KVMSR+UDWeave: Extreme-Scaling with Fine-grained Parallelism on the UpDown Graph Supercomputer

Alexander Fell, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Wenyi Wang, Jose M. Monsalve-Diaz, Andronicus Rajasukumar, Jiya Su, Ruiqi Xu, Rajat Khandelwal, Tianchi Zhang, David F. Gleich, Yanjing Li, Hank Hoffmann, Andrew A. Chien
Department of Computer Science
University of Chicago, Purdue University, Argonne National Lab
USA
aachien@uchicago.edu

Abstract

Programming irregular graph applications is challenging on today's scalable supercomputers. We describe a novel programming model, KVMSR+UDWeave, that supports extreme scaling by exposing fine-grained parallelism. By enabling the expression of maximum parallelism, it opens the door for extreme scaling, even on both small and large graph problems.

KVMSR+UDWeave cleanly separates the three key dimensions of parallel programming: parallelism, computation binding, and data placement. This decomposition reduces effort to achieve scalable, high-performance for graph algorithms on real-world, highly skewed graphs. Key features of the UpDown supercomputer (computation location naming and shared global address space) enable decomposition and scalable, high performance.

In the IARPA AGILE program, we built numerous graph benchmarks and workflows, and use them to illustrate the programming model. Simulation results for UpDown show excellent strong-scaling to million-fold hardware parallelism and high absolute performance. Results suggest KVMSR+UDWeave enables reduced programming effort for scaling the most demanding irregular applications.

CCS Concepts

• **Computer systems organization → Multicore architectures**; **Single instruction, multiple data**; *Interconnection architectures.*

Keywords

parallel computing, high-performance computing, graph computing, supercomputer, mapreduce

ACM Reference Format:
Alexander Fell, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Wenyi Wang, Jose M. Monsalve-Diaz, Andronicus Rajasukumar, Jiya Su, Ruiqi Xu, Rajat Khandelwal, Tianchi Zhang, David F. Gleich, Yanjing Li, Hank Hoffmann, Andrew A. Chien. 2025. KVMSR+UDWeave: Extreme-Scaling with Fine-grained Parallelism on the UpDown Graph Supercomputer. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3731599.3767499

1 Introduction

While high-performance computing (HPC) has succeeded in producing efficient and scalable application performance on regular problems using MPI+X, there are large classes of problems with irregular parallel computation and data structures that remain unconquered. These applications are often never attempted on large-scale HPC systems, or when attempted, use simplified algorithms, and produce either poor scaling or poor efficiency. The net result is poor overall performance.

We focus on the irregular parallelism in graph applications, particularly on real-world or social-network graphs with high skew. These applications have a difficult structure, but tantalizingly high potential parallelism. They also have a tremendous problem scale, with billions of vertices and trillions of edges, making them an appealing opportunity for scalable computation. To date, small-scale systems (multicore) have achieved high efficiency, but limited scalability [12], and large-scale systems (cloud, supercomputer) have the occasional, rare achievements in good scalability but often suffer poor efficiency [24, 25]. Such graph problems are not easily amenable to MPI+X because of their complex data structure, irregular data, and irregular parallelism, which present extreme programming difficulties.

Our KVMSR+UDWeave approach uses a map-reduce style to organize large-scale parallel computation over shared global state. KVMSR builds on the demonstrated power of MapReduce [5, 11, 45], extending it with fine-grained parallelism and flexible sharing of mutable data structures. A critical enabler is UpDown's global address space. UDWeave provides expression of computation, lightweight tasks, and small-scale parallelism, enabling the definition of tasks, data access, synchronization, and basic computation. Together, KVMSR+UDWeave realize the ability to decompose a parallel program into three independent dimensions as illustrated in Figure 1.

The parallel application description in KVMSR+UDWeave expresses the computation structure and particularly its fine-grained parallelism, communication, and data access. Computation binding schemes in KVMSR aid programmers in efficiently placing computation onto compute engines. Data placement is provided by a library called DRAMmalloc and supports programmers in organizing data in the machine for bandwidth and locality. This clean

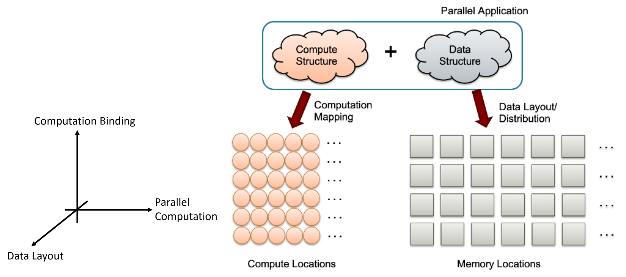

Figure 1: KVSMR+UDWeave enables applications to express parallelism separately from performance optimization (via computation binding and data placement). This makes it possible to build complex applications that can exploit complex, irregular parallelism.

three-dimensional formulation is made possible by architecture features of the UpDown system, including efficient fine-grained tasks, single-cycle messages, global address space, plentiful memory bandwidth, and a high-performance system network.

We describe the KVMSR+UDWeave approach, giving code examples and explaining key features. Next, we show application of the model to challenging graph computations, including PageRank, Breadth-first Search, and Triangle Count. This highlights the programming power of KVMSR+UDWeave model for irregular applications and expressing fine-grained parallelism. Finally, we show the scalable, high performance achieved on the UpDown architecture, which provides powerful architectural support for the programming model.

Specific contributions include

- Design of the KVMSR+UDWeave programming model that enables applications to separate parallelism, computation binding, and data placement.
- Demonstration of the KVMSR+UDWeave model on challenging graph applications, showing how fine-grained, irregular parallelism can be expressed, and managed at high parallelism (16K nodes, 33M lanes).
- Evaluation that shows excellent strong-scaling to million-fold parallelism on small graphs and high absolute performance that exceeds today's supercomputers.
- Empirical evaluation of programmability with several quantitative metrics.

The rest of the paper is organized as follows. In Section 2, we introduce KVMSR+UDWeave, a novel programming model approach. In Section 3.1, we describe the UpDown system architecture. Examples of applying KVMSR to a set of graph applications are discussed in Section 4. We show the strong scaling and high performance that these applications can achieve in Section 5. In Section 6, we frame how KVMSR+UDWeave sits in the landscape of parallel programming models. Section 7 summarizes.

2 KVSMR+UDWeave

We use KVMSR to organize the large-scale parallel structure of programs. KVMSR is written in UDWeave, a system language to express detailed computational structure for the UpDown architecture, including parallelism. Thus, the imperfect analogy is from KVMSR to MPI, and X to UDWeave. Because we need a syntax to describe UpDown code, we begin with a discussion of UDWeave.

2.1 UDWeave: Fine-grained, Small-scale Parallelism

UDWeave exposes UpDown's powerful mechanisms for fine-grained parallelism, including software-directed threading and message send/task creation. These exploit UpDown's hardware thread management for fine-grained tasks. UDWeave also provides basic data types, expressions, and control structures, using a syntax similar to C [34]. We illustrate UDWeave with examples.

2.1.1 Threads and Events. UDWeave programs define threads that each contain one or more events. When instantiated, threads are similar to objects, with events triggered by messages [3, 9, 48]. Events are similar to member functions and execute atomically. Therefore, there are no race conditions for thread state variables, though synchronization is still required for parallel computation over global shared memory.

In Listing 1, the thread `TExample` has several thread variables (`result`, etc.) and one event named `reduction`. Software-controlled thread management is shown by `yield` which exits the event, preserves thread state, and releases the lane to other events. In contrast, a `yield_terminate` would exit the event and deallocate the thread.

```
thread TExample {
  uint64_t result;
  uint64_t* local scratchpad_ptr,
  uint64_t* globalDRAM_ptr;

  event reduction(uint64_t n) {
    result = 0;
    for(uint64_t i=0; i<n; i=i+1) {
        result = result + scratchpad_ptr[i];
    }
    send_event(CCONT, result, IGNRCONT);
    yield;
  } }
```

Listing 1: The Structure of a UDWeave Program

Variables can be declared with thread scope (e.g., `result`), making them accessible by all thread events and preserved across events. They can also be declared with a scope such as an event or for-loops (e.g., `i`). Event parameters (e.g., `n`) use dedicated operand registers, enabling optimized execution for short events [4, 32].

Two memory types are supported: 1) Scratchpad, which is primarily lane private, but can be pooled among the 64 lanes in a UpDown accelerator, and 2) shared global memory (DRAM). Scratchpad allocations are declared as `local`.

An event executes in a computation location, called a lane and identifiable by a network ID, and has a thread context ID. Static properties include the number of operands and the event label (the address of the event in the program). Altogether, they form a 64-bit value called the *event word*.

2.1.2 Intrinsics. UDWeave intrinsics enable manipulation of event words (that include the `networkIDs` used for computation binding), and messaging primitives to launch events (create parallel tasks, communicate with other threads, access shared global memory). These intrinsics enable flexible coordination between events and threads.

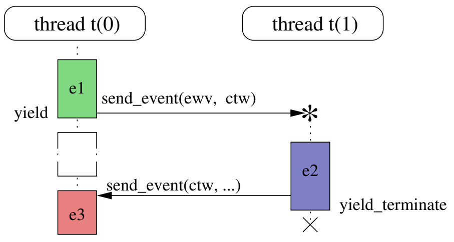

Figure 2: Call-return composition using UpDown's continuation passing style.

- `evw_new(networkID,eventLabel)`: returns an event word for a new thread on the given lane with the event name.
- `evw_update_event(oldEventWord,newEventLabel)`: returns an event word with the new event name, other fields (e.g., thread context ID) remain unchanged.
- `send_event(eventWord,data0,data1,[data2,]continuationWord)`: Sends a message (event) to the target (lane, thread), with optional continuation.

2.1.3 Flexible Event Composition (Continuations). UpDown (and UDWeave) support flexible composition via explicit continuations. Conventional call-return (RPC), third-party, or even more complex compositions can be efficiently implemented. An example of a call-return composition is illustrated in Listing 2 and Figure 2.

```
thread TCallReturn {
    event e1() {
        print("I am in e1");
        uint64_t evw = evw_new(curNetworkID+1, e2);
        uint64_t ctW = evw_update_event(CEVNT, e3);
        // send uint64 0 and uint64 1 to e2 on a new
            thread
        send_event(evw, 0, 1, ctW);
        yield; }
    event e2(uint64_t data0, uint64_t data1) {
        print("I am in e2 and received this data: %lu, %
            lu", data0, data1);
        send_event(CCONT, IGNRCONT);
        yield_terminate; }
    event e3() {
        print("I am back from e2");
        //...
} }
```

Listing 2: Simple call-return event composition in UDWeave.

Event `e1` is invoked and builds an event word to use the next lane (`curNetworkID+1`, with `curNetworkID` the current Lane ID) to execute `e2` in a new thread. A continuation word, `ctW`, is created, indicating the return should come to the current thread. The `send_event` sends a message with two arguments (`0` and `1`), and the continuation word. At this point, `e1` could send messages creating additional parallel tasks in UpDown (small-scale parallelism). As shown in the example, these tasks can be very fine-grained (10-100 instructions) and are supported efficiently by special UpDown hardware.

Flexible third-party composition is often used in UpDown programs. It can affect third-party data transfers, including into a computational pipeline, the composition of map and reduce functions in KVMSR (where the intermediate key-value map does not need to be materialized), and the composition of application phases.

