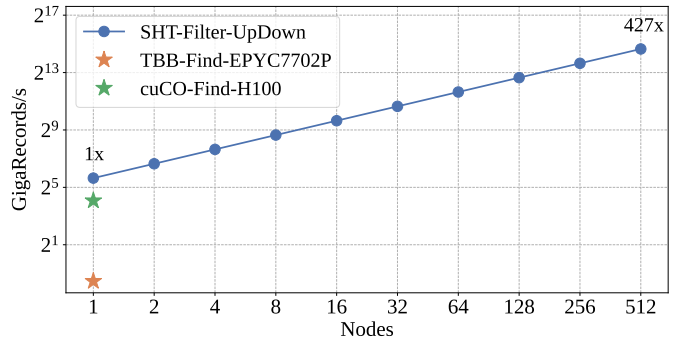


Fig. 19: UpDown can hash with filtering at a scalable rate. Full memory bandwidth can be achieved as the data need not traverse the memory hierarchies of the CPU.

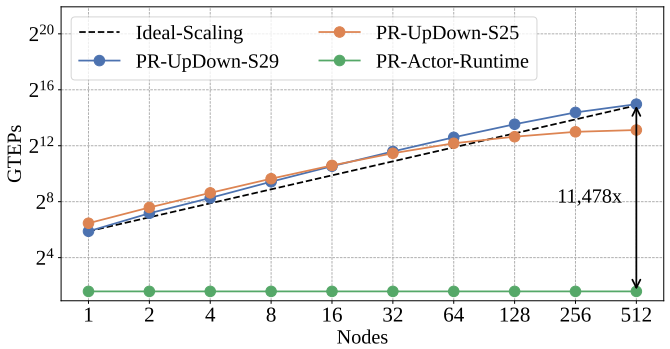


Fig. 20: Giga traversed edges per second (GTEPS) for PageRank on scale 29 and scale 25 RMAT graph.

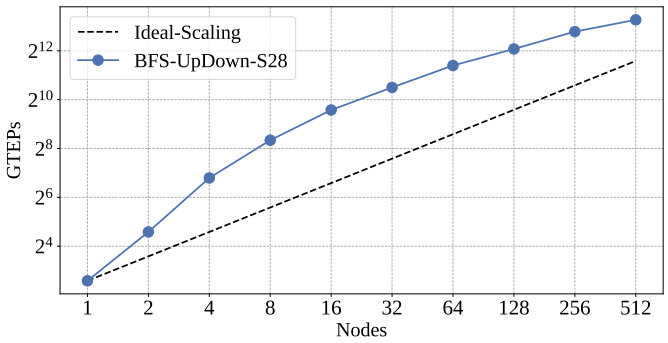


Fig. 21: Giga traversed edges per second (GTEPS) for BFS on Scale 29 RMAT graph.

of the hash operations accesses the global memory, so this computation truly scales up with the size of the machine. In fact, the example shown, first hashes for access and then filters the return values, demonstrating how computation can be efficiently combined with hashed memory access. To show UpDown's performance in context, we show single-node UpDown delivers approximately 150x the hashing performance of a multicore processor (64 cores), and scales to 64,000x performance at 512 nodes. GPU's outperform the CPU, but are still outperformed by the UpDown node.

E. Full UpDown System Evaluation

We evaluate the full UpDown system performance, using three kinds of applications: signature scalable graph applications, additional applications as part of the AGILE program, and finally, latency-sensitive computations that show the novel streaming capabilities of UpDown. These experiments use the datasets in Table IV. For all, we have completed detailed simulation from 1 to 512 nodes (corresponding to 2,048 to 1,048,576 lanes) that take into account graph scaling, load imbalance, and architecture limits. While already an extreme level of parallelism, 512 nodes is short of the full 16,384 node proposed UpDown system architecture.

1) Scalable Fine-grained: Pagerank (PR): For PageRank, we have implemented several algorithms [66], but report performance for a push-based PageRank algorithm, as it achieves high absolute performance and is the simplest to understand. In each iteration, each vertex sends its current value multiplied by its edge weights to its neighbors. These edge-update messages are handled atomically in a software combining library that handles congestion and results in efficient update of the vertex value. Large-scale graph skew is handled by preprocessing the RMAT graph by splitting all vertices with degrees above 1024, resulting in a graph with a maximum degree of 1024. An additional computation step converges the values for split vertices.

Figure 20 shows the performance speedup of the push PageRank in tera-edges per second, as the UpDown system is scaled from 1 to 512 nodes (2,048 to 1M lanes). Using a small graph (scale 25 is approximately 14 M vertices), performance scales well 8,920 giga-updates/second at over 1M lanes; this reflects the UpDown systems ability to exploit fine-grained parallelism extremely efficiently. For comparison, we also plot the best published strong-scaling performance on Perlmutter [14], [67], that shows UpDown achieves 3,000x higher performance aligned based on ISO-power comparison. We also show UpDown performance on a larger, Scale-29 RMAT graph, where UpDown reaches 32,138 giga-updates/second, more than 11,400x the Perlmutter reference.

The performance scales well because (i) the high-degree vertices are split into sibling vertices and are efficiently handled by UpDown's fine-grained vertex-level parallelism and (ii) accelerator-level load balancing.

Note that UpDown should achieve similar rates of edge-updates for larger graphs all the way through those that fill the machine's memory (8.4PB, perhaps 100 trillion edges), as there are no CPU cache effects to change behavior.