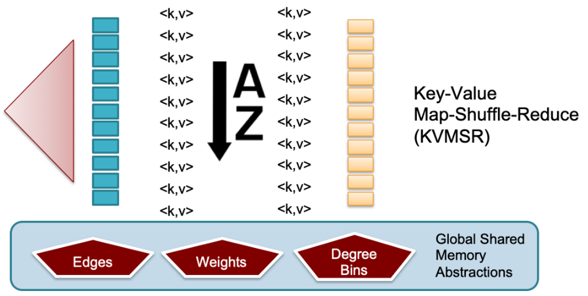

Figure 3: UpDown Software: KVMSR organizes parallelism over mutable data abstractions. UDWeave language is used to write them all.

2.2 KVSMR: Coordinating Massive-Scale Parallelism

KVMSR is used to organize large-scale parallelism, required to fill the UpDown system's 33 million lanes. KVMSR (short for key-value map-shuffle-reduce) is based on the MapReduce paradigm, which has been demonstrated to be an expressive, flexible model for parallelism [5, 11, 45]. KVMSR extends the cloud MapReduce model by allowing computation over a shared global address space, including sharing of mutable state (memory, complex data abstractions) as shown in Figure 3. Programmers use KVMSR and UDWeave to express the computational structure (parallelism) and computation binding as pictured in Figure 1. A more complete description of KVMSR can be found in [42].

It builds on the MapReduce paradigm, decoupling the expression of computation from the management of compute resources. KVMSR exploits the idea that parallelism can be exposed and controlled through user-defined keys. In KVMSR, the user program defines `kv_map()` and `kv_reduce()` events. These are applied to each element in the key-value maps (input, intermediate), expressing data parallelism across all the keys in each map. This can involve massive parallelism in a graph program, where the size of these maps is often a large set of vertices or edges.

KVMSR executes on a machine with a global address space with shared global data structures that can be accessed directly from within `kv_map()` or `kv_reduce()` tasks (see Figure 3). This means, far more complex computational structures can be expressed within a single MapReduce iteration. Compare this to cloud-based MapReduce systems, which rely on coarse-grained tasks and prohibit shared memory access. Examples of shared global data structures range from mutable arrays (such as PageRank values, BFS frontiers, and triangle counts) to scalable data abstractions (including hash tables, histogram bins, and multi-producer/multi-consumer queues), and more. Even basic vertex and edge structures are handled in this manner for ingestion/construction, leveraging fine-grained locking for high-performance streaming graph input.

Listing 3 illustrates KVMSR for PageRank (see Section 4.1). The user logic for PageRank is expressed in the `kv_map` and `returnRead` events.

```
struct Vertex {
    uint64_t id, degree, *neighbors;
    double pr_value; }

thread PageRankWorker {
  // persistent across events
  uint64_t degree, prUpdate, loadedNeighbors;

  event kv_map(uint64_t id, uint64_t degree_, uint64_t *
       neighbors, double pr_value, uint64_t *vertexAddr){
    loadedNeighbors = 0; degree = degree_;
    double out_pr_val = pr_value / degree;
    for(uint i=0; i<degree; i=i+8) {
      uint64_t num_neighbors = min(8, degree - i);
      send_dram_read(neighbors, num_neighbors, returnRead
          );
      neighbors += num_neighbors * sizeof(uint64_t); }
    prUpdate = pr_value / degree;
    yield; }

  event returnRead(uint64_t n0, n1, ..., n7) {
    send_event(kv_map_emit, n0, prUpdate);
    send_event(kv_map_emit, n1, prUpdate);
    ...
    send_event(kv_map_emit, n7, prUpdate);
    loadedNeighbors += 8;
    if(loadedNeighbors == degree) {
        uint64_t evw = evw_update_event(CEVNT,
             kv_map_return);
        send_event(evw, IGNRCONT); }
    yield; }

  event kv_reduce(uint64_t vertexID, double
       updatedPRValue) {
     // write update to fetch-n-add cache for v.pr_value
  } }
```

Listing 3: PageRank definitions for KVMSR, written in UDWeave

`kv_map` initializes computing the outgoing PageRank contributions based on edge weights (read from DRAM in chunks of eight neighbors). Responses are handled by the `returnRead` event, which receives the neighbor identifiers and emits key-value pairs using `kv_map_emit`. The intermediate map defined by these emit operations is connected to the reducers by the KVMSR library. The `kv_reduce` operations combine these updates into each vertex, completing the PageRank iteration (not shown in the listing).

This program expresses task parallelism proportional to the vertices (`kv_map`) and edges (`kv_reduce`). It also expresses memory parallelism proportional to the number of edges. Minimal ordering is enforced for correctness. Memory responses are counted in `loadedNeighbors` to detect completion. KVMSR tracks termination of the map and reduce phases, and thus the application must signal this completion by invoking `kv_map_return` to enable KVMSR to retire completed map threads.

The KVMSR application program described thus expresses no detailed binding of computation to the machine. It only expresses the parallelism. The KVMSR implementation maps it efficiently into threads. In the following sections, we describe how applications express computation binding and data placement, the other two dimensions from Figure 1.

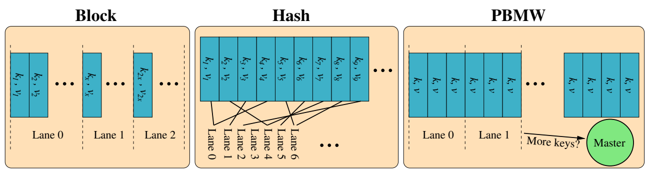

Figure 4: KVMSR creates tasks for each key, and can be used to express computation binding in a large-scale parallel UpDown program. Block, Hash, PBMW, etc.

2.3 Computation Binding

UpDown threads (tasks) are initiated by events (messages), each sent to a `networkID` that specifies a particular `<node, lane>`. These messages can create a new thread or specify an existing thread. `networkID` can be computed by an application program at runtime, allowing software to easily control the mapping of tasks to compute resources by using the `networkID` in messages to distribute work in UpDown.

Because the KVMSR framework is the primary approach to organizing large-scale parallelism, it provides several methods for distributing work based on a key-value map. These predefined methods can be extended with user programs to bind computation locations (lanes or nodes).

- **Block:** Lanes given an equal, contiguous portion of keys
- **Hash:** Each key is hashed to distribute across lanes
- **Partial-block and Master-Worker: (PBMW)** Lanes are given a block of keys; ask master for more
- **Data-driven:** (future) Each task executes on the same node as a key-value pair, provided by the system querying the hardware address translation.

Each KVMSR invocation targets a set of lanes, and the computation binding reflects a mapping of the `kv_map()` tasks on the kvmap keys to the lanes as shown in Figure 4. The kvmap keys are produced by a parallel iterator abstraction, of which appropriate start points in the kvmap are passed to each lane in the KVMSR set. By default, KVMSR uses **Block** for `kv_map()` tasks and **Hash** for the `kv_reduce()` tasks. However, any application can override and provide its own computation binding for a particular KVMSR.

An application can also use UDWeave to directly control task mapping by computing networkIDs directly. For example, the following pseudocode computes a hashed mapping:

```
LaneID = (hash(key) % NRLanes) + 1stLane
```

2.4 Data Placement

UpDown's global address space implements flexible two-level mapping to the memories distributed across the parallel nodes. This system provides a small translation state (for efficient hardware implementation) and flexible application control (enabling locality optimization) as described in [40, 41]. Here we describe the user API, DRAMmalloc, that allows applications to lay out contiguous virtual address space regions across distributed physical memories.

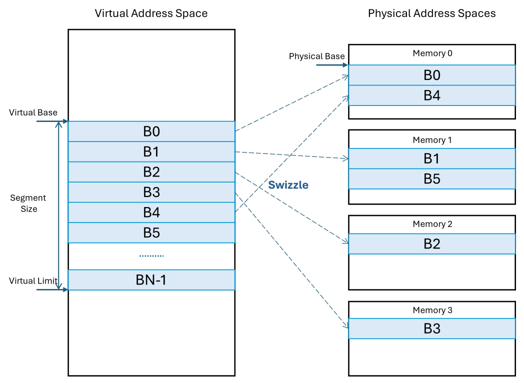

Figure 5: Each allocated virtual memory region is mapped to distributed physical node memories. Contiguous address blocks in the virtual address space are mapped to blocks in distributed memories in a cyclic manner.

Each DRAMmalloc call returns a region of memory (virtual address space), similar to the `malloc()` function. However, the layout of the region can be controlled by the application, according to the well-known block-cyclic [13, 22] distributions as illustrated in Figure 5. A call to DRAMmalloc is defined as follows:

`void* DRAMmalloc(size, 1stNode, NRNodes, BS)`

`size`: total number of bytes to be allocated
`1stNode`: node on which the allocation starts
`NRNodes`: node number for cyclic distribution (power of 2)
`BS`: block size for distribution (power of 2, >4KB)

Distribution is done on a per-node basis; each node has a 4KB interleaved, contiguous physical address space.

This layout is encoded into a single translation descriptor supported natively by the UpDown hardware. This has two benefits: 1) a much smaller number of descriptors is required for a typical program (e.g., 2-4 for our benchmarks), and 2) efficient hardware implementation achieves translation (and protection) with no overhead for software.

Given the DRAMmalloc() parameters, UpDown encodes a translation descriptor (swizzle mask) that encodes the parameters. The UpDown hardware uses it to convert virtual addresses into a physical node number (PNN) and an address within that node (offset). The physical node number and offset are computed as follows:

PNN = size / BS / NRNodes
Offset = size % BS % NRNodes

This computation produces a block-cyclic layout across the physical nodes specified in the DRAMmalloc call. Common examples of allocations are shown in Table 1.

3 The UpDown Architecture

The UpDown system was codesigned for graph applications to achieve supercomputer-level scalable performance [4, 31–33, 41, 43]. Key UpDown system capabilities include: global address space, extraordinary communication performance (efficient 64-byte messages, low-latency of 0.5 microseconds), high communication bandwidth (4 terabytes/second node injection, and 32 petabytes/second bisection bandwidth), high memory bandwidth (system: 153 petabytes/second, node: 9.4 terabytes/second), and efficient execution of fine-grained parallelism (10-100 instructions). These capabilities enable UpDown programs to exploit vertex and edge parallelism directly [39, 43], and to write programs assuming a nearly flat memory space (performance). For the global system view, see Figure 6. UpDown's design is supported by the IARPA AGILE program, an ambitious effort to create new building blocks with breakthrough capability for graph computing [1].

Table 1: Common examples of DRAMmalloc() parameters

Distribution	Description
(.,0,16384,4096)	Distribute cyclically over the entire machine in blocks of size 4 KB
(.,0,1024,4096)	Distribute cyclically across the first 1K nodes of the machine in blocks of size 4 KB
(4TB,0,1024,4GB)	Contiguous 4GB regions on each of the first 1K nodes of machine
(4TB,4K,8K,1MB)	Distribute cyclically across the middle 8K nodes of the machine in 1MB blocks (each node gets 512 x 1MB blocks = 512MB)

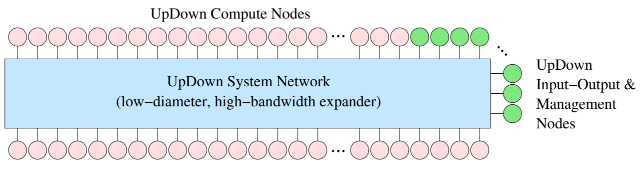
UpDown Compute Nodes
UpDown System Network
(low–diameter, high–bandwidth expander)
UpDown Input–Output & Management Nodes

Figure 6: UpDown System: 16,384 Compute Nodes connected by a diameter-3, high-bandwidth Polarstar Network [19].

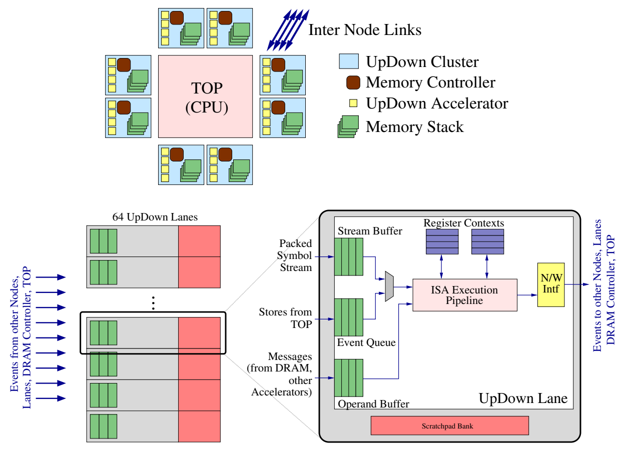

Figure 7: Each UpDown Node (top) has 32 accelerators; each accelerator has 64 lanes, designed for event-driven execution (bottom)

3.1 Node-level Design and Fine-grained Parallelism

Each node in the system consists of 32 UpDown accelerators, 8 stacks of HBM3e (4 accelerators next to each), and a rich connection to the system network as in Figure 7 (top). Each accelerator consists of 64 lanes, which are 2 GHz MIMD compute engines, executing in event-driven fashion as shown in Figure 7 (bottom). UpDown's efficient mechanisms enable fine-grained parallelism; events can execute 10-100 instruction tasks efficiently. The high-bandwidth memory provides ample memory bandwidth, enabling high performance on applications with little data reuse. The UpDown systems' 64 lanes/accelerator produces 2048 lanes per node, and 33 million lanes in a 16K-node machine. The ability to expose fine-grained parallelism enables high performance on graph applications, making UpDown capable of fully exploiting vertex and edge parallelism.

Table 2: Lane Operation Costs (2GHz clock)

Operation	Instructions	Cost (Cycles)
Thread Create	0	0
Thread Yield	1	1
Thread Deallocate	1	1
Load/Store (Scratchpad)	1	1
Send Message	1	1-2
Send DRAM	1	1-2

3.2 Application Challenges in Programming UpDown

The UpDown system architecture presents several unique challenges for application programming – outlined below:

- 33 million parallel lanes, and no caches, critical need to generate maximum parallelism to manage DRAM latency and cross-machine latency
- localization of data access for latency (7:1) and bandwidth (3:1) of memory access
- needs for extremely good load balance, over millions of lanes to get good speedups
- controlling dynamic parallelism to match the hardware parallelism. Such dynamism is essential to maximize parallelism.

There are several issues, common in mainstream supercomputers, that are eased on the UpDown supercomputer. UpDown programmers need not spend effort on:

- Tuning task sizes for efficient execution. On CPUs and GPUs, tasks typically involve millions of instructions for optimal efficiency. On UpDown, parallel tasks of 10-100 instructions execute with high efficiency, enabling the expression of applications with a natural parallelism structure.
- Aggregating communications and data sharing to make transfers large enough for efficient execution. Because cores used in CPUs and GPUs cannot tolerate the long DRAM or system memory latencies, they rely on high cache hit rates for high performance. On UpDown, non-blocking memory accesses and multithreading allow robust latency tolerance. Combined with plentiful memory bandwidth, this obviates the need for data caches.

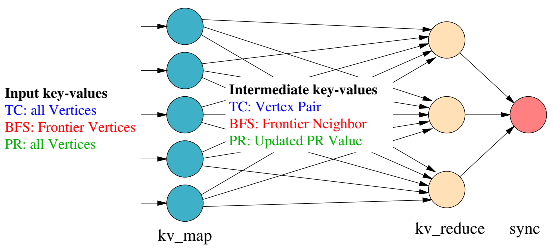

Figure 8: How several program examples employ KVMSR

4 Programming Examples

4.1 PageRank

PageRank (PR) on irregular real-world graphs is challenging due to its fine-grained parallelism and unpredictable communication and computation structures [10, 17, 20, 25, 29]. We implemented a simple push-based PageRank algorithm that exploits edge-level parallelism (each edge propagation is a task), producing maximum parallel scalability and extremely irregular parallelism. This implementation runs on UpDown and processes RMAT graphs in neighbor list format.

4.1.1 Data Structures and Placement. The PageRank implementation has two global data structures: the vertex array and the neighbor-list array. The vertex array stores vertices and associated metadata, as defined in Listing 3. The neighbor-list array contains the destination vertices for all vertices in the graph. Each vertex structure in the vertex array includes a `neighbors` pointer, an offset into the neighbor list, a `degree` (count), and a page-rank value, `pr_value`. The vertex and neighbor-list arrays are each distributed using `DRAMmalloc(size,0,NRnodes,32KB)` across the machine, a simple default spreading that ensures high bandwidth access but makes no attempt to optimize data locality.

4.1.2 Computation Structure and Binding. We use the KVMSR framework to express parallelism across vertices by defining `kv_map()` functions as shown in Figure 8. KVMSR implements the data-parallelism by logically executing a parallel task for each `kv_map(<k, v>)`. To distribute these tasks (millions to billions) to UpDown lanes, we use the default Block binding scheme (see Section 2.3).

As shown in Figure 8, Each `kv_map` event uses the neighbor list pointer to read vertex neighbors, issuing DRAM read requests for them in groups of eight neighbors. as in listing 3). For each neighbor read, the `kv_map()` task emits into the intermediate KV map a tuple `<targetVertex, increment>`. This is done by sending the `kv_map_emit` event. This send is asynchronous with no response, so each generates additional parallelism. These tasks exhibit parallelism proportional to the total number of edges in the graph (billions to trillions).

In the normal map-reduce fashion, each of these intermediate KV map tuples produces a `kv_reduce()` task, which performs computation as specified in Listing 3. These tasks accumulate the contributions to the vertex PageRank value, using a fetch-n-add() operation[1].

[1]The fetch-n-add() operation is implemented in UDWeave; it is not a hardware primitive. The implementation caches the value in the scratchpad for high performance and provides atomicity.

To distribute these reduce tasks to UpDown lanes, we use the default Hash binding scheme (see Section 2.3) to ensure good load balance.

4.1.3 Discussion. A few novel things are worth emphasizing. The KVMSR implementation of PageRank exposes full vertex (kvmap) and edge (kvreduce) parallelism. This enables scaling to full supercomputer size UpDown machines (33M lanes), with relatively small graphs. This is because a significant amount of parallelism is exposed, and the special UpDown architecture features enable the efficient exploitation of fine-grained tasks of 10-100 instructions (see [4]). We used default computation bindings for KVMSR (Block, Hash), but these can all be customized under user control. The KVMSR framework transparently converts the flat - per kvmap or kvreduce - parallelism into groups of tasks and threads, matching them to the machine's memory latency, physical lanes, and physical thread resources without any application programmer effort. This guarantees that the machine is fully utilized (but not overutilized) by ensuring enough thread parallelism (utilize compute units) and memory parallelism (tolerate latency and utilize memories), and reducing programming effort. This is possible because there are no complicating structures such as hardware caching and coherence.

4.2 Breadth-First Search

We use a push-based algorithm for Breadth-First Search (BFS). Starting with a frontier containing the seed vertex, we perform a series of rounds. In each round, we check the neighbors of all vertices in the frontier. All the unmarked neighbors are marked and added to the frontier for the next round. If the new frontier is empty, the algorithm terminates; otherwise, the next round begins with the new frontier.

4.2.1 Data Structures and Placement. The BFS implementation uses three global data structures; two are similar to PageRank (vertex array, neighbor list) with extra fields for each vertex to trace the distance and resulting BFS tree. The third structure is the frontier, which holds the state of the BFS search.

The vertex array and neighbor list arrays are allocated in global memory using the function `DRAMmalloc(size,0,NRnodes,32KB)` and distributed evenly across the system to ensure high-bandwidth access. For data locality, the frontier is allocated differently, as there is a local frontier for each accelerator. That is supported by a contiguous block of memory for the accelerator's local frontier within each node. Thus, it uses a `DRAMmalloc(size,0,NRnodes, size/NRnodes)` allocation that provides a contiguous section of virtual addresses (memory) to be used for the local frontier. This enables data locality for reading the current frontier and writing the next frontier. Within the node, each accelerator uses a contiguous chunk of the allocation to store its part of the frontier for each round.

4.2.2 Computation Structure and Binding. Each round of BFS is implemented using KVMSR. The `kv_map()` tasks are invoked on the sections of the current frontier (one per accelerator). Thus, one `kv_map()` task is created for each accelerator in the machine (32 per node).

The `kv_map()` tasks then use UDWeave mechanisms to organize the accelerators (64 lanes) to process the local portion of the frontier. A master creates a set of subtasks, which are then distributed to other lanes in the accelerator. Each of these subtasks computes the BFS step on a set of vertices in the frontier, and then emits the unmarked neighbors to the `kv_reduce()` tasks to write into the next frontier. The `kv_reduce()` tasks are mapped onto the lanes using the Hash computation binding (default for KVMSR). The vertices are inserted into the frontier for the next round. Applying the hash ensures that the local frontier structures have good load balance, enabling good speedup.

4.2.3 Discussion. BFS highlights several departures from the pure, flat data parallelism used in PageRank. BFS uses each accelerator as a worker for KVMSR, mapping over the local partitions of the frontier. Then, within each of those, it employs a local master-worker scheme to load-balance and coordinate processing of the frontier efficiently. This demonstrates the flexibility of KVMSR+UDWeave in choosing a parallel task size for management by KVMSR and in building custom local parallelism and coordination schemes.

BFS also highlights the flexibility of the data placement system. It not only spreads data for high bandwidth and scalable performance, but the same interface also easily describes a set of local partitions used for the local, per-accelerator frontiers.

4.3 Triangle Counting

We compute triangle count (TC) exploiting vertex parallelism, as our graph representation consists of a vertex and a neighbor list. Conceptually, we compute in parallel over the vertices, enumerating the vertex pairs that could form triangles – vertex parallelism (millions to billions). For each of the vertex pairs (edges), we then compare their neighbor lists, which creates parallelism proportional to (# of edges) * (average degree). However, we do not need this much parallelism, so we constrain it to the number of edges, serializing each neighbor list comparison.

4.3.1 Data Structures and Placement. The graph is represented using the same data structures as PageRank: two global data structures, the vertex array and the neighbor list array. Each is distributed using `DRAMmalloc(size,0,NRnodes,32KB)` across the machine, a simple default spreading that ensures high bandwidth access but makes no attempt to optimize data locality.

4.3.2 Computing Structure and Binding. The `kv_map()` tasks are run on all vertices, and each task enumerates the vertex pairs $\langle v_x, v_y \rangle$ that are connected by an edge. To avoid double-counting, we only enumerate the pairs for which $x > y$. The `kv_reduce()` tasks then intersect the neighbor lists for v_x and v_y. For each v_z that appears in both lists, a triangle, v_x, v_y, v_z, is formed, so the count is incremented.

The `kv_map()` tasks are mapped with the default Block computation binding. The `kv_reduce()` operations on connected vertex-pairs (computes neighbor-list intersections) use a Hash binding using a combination of the vertex names.