2) Scalable Fine-grained: Breadth-first Search (BFS): For Breadth-first Search (BFS), we have implemented several algorithms, but report performance from a simple push, frontier-based algorithm because its performance dynamics are the simplest. In each step, each vertex in the frontier adds each of its neighbors that have not yet been visited to the next frontier. The computation synchronizes when the entire frontier has completed this step, and the next step begins. On UpDown, this algorithm is parallelized at the level of each member (vertex) in the frontier. As with PageRank, we preprocessed the graph to split high-degree vertices, so the execution graph has a max degree of 1024, and that is the max work for a given entry in the frontier. No graph partitioning for data layout is required; data is spread evenly across the machine.

Figure 21 shows the performance speedup of the push BFS algorithm in giga-edges per second, as the UpDown system is scaled from 1 to 512 nodes (2,048 to 1M lanes). Because BFS starts with a single vertex and grows to full parallelism across several iterations and, after the peak, declines in parallelism, its edge-update rates are lower than PageRank. On a scale-29 RMAT graph, performance scales well to 9,900 GTEPS at 1M lanes; this reflects the UpDown system's ability to exploit fine-grained parallelism extremely efficiently. The super-linear scaling at lower node counts arises from BFS caching vertex value updates in scratchpad memory. For larger node counts, scratchpad effectiveness grows, reducing DRAM reads and writes and significantly improving performance and scaling. This demonstrates the flexibility of the software-programmable scratchpad and its benefit for application performance.

From 9,900 GTEPs at 512 nodes, continued good scaling to the full 16K node UpDown system would produce performance of over 300,000 GTEPs, well above Graph 500 champion Fugaku's performance of 198,000 GTEPS using nearly ten times as many nodes (152,064 nodes) [68]. Further, the UpDown numbers do not require the aggressive graph preprocessing done for Fugaku's performance.

As with PR, UpDown should achieve similar rates of edge-updates for BFS on larger graphs all the way through those that fill the machine's memory (8.4PB, perhaps 100 trillion edges), as there are no CPU cache effects to change behavior.

3) Triangle Counting: We consider triangle counting (TC), a more compute-intensive graph comptuation.

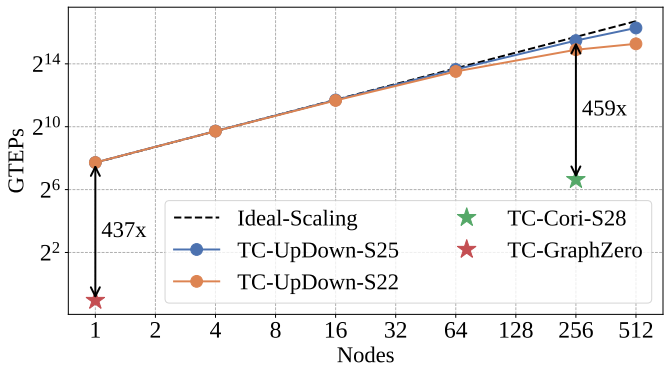


Fig. 22: Giga traversed edges per second (GTEPS) for Triangle Count (1-512 nodes) for scale 22 and scale 25 RMAT graphs.

For TC, we measure a neighbor-based algorithm that compares the edge lists of two vertices connected by an edge (neighbors), optimized for symmetry breaking to avoid redundant work. On UpDown, this algorithm is parallelized at the level of individual edge-list comparisons. A more complete description of the TC algorithm is discussed as an example in Section VII-A. The metric is the number of traversed edges divided by the runtime (after degree orientation).

The performance of TC, with hardware from 1-512 nodes for Scale 22 and 25 RMAT graphs is shown in Figure 22. Speedup for the Scale 22 graph is excellent through 64 nodes (131,072 lanes), but there is not enough parallelism to achieve the extremely good load balance required. Increasing to a Scale 25 graph, improves good performance beyond 256 nodes. We expect that a graph of perhaps Scale 30 or 31 is required to get excellent scaling for the full 16,384 node UpDown system. The performance data supplied by AGILE for the Cori system is measured in triangles per second, calculated using the number of triangles in the graph. To ensure a fair comparison, we convert the Cori performance from 0.033 billion triangles/s to 99.5 billion traversed edges / second. To get this factor, we begin by calculating the Cori execution time of 130 seconds for scale-28 [69]. Next, we estimate the traversed edges for scale-28 based by extrapolating values from graph scale 22 to 25. After transferring the performance data, the performance of UpDown is 459 times greater at 256 nodes.

4) Streaming Graph Ingestion: A natural opportunity for the UpDown system is to operate on streaming data, exploiting its scalable event-driven scheduling. Streaming computation on graphs is an important application area [70]. Such computations require efficient ingestion of graph data – accelerated decoding, parsing, and interning of external graph data. The UpDown system excels at these tasks as it inherited key features from the UDP and UAP [42], [71]. These features, when combined with a full-scale UpDown system, enable extraordinary performance for streaming data – at petabytes/s.

Decoding and parsing tasks are implemented using the TFORM language that embodies the transducer computation model [72], [73]. Input streams are parsed at 1 cycle per symbol, and the encoded state engines recognize arbitrary languages, including FASTA (genome), compressed data (snappy, huffmann), as well as simpler formats such as JSON records or comma separated values (CSV). Streams can be parsed in parallel, and parsing scales well over multiple network streams (e.g., 16 x 400GigE). Parsed objects can be created in the memory of the UpDown machine, using an event that adds it atomically to a shared data structure such as a hash table or B-tree. The AGILE program required several demanding benchmarks of this nature, including graph ingestion with streaming update, network traffic monitoring, and more [24].