4.3.3 Discussion. In another version of the TC code, the `kv_map()` functions are distributed with PBMW, which is more robust to larger work skews across blocks. PBMW allows the latter work to be dynamically distributed, as the early-completing lanes request additional work from the master, improving mapping efficiency

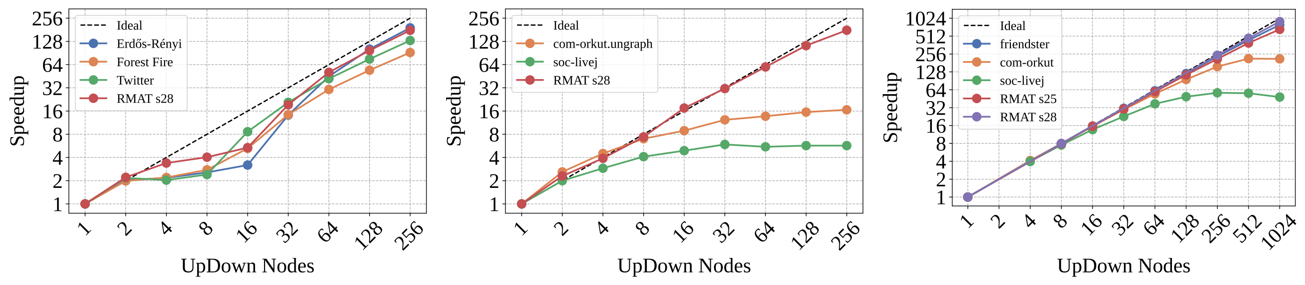

Figure 9: KVMSR+UDWeave Programs (PR, BFS, TC) achieve excellent scalability and high absolute performance.

and better utilization across lanes. However, as the implementation of TC was improved, we discovered that this secondary balancing was not required.

To get the best performance, the `kv_reduce()` functions, written in UDWeave, are optimized for performance. In an early version of the TC program, neighbors of one vector are loaded into scratchpad memory to capture reuse, then streamed to other neighbor lists for comparison. However, this optimization for reuse limited the flexibility of load balancing. Therefore, a second version of TC streams both neighbor lists in the reduce function, consuming more memory bandwidth but improving load balance. This is a net win because on UpDown, TC is compute-limited due to UpDown's extraordinary memory bandwidth.

4.4 Other Examples

We have programmed many other examples, including graph neural networks, mixture of experts, network streaming graph analytics, high-speed ingestion, and more [1, 32, 37, 44, 46]. These further examples illustrate the power of KVSMR+UDWeave in conveniently expressing parallel programs.

5 Performance Results

5.1 Methodology

We evaluate the UpDown system using Fastsim, a scalable simulation framework that delivers cycle-accurate modeling of UpDown accelerators through instruction-level simulation, while employing streamlined capacity and latency models for DRAM and system networking components. To ensure accuracy, we calibrated Fastsim against Gem5sim [23] using 1-4 node configurations. Gem5sim provides comprehensive, cycle-accurate models that include detailed UpDown accelerators, TOP cores (based on the x86 architecture), DRAM memories (DRAMsim3 [21]), and both node-level and system-level interconnects. While Gem5sim offers complete cycle-accurate fidelity, we employ Fastsim for its superior performance characteristics, enabling simulation of large-scale deployments with 1,024 UpDown nodes (exceeding 2 million lanes) and realistic application dataset sizes.

5.2 Large-scale Performance Results

The UpDown architecture has been extensively evaluated [4, 33, 37, 39, 44]. However, here we present a few highlights that demonstrate the KVMSR+UDWeave framework can achieve excellent strong scalability and high absolute performance. In all cases, the graphs studied have a few million vertices (1-10), and one as large as 500M. These are much smaller than the trillion-vertex graphs used in Graph500 on supercomputers [10]. Thus, our studies demonstrate extreme strong-scaling.

5.2.1 PageRank. In Figure 9 (left), we present the PageRank performance on several SNAP and RMAT [8] graphs. This algorithm employs a vertex splitting transformation, which transforms the graph to a maximum degree of 1024, yet yields the correct result for the original graph. Testing on scale-25 and scale-29 RMAT graphs shows excellent strong scaling with speedups exceeding 900 for 1024 nodes (corresponding to 2M UpDown lanes, physical parallelism). For the Scale-28 Erdős-Rényi graph, UpDown achieves maximum performance of 39,617 giga-updates/second on 512 nodes, representing a 12,188× improvement over the best-available Perlmutter supercomputer performance [15, 39].

5.2.2 Breadth-First Search. In Figure 9 (center), we present the BFS performance on Scale-28 RMAT graphs and several SNAP graphs. This push algorithm scales well to 256 nodes (0.5M lanes physical parallelism), achieving a speedup of slightly under 256. On the Scale-28 RMAT graph, UpDown achieves a performance of 35,700 giga-traversed edges per second on 512 nodes (not shown), exceeding the performance of a 4,096-node NVIDIA EOS GPU cluster at 1/12th the power [10, 39].

5.2.3 Triangle Counting. In Figure 9 (right), we present the TC performance on a variety of real-world graphs and RMAT scale 25 and 28 graphs. For the larger graphs of 30-250M vertices, the algorithm has excellent scaling to 1024 nodes (2M lanes physical parallelism), achieving a speedup of 899x over a single node. Performance is excellent, and is compared in detail to other systems in [37].

5.2.4 Ingestion and Partial Match. The MapReduce paradigm is capable of orchestrating parallel and distributed computations, so it is no surprise that KVMSR+UDWeave can be used to express ingestion to support graph analytics that parse JSON-encoded, compressed streams and integrate them into a running analytics program's data structures and ongoing analytics computation (see Figure 10). The data is read as a parallel file, and KVMSR maps over parallel blocks. Inside, `kv_map` deals with variable-size records that can span block boundaries, accessing across blocks, and synchronizing and ordering as necessary. Such access would be impossible in a cloud map-reduce formulation. TFORM enables fast parsing, using an extended transducer model [28]. Graph data structure inserts utilize scalable atomic operations on the graph data structures,

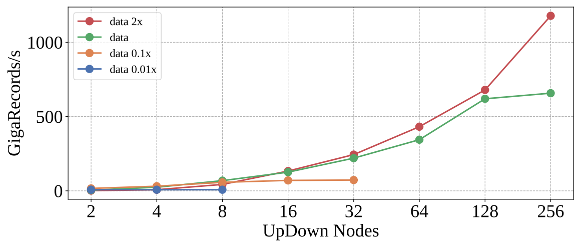

Figure 10: Ingestion: TFORM and KVMSR are used to load, parse a parallel file, and insert it into a graph data structure. Each record is 64 bytes, so 1200 GigaRecords/second is 76.8 TB/s.

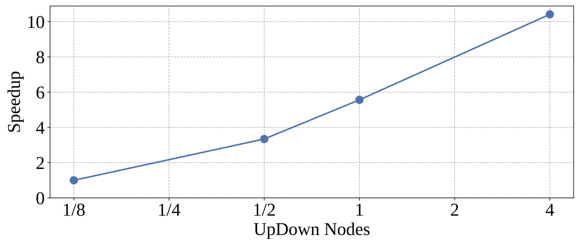

Figure 11: Partial Match: streaming query performance on a stream of edge and vertex updates. The KVSMR program scales, reducing latency.

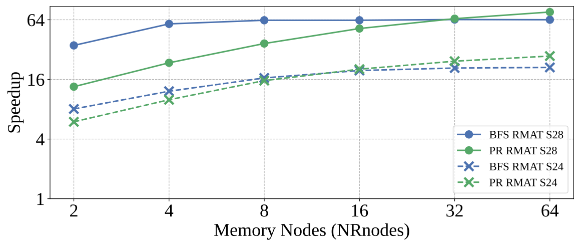

Figure 12: Performance impact of NRnodes in DRAMmalloc() allocation of graph structure (fixed 64 compute nodes, RMAT)

leveraging UpDown's fast synchronization capabilities. Overall, KVMSR+UDWeave achieves data parsing and integration into ongoing computation at a rate of 76.8 TB/s (256 nodes). The application can scale higher, but this already exceeds the external network connection of any existing data center and is one-third of the Internet's bisection bandwidth.

Another novel scenario is the partial match streaming network application built on these ingestion capabilities. Here, the records are received from the network and inserted into the graph. They are processed against a set of registered patterns. The objective is to incrementally evaluate the patterns and identify matches as rapidly as possible! Latency is the metric. In Figure 11, we illustrate the performance of a KVMSR+UDWeave solution, based on scalable hash tables (SHT). The implementation is scalable, adding and matching against small and large graphs, and latency can be decreased (speedup) by adding compute resources.

Table 3: Applications use of UDWeave and the KVMSR library

Application	**UDWeave**	**KVMSR**	**Notes**
BFS	Both	Both	Multiple versions
PageRank	Y	Y	also kvcombine cache
TriangleCount	Y	Y	Multiple versions
Bucket Sort	N	Y	kvmap
GNN (genFeatures)	Y	Y	doAll using kvmap
GNN (integrate)	Y	Y	kvmap
Multihop Ingestion	Y	Y	doAll, kvmap
Multihop Reasoning	Y	Y	doAll, kvmap
Exact Match	Y	Y	doAll using kvmap
Partial Match	Y	Y	doAll using kvmap
Graph Compaction	Y	Y	doAll, kvmap
Construct Sequences	Y	Y	doAll, kvmap
Abstractions			
Scalable Hash Table	Y	n.a.	
Parallel Graph	Y	n.a.	Uses two SHT's
SHMEM Library	Y	Future	as defined in [38]
TFORM Tool	n.a.	n.a.	Sub-byte encode/decode

5.3 Impact of Data Placement

The ability to flexibly control data placement – at runtime – and with no software translation overhead is a powerful tool for optimizing program performance. To illustrate this, we ran PR (graph) and BFS (frontier) with several different data layouts (refer to Figure 5). Only a single number was changed in a DRAMmalloc() call to create each layout! For both cases, the memory parallelism (striping) varies from 2 to 64 nodes (16-fold bandwidth). For PR, changing the data placement of the graph structure increases performance by up to 4-fold (s28) due to increased memory bandwidth, and benefits taper off as the memory bottleneck is eased, and other parts of the system limit performance. For BFS, the benefits exhibit the same trend, albeit less pronounced.

5.4 Programmability Data

5.4.1 Use of KVSMR+UDWeave. We document the use of the KVSMR+UDWeave abstractions across a set of applications in Table 3. Both were used for nearly every application kernel and abstraction that was built for the IARPA AGILE evaluation [2]. Those indicated as `kv_map()` used KVMSR, but with the reduction providing only synchronization, and those using `doAll()` used KVMSR indirectly. All of these applications reach massive parallelism (thousands to millions). And without these tools, writing such complex applications would have been impossible!

At the bottom of Table 3, we list several important abstractions built for scaling graph applications. Notable is a SHMEM library that leverages UpDown's translation-supported data placement to simply describe complex data rearrangement. Also, the TFORM library implements transformer-driven sub-word and sub-byte recoding capabilities [28].