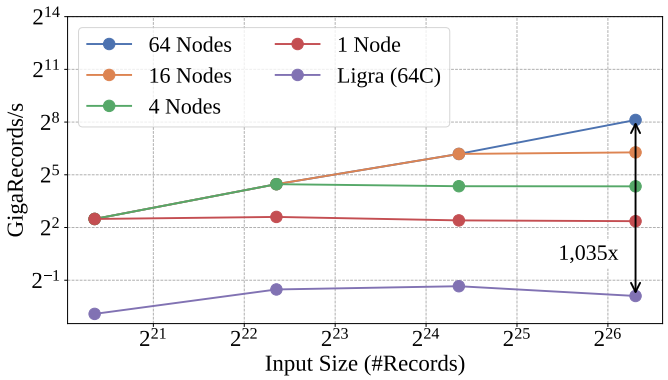


Fig. 23: Scalable ingestion (parsing, filtering, data structure creation). Ligra (GBBS) [74] framework is used as the baseline, parsing Ligra specific text representation of the graphs with as many records as their UpDown test counterparts.

We measure the performance of graph ingestion using TFORM on a range of input sizes (548K to 89M variable-sized records in CSV/ASCII files), and a range of UpDown system sizes available. Each record describes a single vertex or edge with a variable-number of attributes and value-sizes. The performance in records/second, are shown in Figure 23.

For parallelism, each input file is partitioned into 32KB blocks which are each processed as a parallel task on an UpDown lane. The TFORM program correctly handles boundary cases where records span the partition boundary, ensuring each record is added to the in-memory graph exactly once.

The results show that performance scales with the available machine resources and it ultimately gets limited by the input size (application parallelism). The number of nodes given in Figure 23 is the number of nodes available for TFORM. The hash table running in the backend to create parsed memory objects is laid out on a different set of nodes, and its configuration is chosen optimally to maximize its utilization.

With 64 nodes available for TFORM, the UpDown system achieves an ingestion rate of 278 billion records/second, or

a speedup of 1,035x versus a 20-core CPU (see Figure 23). Given enough input parallelism and full 16,384-node UpDown system, performance as high as 8 tera-records/second seems plausible!

5) Latency-sensitive Applications: Partial Match: The partial match computation is an example of a streaming graph computation combining ingestion with stream queries (triggers) that detect patterns completed as data is added to the graphs in memory. As each successive record is received (an edge or a vertex to be added to the graph), the objective is to complete the streaming query (e.g. detect the pattern) with the lowest possible latency. While of general interest as a streaming query, we implemented a specific AGILE program benchmark for the partial match with 8 complex patterns with three elements each [24].

Figure 24 shows that the latency increases linearly as more records are added to the graph. Assigning more computational resources to the PGA decreases latency as the level of parallelism increases.

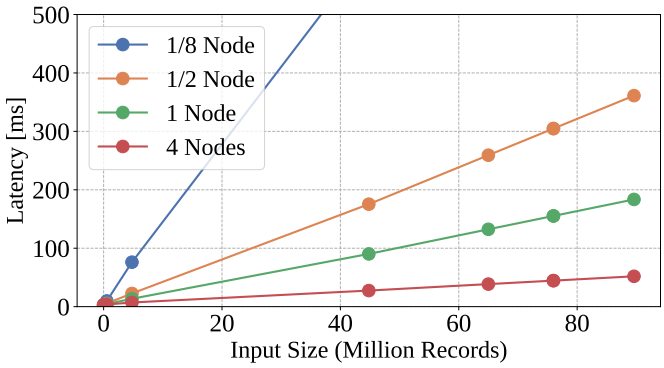


Fig. 24: Streaming pattern match. On the x-axis, the number of records in the database increases as the stream progresses. As the number of lanes used for the PGA increases, the latency decreases.

VII. Flexible Use of UpDown System

A. TC: Flexible Application Programming on UpDown

To illustrate the use of UpDown's architectural mechanisms, we describe the software architecture of the Triangle Counting (TC) ISB. First, consider a traditional CPU program for TC – Figure 25. The graph is represented as an array of neighbor lists. The TC algorithm has been adapted from [75] and [16] (see $Triangle\text{-}Count(G = (V, E))$ in Fig25a). There is parallelism at two levels: vertex-level (master thread) and edge-level (vertex thread). The innermost operation (intersect thread) intersects the neighbor lists of 2 adjacent vertices (see $intersect(v, u)$ in Fig 25b). Three types of memory references cause poor memory hierarchy performance: indirect, when fetching $u.nl[j], (u \in v.nl)$ for intersection, random, when fetching neighbor lists $v.nl[i]$ and $u.nl[j]$ and sequential, for the intersection between $v.nl[i]$ and $u.nl[j]$.

The UpDown TC implementation in Figure 26 illustrates the expression of fine-grained parallelism and programmability. Coarse-grained approaches parallelize using vertex-centric approaches (outer vertex level). UpDown's fine-grained parallelism allows all *vertex iterations* and *intersection iterations* (inner loop) to be executed in parallel as in Figure 26b.

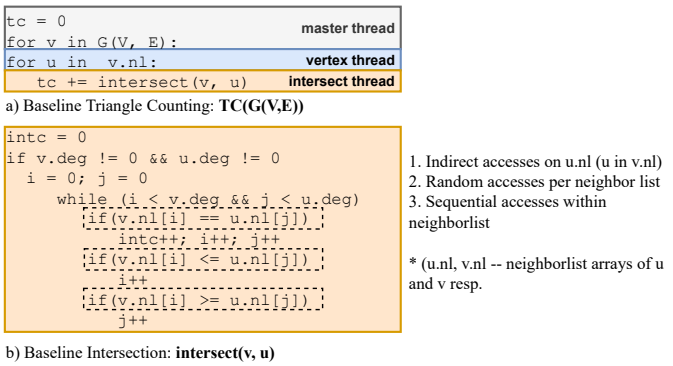


Fig. 25: Triangle Counting Baseline

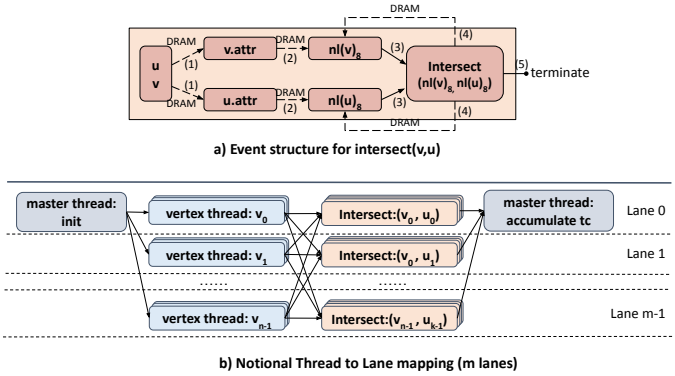


Fig. 26: Triangle Counting on UpDown

Each of the intersections can be executed in parallel (the *fine-grained* TC version in Figure 13). For example, each of these vertex-pair intersection threads could fetch one neighborlist ($v.nl$, using a set of events to tolerate variable DRAM latency) and store it in the scratchpad (16x 8-word fetches and yields), freeing the lane for other computations and thereby tolerating variable latency. When the DRAM responses arrive (100+ cycles later), software events manage out-of-order responses, and when all have arrived, trigger streaming neighborlist comparison.

From a high-level (Figure 26b), there's a hierarchy of vertex and intersect threads that are spread across the UpDown lanes. The master is launched from the CPU which in turn launches vertex threads. After fetching the neighbor vertices ($u \in v.nl$), the vertex thread launches intersect threads. The intersect threads update the local triangle count (tc_val) atomically in scratchpad before termination. Finally, the master thread performs a reduction to complete the computation. The fine-grained parallelization produces balanced, high lane utilization (see Section VI), enabling good performance scaling.

UpDown's hardware threads' ability to receive messages allows them to create both memory and descendant thread parallelism conveniently, much more efficiently than prior advanced messaging architectures [37] [76] [77]. Next the vertex thread uses a second set of events to stream u's neighborlists $u.nl$, intersecting them as they arrive with locally stored $v.nl$. Finally, the vertex thread's local triangle count $TC(v)$ is read by the Top and summed into the global $TC(G)$.

Note that, unlike most graph processing accelerators, none of these graph data-structures are hardwired in UpDown, but