5.4.2 LoC Metrics. For the AGILE programmability evaluation, we computed the lines-of-code (LoC) metric for a range of kernels, comparing them to a high-level notation (SHAD [6, 7]) and MPI+X (C or Fortran). Assembly code is normalized at 1/5 LoC. Results are

Table 4: Cross-dependences in Parallel Programming Models

Approach	Parallel Code	Computation binding	Data placement
KVMSR+UDWeave (this work)	**Independent**	**Independent**	**Independent**
MPI+OpenMP	To binding, placement	To placement (owner computes)	To binding, code
PGAS (UPC/Co-array Fortran)	Independent	To placement (owner computes)	Independent
MapReduce (Spark)	Independent	To placement (owner computes)	Independent
MapReduce (Hadoop)	To data (no placement)	To data (no placement)	None

Table 5: Code Sizes (LoC) for UpDown (UD) Programs compared to SHAD. UpDown Abstraction/Libraries (LoC).

	UD	SHAD
ISBs		
PR	218	–
BFS	226	500
TC	312	289
Workflows		
WF1, K2	997	131
WF2, K1	782	75
WF2, K2	1,817	546
WF3, K3	1,481	445
WF4, K4	187	177
SHAD Libs		30K-256K
UD Libs	6,020	

Abstraction	UD
Data Abstractions	
Scalable Hash Table	4,764
Parallel Graph Abstraction	170
Compute Abstractions	
KV map-shuffle-reduce	1,586
do_all (uses KVMSR)	33
Scalable Global Sort	158
SHMEM (put/get, reductions)	1,914
Memory Abstractions	
spMalloc (scratchpad malloc)	83
DRAMmalloc (global malloc)	52
Combining Cache (fetch&add)	232

summarized in Table 5 (left). The results in Table 5 (right) document the use of UDWeave to build various software utilities needed for UpDown programs of any scale. These include the complexity of the compute and data abstractions as well as memory management libraries.

Our results show that KVMSR+UDWeave enabled the UpDown team to build complex graph benchmarks and workflows. Further, powerful UpDown libraries can be implemented with high productivity. The LoC required to implement powerful abstractions such as scalable hash tables (SHT) and key-value-map-shuffle-reduce (KVMSR) is moderate and supports various parallel applications.

6 Discussion and Related Work

The KVMSR+UDweave is a decided contrast to the MPI+X model, supporting fine-grained parallelism and flexible parallel computation structures. Perhaps the most significant difference is that the expression of computation is orthogonal to its placement on the machine (binding) and the data layout. Generally distributed address space models (e.g., MPI+X) intermix computation structure expression with data layout, with a hard distinction between local data access and messaging [14, 18]. To a degree, most MPI+X models also bind computation to ranks at least if not physical resources, coupling the three dimensions (see Table 4). This approach can work well for regular applications [16] and moderately irregular structures. Still, it complicates program expression, change, and optimization, particularly for applications that use complex pointer-based data structures.

Walking through the table, PGAS (and APGAS) models have been developed to ease programming efforts, providing a global address space [26, 27, 30, 47]. These models can provide the ability to change data placement without requiring radical code changes, but the single-program multiple data (SPMD) and owner-computes execution models couple computation binding to data placement. This is due to the static parallelism approach of SPMD that specifies parallelism against a fixed array of workers, a complex proposition for irregular, graph applications in which the available parallelism is often discovered at runtime as the graph structure is traversed. In contrast, UDWeave and UpDown provide a dynamic parallelism model, where tasks are created as the parallelism is discovered, and they can be flexibly bound by the application program to computational resources (e.g., nodes and lanes).

Our KVSMR+UDWeave framework is more capable than cloud map-reduce because the map and reduce functions operate over a shared mutable address space [11, 45]. They can have full sharing, synchronization, and share data structure interactions. Examples include BFS frontiers, triangle counters in K-Truss, graph structure updates in ingestion, scalable hash tables, producer-consumer queues, match pattern state in partial match, and more. Functional variants of MapReduce can share state, but generally cannot generate side effects. Our KVSMR+UDWeave framework enables arbitrary sharing and coordination of tasks (e.g., master-worker within BFS or GNN parameter server), making it adaptable to various application architectures. Spark utilizes data decompositions and SPMD execution, making it more similar to PGAS models, but provides less control over data placement.

7 Summary and Future Work

We have described the KVMSR+UDWeave model with orthogonal specification of compute, binding, and placement. Our KVSMR+UDWeave programming model exploits UpDown's novel hardware features extensively, including architectural support for fine-grained tasks and shared global address space. The ability to exploit tasks for individual vertex or edge operations exposes million and billion-fold parallelism that enables extreme strong scaling. Simply put, UpDown can achieve excellent speedups and exceeds the performance of supercomputers on small and large problems. Cerebras has highlighted the power of such a capability for practical HPC [35, 36]. A broader question is whether KVMSR+UDWeave could be impactful on other scalable hardware platforms, perhaps easing programming on systems with a portion of the UpDown hardware features.

Acknowledgments

Supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing

Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082. The views and conclusions contained herein are those of the authors. They should not be interpreted as representing the official policies or endorsements, expressed or implied, of the ODNI, IARPA, or the U.S. Government.

References

[1] 2022. Advanced Graphic Intelligence Logic computing Environment. https://www.iarpa.gov/research-programs/agile.

[2] 2022. AGILE Program Workflows and Benchmarks. https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Program_Workflows_FINAL.pdf.

[3] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT press.

[4] Andrew A. Chien, et. al. 2025. UpDown: A Supercomputer Co-designed for Scalable Graph Processing. *under review* (2025). Available from http://people.cs.uchicago.edu/~aachien/lssg/research/10x10/UpDown_System_Paper___TPDS_submittedv2.pdf.

[5] Various Authors. 2025. *Ocaml-5.3 Reference Manual.* INRIA. Available from https://ocaml.org/manual/5.3/index.html.

[6] Vito Giovanni Castellana and Marco Minutoli. 2018. SHAD: The Scalable High-Performance Algorithms and Data-Structures Library. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* 442–451. https://doi.org/10.1109/CCGRID.2018.00071

[7] Vito Giovanni Castellana, Burcu O. Mutlu, Ian Di Dio Lavore, Jesun Firoz, Katherine Wolf, Marco Minutoli, and John Feo. 2024. Custom Accessors: Enabling Scalable Data Ingestion, (Re-)Organization, and Analysis on Distributed Systems. In *2024 IEEE International Conference on Big Data (BigData).* 189–198. https://doi.org/10.1109/BigData62323.2024.10825020

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. [n. d.]. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM).* 442–446. https://doi.org/10.1137/1.9781611972740.43 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43

[9] Andrew A Chien and William J Dally. 1990. Concurrent aggregates (CA). In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming.* 187–196.

[10] Community. [n. d.]. The Graph 500:large scale graph processing benchmarks. https://graph500.org/

[11] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[12] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures.* ACM, Vienna Austria, 393–404. https://doi.org/10.1145/3210377.3210414

[13] Jack Dongarra and Antoine Petitet. 1995. ScaLAPACK tutorial. In *International Workshop on Applied Parallel Computing.* Springer, 166–176.

[14] Jack J Dongarra, Steve W Otto, Marc Snir, David Walker, et al. 1995. An introduction to the MPI standard. *Commun. ACM* 18, 11 (1995).

[15] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. 2023. Highly Scalable Large-Scale Asynchronous Graph Processing using Actors. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW).* 242–248. https://doi.org/10.1109/CCGridW59191.2023.00049

[16] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. 1988. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems.* Prentice-Hall, Inc., USA.

[17] David F. Gleich. 2015. PageRank Beyond the Web. *SIAM Rev.* 57, 3 (2015), 321–363. https://doi.org/10.1137/140976649

[18] William Gropp, Ewing Lusk, Anthony Skjellum, and Rajeev Thahur. 1999. Using MPI—2nd Edition: Portable Parallel Programming with the Message Passing Interface.

[19] Kartik Lakhotia, Laura Monroe, Kelly Isham, Maciej Besta, Nils Blach, Torsten Hoefler, and Fabrizio Petrini. 2023. PolarStar: Expanding the Scalability Horizon of Diameter-3 Networks. *arXiv preprint arXiv:2302.07217* (2023).

[20] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1, Article 1 (July 2016), 20 pages. https://doi.org/10.1145/2898361

[21] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator. *IEEE Computer Architecture Letters* 19, 2 (July 2020), 106–109. https://doi.org/10.1109/LCA.2020.2973991 Conference Name: IEEE Computer Architecture Letters.

[22] David B Loveman. 2002. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications* 1, 1 (2002), 25–42.

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. https://doi.org/10.48550/arXiv.2007.03152 arXiv:2007.03152 [cs].

[24] Junnan Lu and Alex Thomo. 2016. An experimental evaluation of Giraph and GraphCHI. In *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* (Davis, California) *(ASONAM '16).* IEEE Press, 993–996.

[25] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10).* Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[26] Matthias Mayr, Alexander Heinlein, Christian Glusa, Siva Rajamanickam, Maarten Arnst, Roscoe Bartlett, Luc Berger-Vergiat, Erik Boman, Karen Devine, Graham Harper, Michael Heroux, Mark Hoemmen, Jonathan Hu, Brian Kelley, Kyungjoo Kim, Drew P. Kouri, Paul Kuberry, Kim Liegeois, Curtis C. Ober, Roger Pawlowski, Carl Pearson, Mauro Perego, Eric Phipps, Denis Ridzal, Nathan V. Roberts, Christopher Siefert, Heidi Thornquist, Romin Tomasetti, Christian R. Trott, Raymond S. Tuminaro, James M. Willenbring, Michael M. Wolf, and Ichitaro Yamazaki. 2025. Trilinos: Enabling Scientific Computing Across Diverse Hardware Architectures at Scale. arXiv:2503.08126 [cs.MS] https://arxiv.org/abs/2503.08126

[27] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. 1994. Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing* (Washington, D.C.) *(Supercomputing '94).* IEEE Computer Society Press, Washington, DC, USA, 340–349.

[28] Marziyeh Nourian, Tri Nguyen, Andrew A. Chien, and Michela Becchi. 2022. Data Transformation Acceleration using Deterministic Finite-State Transducers. In *2022 IEEE International Conference on Big Data (Big Data).* 141–150. https://doi.org/10.1109/BigData55660.2022.10020756

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[30] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A productive and scalable actor-based programming system for pgas applications. In *International Conference on Computational Science.* Springer, 233–247.

[31] Andronicus Rajasukumar. 2023. *UpDown: An Intelligent Data Movement Architecture for Large Scale Graph Processing.* Master's thesis. University of Chicago, Chicago.

[32] Andronicus Rajasukumar, Jiya Su, Yuqing Wang, Tianshuo Su, Marziyeh Nourian, Jose M. Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. 2024. UpDown: Programmable fine-grained Events for Scalable Performance on Irregular Applications. arXiv:2407.20773 [cs.AR] https://arxiv.org/abs/2407.20773

[33] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. 2024. UpDown: A Novel Architecture for Unlimited Memory Parallelism. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '24).* Association for Computing Machinery, New York, NY, USA, 61–77. https://doi.org/10.1145/3695794.3695801

[34] Dennis M Ritchie, Stephen C Johnson, ME Lesk, BW Kernighan, et al. 1978. The C programming language. *Bell Sys. Tech. J* 57, 6 (1978), 1991–2019.

[35] Kamil Rocki, Dirk Van Essendelft, Ilya Sharapov, Robert Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean Francois Dietiker, Madhava Syamlal, and Michael James. 2020. Fast Stencil-Code Computation on a Wafer-Scale Processor. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–14. https://doi.org/10.1109/SC41405.2020.00062

[36] Kylee Santos, Stan Moore, Tomas Oppelstrup, Amirali Sharifian, Ilya Sharapov, Aidan Thompson, Delyan Z. Kalchev, Danny Perez, Robert Schreiber, Scott Pakin, Edgar A. Leon, James H. Laros, Michael James, and Sivasankaran Rajamanickam. 2024. Breaking the Molecular Dynamics Timescale Barrier Using a Wafer-Scale System. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–13. https://doi.org/10.1109/SC41406.2024.00014

[37] Jiya Su, Alexander Fell, David F. Gleich, and Andrew A. Chien. 2025. Scaling Triangle Counting and K-Truss on the UpDown Architecture. In *IEEE High Performance Extreme Computing Conference*.

[38] Tipparaju, J Nieplocha, D Baxter, C Rassmunsen, and Robert Numrich. 2005. *Symmetric Data Objects and Remote Memory Access Communication for Fortran 95 Applications*. Technical Report. Pacific Northwest National Laboratory.

[39] Yuqing Wang, Charles Colley, Brian Wheatman, Jiya Su, David F. Gleich, and Andrew A. Chien. 2025. How Fast Can Graph Computations Go on Fine-grained Parallel Architectures. arXiv:2507.00949 [cs.DC] https://arxiv.org/abs/2507.00949

[40] Yuqing Wang, Ahsan Parvaiz, Swann Perarnau, and Andrew A. Chien. 2025. Drammalloc: API and Design Documents for a Shared Global Memory Manager. Available from https://people.cs.uchicago.edu/~aachien/lssg/research/10x10/.

[41] Yuqing Wang, Swann Perarnau, and Andrew A. Chien. 2024. UpDown: Combining Scalable Address Translation with Locality Control. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1014–1024. https://doi.org/10.1109/SCW63240.2024.00141

[42] Yuqing Wang, Andronicus Rajasukumar, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Ahsan Pervaiz, Jerry Ding, Charles Colley, Wenyi Wang, Yanjing Li, David F. Gleich, Hank Hoffmann, and Andrew A. Chien. 2023. Efficiently Exploiting Irregular Parallelism Using Keys at Scale. *The 36th International Workshop on Languages and Compilers for Parallel Computing* (October 2023).

[43] Yuqing Wang, Andronicus Rajasukumar, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Ahsan Pervaiz, Jerry Ding, Charles Colley, Wenyi Wang, Yanjing Li, David F. Gleich, Hank Hoffmann, and Andrew A. Chien. 2023. Efficiently Exploiting Irregular Parallelism Using Keys at Scale. In *Workshop on Languages and Compilers for Parallel Computing*.

[44] Brian Wheatman and Andrew A. Chien. 2025. Scalable Graph Algorithms on Distributed UpDown Accelerators. In *IEEE High Performance Extreme Computing Conference*.

[45] Tom White. 2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[46] Ruiqi Xu. 2025. *Accelerating GNN Aggregation Using a Vertex-Centric Approach*. Master's thesis. University of Chicago.

[47] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation* (London, Ontario, Canada) *(PASCO '07)*. Association for Computing Machinery, New York, NY, USA, 24–32. https://doi.org/10.1145/1278177.1278183

[48] Akinori Yonezawa. 1990. *ABCL: An object-oriented concurrent system*. MIT press.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

C_1 Design of the KVMSR+UDWeave programming model that enables applications to separate parallelism, computation binding, and data placement. This enables irregular graph applications with maximum parallelism using UpDown's novel architecture features.

C_2 Demonstration of KVMSR+UDWeave model on challenging graph applications, shows how fine-grained, irregular parallelism can be expressed and managed at high parallelism

C_3 Evaluation that shows excellent strong-scaling to million-fold parallelism on small graphs and high absolute performance that exceeds today's supercomputers

C_4 Empirical evaluation of programmability with several quantitative metrics

A.2 Computational Artifacts

A_1 https://bitbucket.org/achien7242/updown-mpix-sc25/src/main/

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_3, C_4	Figures 9-12 Table V

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

This artifact contains the UpDown simulator and application programs. We use this artifact to evaluate the UpDown's performance and scalability of various graph algorithms, such as BFS, PageRank, and TC, on both real-world and generated graphs. The simulation results demonstrate that UpDown's fine-grained architecture scales well as hardware parallelism increases.

Expected Results

First, as the system scales from 1 UpDown node to 1024 UpDown nodes, the performance of all the algorithms introduced in the paper should scale. This demonstrates the scalability of the UpDown accelerator's fine-grained architectural parallelism for irregular graph computations.

Second, different algorithms such as PageRank, Breadth-First-Search, Triangle Counting, and Ingestion can be implemented and run efficiently on the UpDown accelerator. They should all show good performance scaling despite different scaling trends. This showcases that the UpDown accelerator can support various graph algorithms and efficiently accelerate programs with different structures.

Third, the algorithms do not need to be adopted as more computational resources become available. The resource binding is completed by the KVMSR library, determining where the map and reduce functionality is launched.

Table 6: Estimated reproduction time (in mins) for the PR, BFS, and TC programs. Each entry shows the sum of time to simulate all nine (PR and BFS, 1 to 256 nodes) and 11 (TC, 1 to 1024 nodes) system configurations for a program on a given graph.

Graph	PR	BFS	TC
Forest Fire s28	270	1080	–
RMAT s28	780	1200	31993
RMAT s25	–	–	1002
Erdős-Rényi	1350	1080	–
soc-livej	60	60	20
com-orkut.ungraph	60	120	74
Twitter	420	1080	–
friendster	–	–	626

Table 7: Estimated reproduction time (in mins) to run all experiments for the Ingestion (TFORM) and Partial Match.

Input Data	TFORM	Partial Match
data 2x	320	11160
data	160	–
data 0.1x	40	–
data 0.01	10	–

Expected Reproduction Time (in Minutes)

Due to the large number of experiment configurations in the paper, we will provide an estimation of the expected reproduction time in Tables 6 and 7 grouped by programs and graphs.

Artifact Setup (incl. Inputs)

Hardware. UpDown has been tested on X86 architectures. While there are no specific hardware requirements, a machine with a large amount of memory is required for processing large input graphs.

Software. All experiments were completed on Ubuntu 22.04.4 LTS (Jammy). Additional software packages need to be installed using Ubuntu's package manager:

- **git:** The code repository uses git for version control.
- **gcc:** gcc is used to compile the simulator.
- **Clang:** Clang can also be used. We currently support Clang 7 to Clang 16 (inclusive).
- **Python 3.8+:** The simulator relies on Python development libraries. It can be compiled and run in environments that use Python 3.8 or later.
- **OpenMP:** The simulator uses OpenMP to parallelize the execution.

For UDWeave additionally:

- **treelib:** To build trees for the UDWeave compiler
- **ply:** Interface to the lex and yacc tools

Since those packages are not part of the standard Ubuntu distribution, **python3-pip** from the Ubuntu repository enables the installation of treelib and ply.

To preprocess the input data for the UpDown applications, the following packages are additionally required.

- **numpy:** To be used by the RMAT graph generator for TC
- **scipy:** To be used by the RMAT graph generator for TC

Datasets / Inputs. For PR and BFS, download the data from the Google Drive (link). The raw graph files are in plain text edge list format. For PageRank and BFS, the preprocessing programs convert the graph from edge list to neighbor list format and split high-degree vertices into sub-vertices. The steps for preprocessing are described in the appropriate task sections.

For TC, the input files are from the graph challenge available here: https://graphchallenge.mit.edu/data-sets/. The preprocessing step is also described in the TC task section.

For the ingestion kernels (TFORM and Partial Match), the input files are CSV files in text format (link). No preprocessing is required for those programs.

Installation and Deployment. To start installing the UpDown simulator, execute the following commands to clone the UpDown repository:

```
git clone git@bitbucket.org:achien7242/
    updown-mpix-sc25.git
cd updown
source setup_env.sh
mkdir build; cd build
cmake .. -DUPDOWNRT_ENABLE_TESTS=OFF \
     -DUPDOWNRT_ENABLE_LIBRARIES=ON \
     -DCMAKE_INSTALL_PREFIX=../install \
     -DUPDOWNRT_ENABLE_APPS=ON \
     -DDEBUG_SYMBOLS=OFF \
     -DUPDOWN_ENABLE_FASTSIM=ON \
     -DUPDOWN_ENABLE_BASIM=ON \
     -DUPDOWN_ENABLE_DEBUG=OFF \
     -DUPDOWN_DETAIL_STATS=OFF \
     -DENABLE_SQLITE=OFF
make -j; make install
```

Listing 4: Compilation and Installation Recording of the UpDown simulator

This will build the UpDown simulator in the current `./build/` directory. After installation, the binaries for the artifacts are located in `./install/updown/apps/` and `./install/updown/te/`.

This simulator uses multiple OpenMP threads to accelerate the simulation. For the current terminal session and any UpDown application executed, the number of spawned threads can be controlled by the exporting variable `OMP_NUM_THREADS` or by preceding the simulator executable with the same variable, as shown in Listing 5.

```
export OMP_NUM_THREADS=<omp_threads>

# or
OMP_NUM_THREADS=<omp_threads>
    ./upDownApplication <parameters>
```

Listing 5: Limiting the number of OpenMP threads

The number of threads depends on the host machine configuration. If not set, the simulator maximizes the number of threads but does not exceed the number of cores on the host system.

Artifact Execution

There are two main steps for the performance experiments on the simulator: T_1 prepares the data into the format and T_2 simulates the UpDown programs and baseline CPU programs on the simulator. The following programs in this section are assumed to execute under the root directory of the simulator, the updown folder of the repository. First, execute `source setup_env.sh` to set the environment variables used in the rest of the session.

The details of each step are listed here:

T_1: Data Preparation.

PageRank. The Google Drive link (link) leads to three subdirectories:

1. `preprocessed_data`: This directory is for convenience since it contains the graphs that have already been preprocessed. The process to convert the graphs into this format is described below.
2. `randomGraphs`: This directory hold the random input graphs such Erdős-Rényi, ForestFire and RMAT.
3. `SNAP`: The input graphs from the SNAP repository.

The raw graph files are in plain text edge list format. For PageRank, the preprocessing programs convert the graph from edge list to neighbor list format and split high-degree vertices into sub-vertices. The program's command-line argument specifies the maximum degree to be split.

The preprocessing program is located in the directory `./install/updown/apps` after the simulator is compiled. The command to execute the program is shown in Listing 6:

```
./split_and_shuffle -f <raw_graph_file>
    -m <max_degree> -d -s -l <offset>
```

Listing 6: Preprocessing for PageRank

- **`-f`** is the path to the raw graph text file.
- **`-m`** specifies the criteria of splitting, i.e., the maximum vertex degree of the graph after the splitting. In the paper, we set the max degree after splitting to 512 for PageRank. The actual maximum degree after splitting may be lower due to implementation, but it will never exceed the specified maximum degree.
- **`-d`** indicates that the graph to be split is a directed graph. Without specification, we assume the input is undirected and will create an edge in both directions during the conversion.
- **`-s`** optionally outputs the statistics of the graph before and after splitting to standard output.
- **`-l`** optionally skips the first offset lines of the input (default is 0). The first few lines of the graphs may contain additional information beyond the edges and should be ignored when reading the graphs.

The output split graph files for PageRank's preprocessing program are in binary format and located under the same directory as the input raw graph file with the suffix `_shuffle_max_deg_<max_degree>_gv.bin` and `_shuffle_max_deg_<max_degree>_nl.bin`. The statistics are in a file with the suffix `_m<max_degree>_stats.txt`.