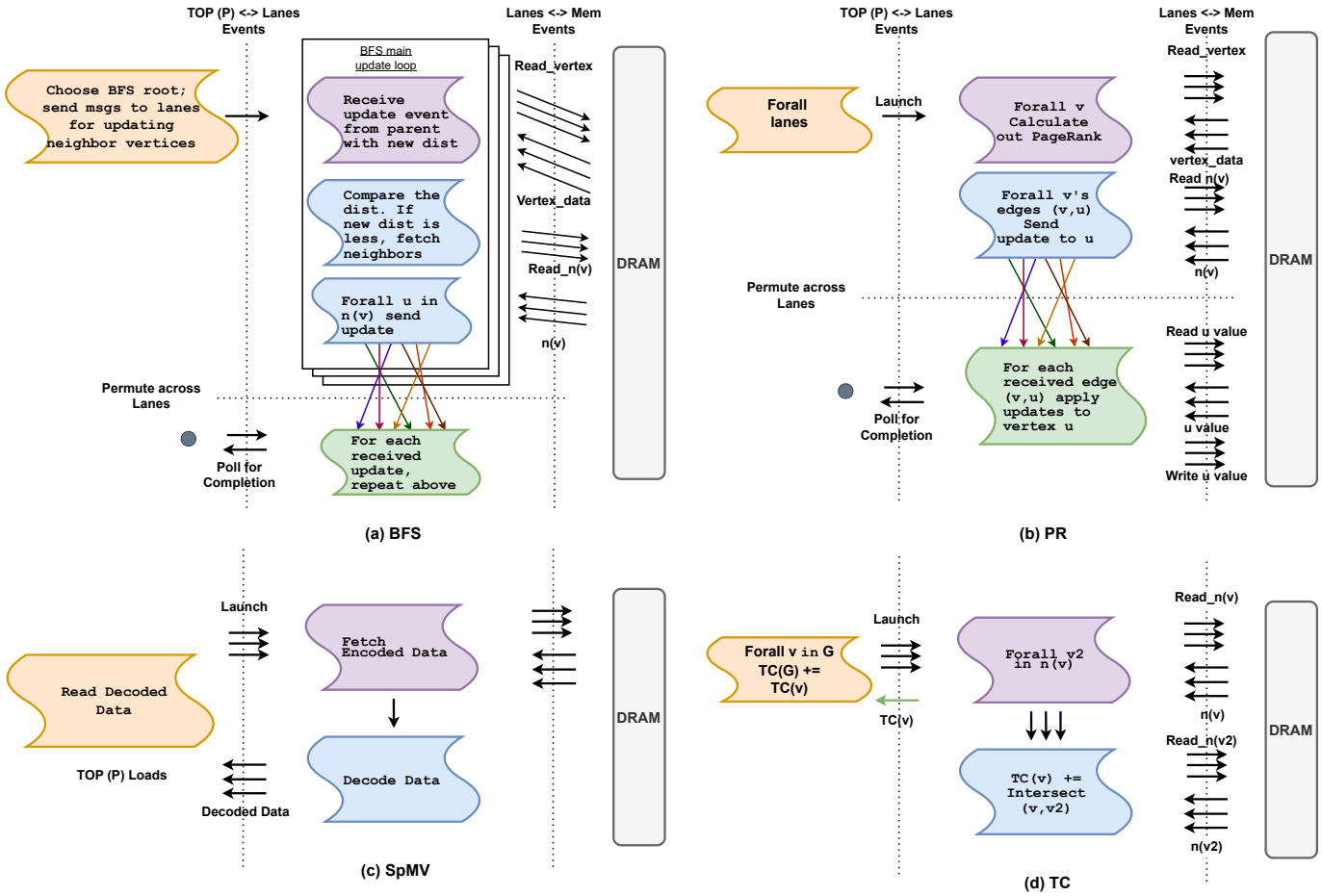


Fig. 27: Application Offloads to UpDown. PageRank is radically fine-grained and shuffles lanes between parts, SpMV uses CPU loads for final data movement. BFS creates an expanding set of tasks for each visited vertex. All exploit UpDown's programmability for application data structures and algorithms.

are rather reflected in the programmable software, enabling expression of memory parallelism and manipulation of arbitrary program data structures.

UpDown's events and threading model enable creation of custom vertex-centric paradigm. Such programs achieve very fine-grained parallelism with UpDown threads by triggering events on any UpDown lane and other system parts such as DRAM and TOP CPU. These models enable convenient high-level programming and scalable high performance.

B. Varied Applications Use of UpDown

UpDown events have varied use in our graph and sparse matrix applications (see Figure 27). Pagerank has a very different structure, using an event for each vertex_increment per weight pushed down each edge. This result is millions of very small threads! (see Table VI). Furthermore, these events are spread across the entire collection of lanes, dictated by the graph edges, creating load-balance challenges. The result is many single word DRAM read and write operations. SpMV on encoded memory has a third distinct event structure, using matrix read events in a block-level pipeline to drive transformation of three CSR arrays (row, column, value). Facing the DRAM, events are used to stream blocks of compressed data.

These different application architectures illustrate the generality of UpDown mechanisms and ISA. The applications use custom software events like vertex fetch, vertex update, count triangles (v1, v2), compute Jaccard similarity (v1, v3), fetch edge, next block, decompress block, memory block fetch, memory block receive, etc. These are just examples of an infinite set of possible software-defined events. Further, UpDown software can generate/receive hardware events. Thus, it enables application software to respond efficiently to DRAM memory read returns, cache events, etc. – with customized application-specific functions.

VIII. Discussion and Related Work

A. HPC Supercomputers

The CPU- and GPU-based supercomputers are designed for highest performance on dense, regular computations [7]. They employ single-node system building blocks (eg. discrete CPU's and GPU's designed for single-server use), and thus communication is expensive (overhead), and incurs high latency [12], [14]. They depend heavily on data-reuse, and thus achieve their best performance on coarse-grained parallelism (billions of instructions) with high data-reuse and limited communication. On irregular applications, they deliver only a small fraction (1-2%) of potential performance [9]; many hardware structures (eg. vector units, tensor cores) consume power but contribute little performance.

	Nodes	Sockets	System Power	Node Injection	Peak Network Injection	Per-Node Bisection	System Bisection	Per-Socket Bisection
UpDown System	16,384	16,384	4.7-9.4 MW	4 TBps	64 PBps	2 TBps	32 PBps	2 TBps
Aurora System	10,624	84,992	55+ MW	0.2 TBps	2.12 PBps	0.069 TBps	0.69 PBps	0.014 TBps
Ratio (UD/Aur)	1.6	0.19	0.09-0.17	20	30.2	29	46.4	148

TABLE VII: Comparing System Balance of UpDown and a conventional Supercomputer Aurora [12]

UpDown is radically different, optimized for fine-grained parallelism of 10's of instructions, low data-reuse, and high communicaton rate. It can achieve a large fraction of its peak performance at lower programming effort. UpDown also supports global programming with an extraordinary system network. We compare a few key dimensions of system and communication performance to a leading supercomputer system, Argonne's Aurora [12] in Table VII. UpDown's codesign produces a radically different system balance.