Breadth-First-Search. The input graphs for BFS are located at the same Google Drive location as PageRank (refer to the previous section). The preprocessing program is located in the directory `./install/updown/apps/` after the simulator is compiled. The command to execute the program is shown in Listing 7:

```
./split_shuffle -f <raw_graph_file>
    -m <max_degree> -s -l <offset>
```

Listing 7: Preprocessing for BFS

- **-f** is the path to the raw graph text file.
- **-m** specifies the criteria of splitting (i.e., the max vertex degree of the graph after the splitting). In the paper, we set the max degree after splitting to 4096 for BFS.
- **-s** optionally outputs the statistics of the graph before and after splitting to standard output.
- **-l** optionally skips the first offset lines of the input (default is 0). The first few lines of the graphs may contain additional information beyond the edges and should be ignored when reading the graphs.

The output files for this preprocessing program are in binary format and located under the same directory as the input raw graph file with the suffix `_shuffle_max_deg_<max_degree>.bin`. The statistics are in a file with the suffix `_m<max_degree>_stats.txt`.

Triangle Count. The graphs used in this study include the datasets *friendster*, *com-orkut*, *soc-livej*, and a synthetic RMAT graph of scale 25, all of which are publicly available through the Graph Challenge repository (https://graphchallenge.mit.edu/data-sets/). An additional synthetic graph of scale 28 was generated using a standard RMAT generator configured with parameters $a = 0.57$, $b = 0.19$, $c = 0.19$, and an edge factor of 16.

A Python implementation of the RMAT generator is provided in the directory `./install/updown/apps/`. To generate a graph of scale s, the script can be executed as follows:

```
python3 RMAT_agile.py <s>
```

Listing 8: Execution of the RMAT graph generator for a given scale parameter s

All input graphs are represented as plain text files in which each line specifies an edge via a source and destination vertex identifier. To be used within the UpDown application, these textual graph files must be preprocessed to eliminate duplicate edges and to sort entries by the source vertex ID. This transformation is performed using a preprocessor located in the directory `./install/updown/apps/`. An example invocation of the preprocessing tool is shown below:

```
./tsv rmat-s28.txt rmat-s28
```

Listing 9: Preprocessing of a textual RMAT input graph

The preprocessing step produces two binary output files: `*_gv.bin`, which contains the vertex array, and `*_nl.bin`, which stores the neighbor lists.

Ingestion (TFORM). The input for the Ingestion is a file containing comma-separated values (CSV), and each line represents a record. The record is parsed by TFORM and stored in a graph structure, so that other applications (such as Partial Match) can access the graph's properties. For the graphs downloadable using the Google Drive link, a preprocessing step is not required.

In this paper, we use the notion of `data <m>`, where `m` represents a multiplier. For instance, a multiplier of `0.1x` means that this dataset contains only 10% of the records of the original dataset. This allows us to test the performance for various graph sizes.

Partial Match. This artifact uses a variant of the ingestion, and hence, the inputs are the same files as described in the previous section. Instead of processing the graph in its entirety, it is streamed record-by-record into the ingestion. If an edge or vertex of interest has been discovered during that phase, it is forwarded to the partial match kernel. Its task is to search for patterns of interest and to print an alert on the terminal when such a pattern has been discovered.

T_2: Simulation.

PageRank. The program is located in the directory `./install/updown/apps` after the simulator is compiled, while its sources are available in `./apps/pagerank_udkvmsr/`. Two binaries are created: `pagerankMSRdramalloc` and `pagerankMSRdramallocEFA.bin`. The former is the program that is executed on the TOP core, while the latter contains the instructions for the UpDown architecture. Both of them have to reside in the same directory. To run the program, execute the following command:

```
./pagerankMSRdramalloc <graph> <nodes>
            <accel> <part> <mem>

# Example for 64 UpDown Nodes
./pagerankMSRdramalloc
    rmat_scale_28_seed_48_shuffle_max_deg_512
     64 32 4 64

# Example for 64 UpDown Nodes and 8 Memory
    Nodes (refer to Figure 12 in the paper)
./pagerankMSRdramalloc
    rmat_scale_28_seed_48_shuffle_max_deg_512
     64 32 4 8
```

Listing 10: Executing the UpDown PR application

Replace the arguments enclosed in <> with:

- **graph** path to the binary neighbor list graphs, the output of the preprocessing program.
- **nodes** number of nodes of the UpDown system. In the paper, we simulated machine configurations ranging from 1 node to 256 nodes.
- **accel** The number of UpDown accelerators per node. Always set to 32 for PageRank.

part The number of partitions that each lane should handle (Block binding scheme). For this paper, we set this parameter to 4.

mem The number of memory nodes that are available for DRAM-malloc. This allows us to experiment with a sweeping number of memory nodes. The number needs to be a power of 2.

Breadth-First-Search. The sources for the BFS program are located in the directory `./apps/sync_bfs/`. After compilation, the binaries will be available in `./install/updown/apps`: `bfs_udweave` (TOP program) and `bfs_udweave_exe.bin` (UpDown program). To run it, execute the following command:

```
./bfs_udweave <graph> <lanes> <accel>
    <root_VID> <mem>

# Example for 64 UpDown Nodes
./bfs_udweave
    rmat_scale_28_seed_5_shuffle_max_deg_4096
    .bin 131072 32 58879427 64

# Example for 64 UpDown Nodes and 8 Memory
    Nodes (refer to Figure 12 in the paper)
./bfs_udweave
    rmat_scale_28_seed_5_shuffle_max_deg_4096
    .bin 131072 32 58879427 8
```

Listing 11: Executing the UpDown BFS application

Replace the arguments enclosed in <> with:

graph The path to the neighbor list graphs, the output of the preprocessing step.

lanes The number of UpDown lanes. One UpDown node has 2048 lanes. For example, 4096 corresponds to the 2-node data point in the paper's Figure 9. In the paper, we simulated the UpDown system, ranging from 1 node to 256 nodes, which corresponds to 2,048 to 524,288 lanes.

accel The number of UpDown lanes controlled by a single master. The UpDown system, which ranges from 1 node to 256 nodes, requires multi-level control for synchronization and broadcast to reduce synchronization and broadcast overhead. At each level, one control lane manages `accel` lanes. For all BFS simulations in the paper, we set `accel` to 32.

root_VID The root vertex ID of the BFS tree. For the ER graphs, the selected root vertex ID is set to 0, while for the RMAT graphs, it is 28.

mem The number of memory nodes which are available to DRAM-malloc for the swiping experiment shown in Figure 12 of the paper.

Triangle Count. The compiled binary for the triangle counting (TC) application is located in the directory `./install/updown/apps/`. The program `three_clique_count_mm_global` serves as the host-side TOP application, executed on the main processor, while the file `three_clique_mm_exe.bin` is the corresponding executable for the UpDown architecture. The source code for both components resides in `./apps/three_clique_mm_dramalloc` and `./apps/three_clique_mm_dramalloc/udwsrc`, respectively.

Listing 12 illustrates the execution of the TC program. Its input parameters are as follows:

gv The binary file containing the vertices as generated in the preprocessing step.

nl The binary file containing the list of neighbors as generated in the preprocessing step.

u The number of UpDown lanes that are tasks to compute TC. An UpDown node has 2048 lanes in total.

t A stride of master lanes managing thread creation of the worker lanes. For the experiments, we use 32 masters independent of the number of lanes.

m A debugging flag that indicates whether TC should compute the triangles over the whole input graph or only a partial one. For the experiments, set $m = 0$.

```
./three_clique_count_mm_global <gv> <nl> <u>
    <t> <m>

# Example for 64 UpDown Nodes = 131072 lanes
time ./three_clique_count_mm_global graph500
    -scale25-ef16_adj_gv.bin graph500-
    scale25-ef16_adj_nl.bin 131072 32 0
```

Listing 12: Executing the UpDown TC application

Ingestion (TFORM). The compiled binary for the ingestion application is located in the directory `./install/updown/te/wf2/wf2k1_constructGraph/`. The program `wf2k1a_variant2_3` serves as the host-side TOP application, executed on the main processor, while the file `wf2k1a_fsr_exe.bin` is the corresponding executable for the UpDown architecture. The source code for both components resides in `./te/wf2/wf2k1_constructGraph/Variant2_3`.

Listing 13 illustrates the execution of the TC program. Its input parameters are as follows:

inFile The input CSV file

cfgFile This is the configuration file containing configurations for PGA, worker lanes, etc. For Figure 10 of the paper, we provide configuration files for all datasets in the `./te/wf2/wf2k1_constructGraph Variant2_3/config urations` directory. An example of such a configuration file with additional comments is shown in Listing 14.

```
./wf2k1a_variant2_3 <inFile> <cfgFile>

# Example for 64 UpDown Nodes
./wf2k1a_variant2_3 data.10.csv Variant2_3/
    configurations/data.10.csv_64nodes.cfg
```

Listing 13: Executing the UpDown Ingestion Application

```
NUM_TFORM_LANES 65536 # lanes for parsing (
    TFORM)
NUM_PGA_LANES    65536 # lanes for graph
    structure
# entries per bucket
```

```
VERTEX_EB        16     # vertices
EDGE_EB          64     # edges
# buckets per lane
VERTEX_BL        256    # vertices
EDGE_BL          256    # edges
```

Listing 14: An example for a configuration file for the ingestion application

Partial Match. The source code for this artifact is located in `./te/wf2/wf2k4_partialMatch`. To configure the application, the user must modify a UDWeave header file named `partialMatch.udwh` and recompile the code using the UDWeave compiler. Listing 15 highlights the relevant section of the header file, where different configurations for vertex allocation lanes and bucket distribution per lane can be selected by commenting or uncommenting the corresponding preprocessor directives.

```
// PGA
#define PGA_VERTEX_NUM_ALLOC_LANES   256
#define PGA_VERTEX_BUCKETS_PER_LANE  128

// #define PGA_VERTEX_NUM_ALLOC_LANES   1024
// #define PGA_VERTEX_BUCKETS_PER_LANE  256

// #define PGA_VERTEX_NUM_ALLOC_LANES   2048
// #define PGA_VERTEX_BUCKETS_PER_LANE  256

// #define PGA_VERTEX_NUM_ALLOC_LANES   8192
// #define PGA_VERTEX_BUCKETS_PER_LANE  256

// #define PGA_VERTEX_NUM_ALLOC_LANES   32768
// #define PGA_VERTEX_BUCKETS_PER_LANE  16

// #define PGA_VERTEX_NUM_ALLOC_LANES   65536
// #define PGA_VERTEX_BUCKETS_PER_LANE  8
```

Listing 15: Configuring the Partial Match Application

Once the configuration parameters have been adjusted, the application must be recompiled. This is done by running `make`, which invokes the UDWeave compiler, linker, and assembler to produce a new UpDown binary. After compilation, the user should navigate to the build directory `./build/te/wf2/wf2k4_partialMatch/` and execute `make -j; make install`. The resulting binaries, reflecting the updated configuration, will be available in `./install/updown/te/wf2/wf2k4_partialMatch/`.