For example, UpDown's network is has 46x bisection bandwidth and 32x peak injection bandwidth. This despite being a much smaller machine; UpDown is >5.5x smaller by power. While Aurora has fewer nodes, each is much larger. Factoring in two CPUs and 6 GPUs per node, UpDown is >5.1x smaller by socket count. Per-node, UpDown provides 20x injection bandwidth and 29x bisection bandwidth. However, even that understates the difference; on a per-socket basis, the ratio is over 148x for UpDown vs Aurora. The network bandwidth of UpDown is possible because of low-power in-package optical integration that enables > 51Tbps in-package as a volume technology by 2027 [46]–[48]. Overall, UpDown represents a radically different design, enabling dramatically higher performance on graph computing workloads in a much smaller, more efficient system.

B. Heroic Programming of the Sunway System

Several efforts employed heroic distributed memory programming of the 2nd generation Sunway systems [78], [79], an exascale system built in China around 2020. These efforts targeted single-source shortest path (SSSP) and breadth-first search (BFS), and achieved high performance using 40 million cores [78], [79]. These efforts achieved 7,654 GTEPS for SSSP and over 180,000 GTEPS for BFS.

While significant milestones, the programming style used involved optimized degree-aware graph partitioning for data layout, memory-hierarchy specific sorting, and new algorithmic innovation, required by the various bottlenecks in the system – memory and system network bandwidth as well as the capacities of the various memories. And, programmers hand-tuned both parallelism grain-size and problem size to give best performance. It is unclear how this performance could be generalized to a more diverse set of graph computations, or if the fundamental performance demonstrated could be built upon by larger graph applications.

In contrast, the UpDown system's for fine-grained parallelism and scalability eases programming and broadens the range of high performance. UpDown programmers express natural fine-grained parallelism, and lean on the UpDown's global address space to ease programming effort and not perform custom data partition. The ability to tolerate memory latency enables global data layout – and no manual optimization in nearly all of our applications. Our evaluation demonstrates that performance for each application source program scales well over machine and problem size.

C. Co-designed Graph Accelerators

Numerous research projects proposed small-scale (1 to 8 nodes) accelerators for graph mining [80]–[83] and graph analytics [84]–[86], achieving good speedups. However, these systems generally hardwire data-structures, algorithms and graph-specific mechanisms (like symmetry breaking, hardware accelerated set operations, accelerated DFS walkers, high-degree vertex caching etc.) limiting their generality. Rapid algorithmic innovation in graph processing makes flexible acceleration valuable. UpDown takes a different approach, providing software-programmable, flexible acceleration. UpDown mechanisms described in Section IV-B target the key properties needed, e.g., supporting large numbers of ultra-short threads (compute parallelism) and maximizing outstanding memory requests (memory parallelism) to achieve high performance on irregular computations in general.

Several research projects have proposed modular graph accelerators and demonstrated some scalability in performance for a fixed set of graph data structures and algorithms. However, none of these were designed for the massive 10,000-fold scalability required in supercomputers. For example one scales to only a few HMC stacks (no scalable interconnection structure) and another was designed to use only on-chip SRAM, limiting memory capacity [87], [88].

IX. Summary and Future Work

We have described UpDown, a supercomputer co-designed for efficient and high performance execution of irregular graph computations. To overcome irregularity in compute, parallelism, and memory access, UpDown supports architecture mechanisms that make fine-grained task parallelism of 10-instruction invocations efficient, deploys massive memory bandwidth and mechanisms for high memory parallelism.

Flexible programmability, UpDown's mechanisms are general. None are specific to an algorithm or data structure, supporting general events and computation, a variety of data structures and computations as described in Section VII.

Evaluation shows scalable high performance. Communication supports dramatically (1,000x) smaller message sizes, (50x) higher communication rates. Application performance on a range of computations shows excellent strong scaling to 1 million lanes (512 nodes) on small graphs, which is a difficult test. Results show 500-fold to 10,000-fold performance advantage compared to conventional supercomputers.

This large performance increase is the opportunity of co-design for fine-grained, irregular computations. With larger graphs, performance will also be high.

Promising future directions include further evaluation of UpDown, with a wider suite of graph applications, and moving beyond to a broader range of challenging irregular applications. Another exciting area is study of UpDown scaling beyond 1 million lanes, to the full-scale design of 33.5 million lanes, 16,384 nodes; a major simulation challenge.

REFERENCES

[1] Gautham Dharuman, et. al., "Mprot-dpo: Breaking the exaflops barrier for multimodal protein design workflows with direct preference optimization," in *Supercomputing*, ser. SC '24. IEEE, 2024.

[2] Honghui Shang, et.al., "Pushing the limit of quantum mechanical simulation to the raman spectra of a biological system with 100 million atoms," in *Supercomputing*, ser. SC '24. IEEE, 2024.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107–113, Jan. 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[4] 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 *ACM SIGMOD*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146.

[5] J. Lu and A. Thomo, "An experimental evaluation of giraph and graphchi," in *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '16. IEEE Press, 2016, p. 993–996.

[6] A. Petitet, et. al,, "HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," https://www.netlib.org/benchmark/hpl/.

[7] Community, "The top 500 list," https://top500.org/lists/top500/2024/11/.

[8] "The ml perf benchmark," MLcommons, https://mlcommons.org/benchmarks/.

[9] Community, "Top 500 list for high performance conjugate gradient - HPCG," https://top500.org/lists/top500/2024/11/.

[10] J. Markoff, "Attack of the killer micros," *New York Times*, 1991, https://www.nytimes.com/1991/05/06/business/the-attack-of-the-killer-micros.html.

[11] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Connelly, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane, "Design and evaluation of an hpvm-based windows nt supercomputer," *IJHPCA*, vol. 13, no. 3, pp. 201–219, 1999.

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

[13] J. Mccalpin, "Stream : Sustainable memory bandwidth in high performance computers," *http://www.cs.virginia.edu/stream/*, 2006. [Online]. Available: https://cir.nii.ac.jp/crid/1572824500380421760

[14] N. Ding, M. Haseeb, T. L. Groves, and S. Williams, "Evaluating the Performance of One-sided Communication on CPUs and GPUs," in *Supercomputing '23 Workshops*. ACM, 2023, pp. 1059–1069.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[16] A. S. Tom and G. Karypis, "A 2d parallel triangle counting algorithm for distributed-memory architectures," in *48th ICPP*, ser. ICPP 2019. ACM, 2019.

[17] P. M. Kogge, "Jaccard coefficients as a potential graph benchmark," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 921–928.

[18] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, 2013, pp. 135–146.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel computation on natural graphs," in *10th USENIX OSDI 12)*. USENIX, Oct. 2012, pp. 17–30.

[20] R. R. McCune, T. Weninger, and G. Madey, "Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing," *ACM Computing Surveys*, vol. 48, no. 2, pp. 1–39, Nov. 2015.

[21] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *30th SPAA*. ACM, 2018, p. 393–404.

[22] C.-Y. Gui, L. Zheng, B. He, C. Liu, X.-Y. Chen, X.-F. Liao, and H. Jin, "A Survey on Graph Processing Accelerators: Challenges and Opportunities," *Journal of Computer Science and Technology*, vol. 34, no. 2, pp. 339–371, Mar. 2019.

[23] W. Gropp, E. Lusk, A. Skjellum, and R. Thahur, "Using mpi—2nd edition: Portable parallel programming with the message passing interface," 1999.

[24] IARPA, "AGILE: Advanced graphic intelligence logical computing environment program," US Intelligence Advanced Research Program Administration (IARPA), January 2022, https://www.iarpa.gov/research-programs/agile.

[25] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 293–302.

[26] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 56–65.

[27] X. Chen, R. Dathathri, G. Gill, L. Hoang, and K. Pingali, "Sandslash: a two-level framework for efficient graph pattern mining," in *ICS*, ser. ICS '21. ACM, 2021, pp. 378–391.

[28] Lu and Chung, *Complex Graphs and Networks (CBMS Regional Conference Series in Mathematics) (CBMS Regional Conference Series in Mathematics, 107)*. Springer, 2006.

[29] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education, 2005.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[31] A. Rajasukumar, T. Zhang, R. Xu, and A. A. Chien, "Updown: A novel architecture for unlimited memory parallelism," in *International Symposium on Memory Systems*. ACM, 2024, p. 61–77.

[32] Y. Wang, S. Pernarnau, and A. A. Chien, "Updown: Combining scalable address translation with locality control," in *In MEMO '24: International Workshop on Memory System, Management and Optimization*, November 2024, held in Conjunction with SC 2024.

[33] ARM, "Arm a-profile a64 instruction set architecture," 2024, https://developer.arm.com/documentation/ddi0602/2024-09/?lang=en.

[34] R. Foundation, "Risc-v technical specifications," 2024, https://riscv.org/specifications/ratified/.

[35] B. J. Smith, "Architecture and applications of the hep multiprocessor computer system," in *Real-Time signal processing IV*, vol. 298. SPIE, 1982, pp. 241–248.

[36] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *2009 IEEE International Symposium on Parallel and Distributed Processing*, 2009, pp. 1–9.

[37] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a message-driven processor," in *ISCA*, ser. ISCA '87. ACM, 1987, p. 189–196.

[38] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *ISCA*. IEEE, 2004, p. 2.

[39] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "T: A multithreaded massively parallel architecture," in *ISCA*, ser. ISCA '92. ACM, 1992, p. 156–167.

[40] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iwarp," in *ISCA*. ACM, 1990, p. 70–81.

[41] J.-P. Fricker, "The cerebras cs-2: Designing an ai accelerator around the world's largest 2.6 trillion transistor chip," in *Proceedings of the 2022 International Symposium on Physical Design*, ser. ISPD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 71.

[42] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "UDP: a programmable accelerator for extract-transform-load workloads and more," in *50th MICRO*. ACM, Oct. 2017, pp. 55–68.

[43] K. Lakhotia, L. Monroe, K. Isham, M. Besta, N. Blach, T. Hoefler, and F. Petrini, "Polarstar: Expanding the scalability horizon of diameter-3 networks," *arXiv preprint arXiv:2302.07217*, 2023.

[44] K. Lakhotia, M. Besta, L. Monroe, K. Isham, P. Iff, T. Hoefler, and F. Petrini, "Polarfly: A cost-effective and flexible low-diameter topology," *arXiv preprint arXiv:2208.01695*, 2022.

[45] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, p. 77–88, jun 2008.

[46] "Solving bandwidth and power bottlenecks with optics," https://ayarlabs.com/.

[47] Intel Corporation, "4 tbit/s optical compute interconnect chiplet for xpu-to-xpu connectivity," Hot Chips, 2024, http://hotchips.org/.

[48] Broadcom, "An ai compute asic with optical attach to enable next generation scale-up architectures," Hot Chips, 2024, http://hotchips.org/.