The artifact can then be executed as demonstrated in Listing 16. It accepts as input the same CSV file as the Ingestion (TFORM), specified via the `<inFile>` parameter.

```
./wf2k4 <inFile>

# Example
./wf2k4 data.01.csv
```

Listing 16: Executing the UpDown Partial Match Application

Artifact Analysis (incl. Outputs)

While a program is executed, the textual output will be printed on the terminal. An example might look like this:

```
[BASIM_PRINT] 527500: [NWID 0][TID 12]...
[BASIM_PRINT] 553400: [NWID 0][TID 12]...
[BASIM_PRINT] 560500: [NWID 0][TID 12]...
[BASIM_PRINT] 570000: [NWID 0][TID 12]...
[BASIM_PRINT] 570000: [NWID 0][TID 12]...
```

Listing 17: Example Output

The first number represents a time stamp in the form of simulated ticks of the simulator, enabling us to calculate the time at which the output was produced. The target operating frequency of UpDown is 2 GHz. Therefore, to convert the ticks into time in seconds, compute

$$time[s] = \frac{ticks}{2 \times 10^9}$$

There is no automated script available to extract the timings from the log files. Most results are obtained by computing the difference between two events in the generated log file.

PageRank. Listing 18 shows a snippet of the output after the PR application completes. Here, `updown_init` refers to the event label that starts the algorithm, while the event label `updown_terminate` is called when the PR application finishes.

For this example, the simulated time was

$$\frac{10582600 - 15000}{2 \times 10^9} \text{s} = 0.0053\text{s}$$

To recreate Figure 9, choose the desired input graph and vary the number of nodes from 1 to 256.

Table 8: Raw Speedup Data for the PR Application

Nodes	Erdős-Rényi	Forest Fire	Twitter	RMAT s28
1	1.00	1.00	1.00	1.00
2	2.03	1.99	2.18	2.21
4	2.17	2.20	2.03	3.39
8	2.56	2.76	2.40	4.03
16	3.19	5.25	8.63	5.36
32	14.19	14.38	20.74	19.29
64	45.01	30.48	42.02	50.83
128	101.60	54.13	75.42	97.46
256	191.74	91.84	131.37	178.21

Breadth-First-Search. Listing 19 shows a snippet of the output after the BFS application completes. In the log, each `[main_master__init] BFS Start` indicates the start of an iteration. The program completes when no new frontier is added to the queue. Therefore, the program execution is timed from the first occurrence of `[main_master__init] BFS Start` until `[main_master__reduce_launcher_done] BFS finish`.

For this example, the simulated time was

$$\frac{6780000 - 9900}{2 \times 10^9} \text{s} = 0.0034\text{s}$$

Table 9 shows the raw data for Figure 9 in the paper.

```
[BASIM_PRINT] 15000: [NWID 0][TID 15][updown_init] [DEBUG][NWID 0] Event <updown_init>
[BASIM_PRINT] 10582600: [NWID 0][TID 16][pr__cache_flush_return]
[BASIM_PRINT] 10582600: [NWID 0][TID 16][pr__cache_flush_return] [DEBUG][NWID 0][pr::
    cache_flush_return] Finish flushing the cache. UDKVMSR program pr terminates, number of
    reduce task process = 2039314526. Return to user continuation 256904628
[BASIM_PRINT] 10582700: [NWID 0][TID 15][updown_terminate] [DEBUG][NWID 0] Finish PageRank,
    number of edge updates = 2039314526
```

Listing 18: A snippet of the log output of PR

```
[BASIM_PRINT] 9900: [NWID 0][TID 9][main_master__init] BFS Start
[BASIM_PRINT] 47000: [NWID 0][TID 9][main_master__map_launcher_done] [DEBUG][NWID 0] <
    map_launcher_done> update_v:1
[BASIM_PRINT] 54000: [NWID 0][TID 9][main_master__map_split_launcher_done] [DEBUG][NWID 0] <
    map_split_launcher_done> update_split_v:1
[BASIM_PRINT] 61200: [NWID 0][TID 9][main_master__reduce_launcher_done] [Itera 0]: add queue
    7 traversed edges 7
...
[BASIM_PRINT] 6744400: [NWID 0][TID 9][main_master__init] BFS Start
[BASIM_PRINT] 6765800: [NWID 0][TID 9][main_master__map_launcher_done] [DEBUG][NWID 0] <
    map_launcher_done> update_v:5283
[BASIM_PRINT] 6772800: [NWID 0][TID 9][main_master__map_split_launcher_done] [DEBUG][NWID 0]
    <map_split_launcher_done> update_split_v:5283
[BASIM_PRINT] 6780000: [NWID 0][TID 9][main_master__reduce_launcher_done] [Itera 6]: add
    queue 0 traversed edges 64
[BASIM_PRINT] 6780000: [NWID 0][TID 9][main_master__reduce_launcher_done] BFS finish
```

Listing 19: A snippet of the log output of BFS

Table 9: Raw Speedup Data for the BFS Application

UpDown Nodes	com-orkut	soc-livej	RMAT s28
1	1.0	1.0	1.0
2	2.6	2.0	2.3
4	4.5	2.9	3.9
8	7.0	4.1	7.4
16	8.9	4.9	17.5
32	12.3	5.9	31.3
64	13.7	5.5	59.7
128	15.5	5.7	112.8
256	16.6	5.7	178.7

Triangle Count. Listing 20 shows a snippet of the output after the TC application completes. Here, `main_master__init_tc` refers to the event label that starts the triangle counting, while the event label `main_master__tc_launcher_done` is called when the TC application finishes returning the result. For this example, the simulated time was

$$\frac{1822690400 - 9900}{2 \times 10^9}\mathrm{s} = 0.91\mathrm{s}$$

Table 10 shows the raw data for Figure 9 in the paper.

Table 10: Raw Speedup Data for the TC Application

Nodes	friendster	orkut	soc-livej	RMAT25	RMAT28
1	1.00	1.00	1.00	1.00	1.00
4	3.98	4.13	3.99		
8	7.92	7.95	7.52	7.87	7.99
16	15.71	15.62	13.66	15.53	
32	31.17	29.68	22.76	30.41	
64	61.55	54.60	37.11	59.47	
128	119.96	95.32	48.89	113.90	
256	232.66	157.12	56.88	210.70	243.86
512	436.70	213.91	55.95	391.73	470.83
1024	790.82	211.93	48.24	665.18	899.24

Ingestion (TFORM). Unlike the other artifacts, this program does not display messages directly on the terminal. Instead, it generates a log file named `perflog.tsv`, which is stored in a newly created directory `wf2k1a_fsr_exe.logs/` located in the same directory from which the program is executed. To monitor the program's execution, users should open this file in a text editor and search for the entries `UDKVMSR started` and `UDKVMSR finished for phase2`, which mark the beginning and end of the UpDown execution, respectively. An example of such a log is provided in Listing 21.

```
[BASIM_PRINT] 9900: [NWID 0][TID 9][main_master__init_tc] [DEBUG] Main TC Master Start
[BASIM_PRINT] 19300: [NWID 0][TID 9][main_master__init_done] [DEBUG][NWID 0] <init_done>
    finsh
[BASIM_PRINT] 1820144000: [NWID 0][TID 9][main_master__v_launcher_done] [DEBUG][NWID 0] <
    map_launcher_done>
[BASIM_PRINT] 1822679300: [NWID 0][TID 9][main_master__int_launcher_done] [DEBUG][NWID 0] <
    int_launcher_done>
[BASIM_PRINT] 1822690400: [NWID 0][TID 9][main_master__tc_launcher_done] [DEBUG] <tc_return>
    result:616168188120
```

Listing 20: A snippet of the log output of TC

```
HOST_SEC   FINAL_TICK   SIM_TICKS   SIM_SEC   CPU_ID   NETWORK_ID   THREAD_ID   EVENT_LABEL
    LANE_EXEC_TICKS   MSG_ID   MSG_STR
64.59    15500    15500    0.000000  0    15  40312  1        1  UDKVMSR started
110.54   1503400  1503400  0.000000  0    15  40404  1226826  1  UDKVMSR finished
184.72   2076100  2076100  0.000000  0    66  40988  1        1  UDKVMSR started for phase2
279.54   2787400  2787400  0.000000  0    66  41080  538469   1  UDKVMSR finished for phase 2
```

Listing 21: A snippet of the log output of the Ingestion Application

The log reveals that the ingestion process proceeds in two distinct phases. First, the CSV input is parsed, and each record is converted into a 64-byte binary format. Second, the resulting binary records are inserted into the underlying graph data structure.

For this example, the number of ticks in the second column is used to extract the simulated time:

$$\frac{2787400 - 15500}{2 \times 10^9}\text{s} = 0.0014\text{s}$$

Table 11 shows the raw data for Figure 10 in the paper.

Table 11: Raw Speedup Data for the Ingestion Application

UpDown Nodes	data 0.01x	data 0.1x	data	data 2x
1	1.00	1.00	1.00	1.00
2	7.52	16.27	4.65	1.57
4	7.47	31.00	23.99	7.43
8	7.49	57.20	68.51	43.07
16		70.23	125.69	133.13
32		72.52	219.94	243.78
64			344.23	431.71
128			619.65	679.32
256			657.39	1178.20

Partial Match. Like the Ingestion (TFORM) program, this program does not print the data needed to reproduce Figure 11 directly to the terminal. Instead, it writes performance logs to a file named `perflog.tsv`, which is placed in a newly created directory `./wf2k4EFA.bin.logs/` in the same location where the program is executed. To monitor execution, users should open this log file and search for entries of the form `F`n `called`, where $2 \le n \le 9$. These indicate invocations of the KVMSR library to filter for partial patterns.

Table 12: Raw Speedup Data for the Partial Match Application

UpDown Nodes	Speedup
1/8	1.00
1/2	3.34
1	5.56
4	10.42

To extract timing data, locate a sequence in the log where a `Record detected` entry precedes and follows an `F2 called` entry. A sample log segment is shown in Listing 22. The time between the two `Record detected` entries includes the time spent in KVMSR.

The simulated execution time for ingestion, sub-pattern matching, and filtering (KVMSR) is calculated as the difference between the timestamps (in ticks) in the second column of the two `Record detected` entries. For example:

$$\frac{22212451500 - 22182912800}{2 \times 10^9}\text{s} = 0.015\text{s}$$

To produce Figure 11, compute the average of all such timings where `F2 called` occurs between two `Record detected` entries. The raw data to recreate Figure 11 of the paper is tabulated in 12.

```
18380.46   22182912800   22182912800     0.000000   1024    165     33316   1994458800      8
      Record detected
18380.46   22182913300   22182913300     0.000000   1024    141     90680   1994459276      2
      EdgUpdCall 8802991356(8397038458079794983, 0b11111)
18380.46   22182913300   22182913300     0.000000   1024    141     90680   1994459287      2
      EdgUpdCall 8802991356(12081715347070129699, 0b11111)
18380.46   22182913300   22182913300     0.000000   1024    141     90680   1994459302      9
      partialMatch
18380.46   22182913400   22182913400     0.000000   1025    248     87828   70626924        10
    startPartialMatch: srcID: 1472154222902711100, dstID, 1128501731262832684, type_oid: 3,
    edgeAddress: 8802991356

...

18380.83   22183294500   22183294500     0.000000   1025    204     85792   70652340
    1020  F2 called
18380.92   22183312500   22183312500     0.000000   4739    160     85900   20176   1021
    F2doAll: eid -1661631708168496906, type_oid: 8, author: 210057285575660412, item:
    942274057961536388, type: 6

...

18461.32   22212451500   22212451500     0.000000   1024    165     33316   1994459481      8
    Record detected
```

Listing 22: A snippet of the log output of the Partial Match Application