[49] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global arrays: A portable "shared-memory" programming model for distributed memory computers," in *Supercomputing*. Washington, DC, USA: IEEE Computer Society Press, 1994, p. 340–349.

[50] A. Chien, A. Rajasukumar, M. Nourian, Y. Wang, T. Su, C. Zou, and Y. Fang, "Updown accelerator instruction set architecture (isa) v2.4," Univ Chicago, Technical Report TR-2024-03, July 2024. [Online]. Available: https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2024-03

[51] M. Jalili, I. Manousakis, Í. Goiri, P. A. Misra, A. Raniwala, H. Alissa, B. Ramakrishnan, P. Tuma, C. Belady, M. Fontoura *et al.*, "Cost-efficient overclocking in immersion-cooled datacenters," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 623–636.

[52] LiquidStack Inc., "The world leader in 2 phase immersion cooling," https://liquidstack.com/.

[53] S. Eguchi, M. Umematsu, and O. Ohshima. (2020, Nov.) System implementation technologies of supercomputer fugaku. [Online]. Available: https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article04.html

[54] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012, pp. 1–8.

[55] 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," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[56] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *ACM SIGKDD*, 2005, pp. 177–187.

[57] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[58] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[59] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*. SIAM, 2004, pp. 442–446. [Online]. Available: https://doi.org/10.1137/1.9781611972740.43

[60] A. Rajasukumar, "Updown: An intelligent data movement architecture for large scale graph processing," Master's thesis, University of Chicago, Chicago, March 2023.

[61] "Synopsys teaching resources," https://www.synopsys.com/community/university-program/teaching-resources.html.

[62] "The Linux Kernel Documentation - CFS Bandwidth Control," https://www.kernel.org/doc/html/latest/scheduler/sched-bwc.html, accessed: 2024-09-23.

[63] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM New York, NY, USA, 1998, pp. 1–31.

[64] J. Nieplocha, D. Baxter, V. Tipparaju, C. Rasmunssen, and R. W. Numrich, "Symmetric data objects and remote memory access communication for fortran-95 applications," in *Euro-Par 2005 Parallel Processing*, J. C. Cunha and P. D. Medeiros, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 720–729.

[65] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *4th PGAS Conferencel*, 2010, pp. 1–3.

[66] 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

[67] Y. Elmougy, A. Hayashi, and V. Sarkar, "Highly scalable large-scale asynchronous graph processing using actors," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, 2023, pp. 242–248.

[68] 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 *Supercomputing*. IEEE, 2024.

[69] IARPA, "AGILE program workflows," https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Program_Workflows_FINAL.pdf.

[70] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, p. 1860–1876, Jun. 2023.

[71] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *48th MICRO*, ser. MICRO-48. New York, NY, USA: ACM, 2015, p. 533–545.

[72] M. Nourian, T. Nguyen, A. A. Chien, and M. Becchi, "Data transformation acceleration using deterministic finite-state transducers," in *2022 IEEE Big Data*, 2022, pp. 141–150.

[73] M. Nourian, "Analysis of finite state automata and transducers processing acceleration on disparate hardware technologies," PhD thesis, North Carolina State University, Rayleigh, 2021.

[74] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *30th SPAA*, ser. SPAA '18. ACM, Jul. 2018, p. 393–404.

[75] J. Shun and K. Tangwongsan, "Multicore triangle computations without tuning," in *2015 IEEE 31st International Conference on Data Engineering*, Apr. 2015, pp. 149–160, iSSN: 2375-026X.

[76] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The j-machine multicomputer: An architectural evaluation," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 224–235.

[77] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[78] Y. Wang, H. Cao, Z. Ma, W. Yin, and W. Chen, "Scaling graph 500 sssp to 140 trillion edges with over 40 million cores," in *Supercomputing*, 2022, pp. 1–15.

[79] H. Cao, Y. Wang, H. Wang, H. Lin, Z. Ma, W. Yin, and W. Chen, "Scaling graph traversal to 281 trillion edges with 40 million cores," in *27th PPoPP*. ACM, 2022, p. 234–245.

[80] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. G. Luna, J. Golinowski, M. Copik, L. Kapp-Schwoerer, S. Di Girolamo, N. Blach, M. Konieczny, O. Mutlu, and T. Hoefler, "SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems," in *MICRO-54*, ser. MICRO '21. New York, NY, USA: ACM, Oct. 2021, pp. 282–297.

[81] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining," in *ISCA 2021*. Valencia, Spain: IEEE, Jun. 2021, pp. 581–594.

[82] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. Blaauw, T. Mudge, and R. Dreslinski, "NDMiner: accelerating graph pattern mining using near data processing," in *ISCA*. New York New York: ACM, Jun. 2022, pp. 146–159.

[83] Y. Wu, J. Zhu, W. Wei, L. Chen, L. Wang, S. Wei, and L. Liu, "Shogun: A task scheduling framework for graph mining accelerators," in *50th ISCA*. ACM, 2023.

[84] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, "DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing," in *2021 HPCA*, Feb. 2021, pp. 371–384.

[85] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: a high-performance and energy-efficient accelerator for graph analytics," in *49th MICRO*, ser. MICRO-49. Taipei, Taiwan: IEEE Press, Oct. 2016, pp. 1–13.

[86] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2020, pp. 908–921.

[87] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[88] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A data-local program execution and architecture for memory-bound applications," in *2023 HPCA*, 2023, pp. 718–730.