THE UNIVERSITY OF CHICAGO

ORGANIZING FINE-GRAINED PARALLELISM USING KEYS AT SCALE

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
YUQING WANG

CHICAGO, ILLINOIS
GRADUATION DATE

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my advisor, Dr. Andrew A. Chein, whose expertise and insight added considerably to my graduate experience. I am extremely grateful for his guidance throughout the research and writing of this thesis.

ABSTRACT

Rapidly proliferating machine learning and graph processing applications, demand high-performance on petascale datasets. Achieving this performance requires efficient exploitation of irregular parallelism, as their sophisticated structures and real-world data produce computations with extreme irregularity. The need to exploit large-scale parallel hardware (million-fold parallelism) is a further challenge.

Programming irregular data and parallelism using existing models (e.g., MPI) is difficult because they couple naming, data mapping, and computation mapping. Further, they only exploit coarse-grained parallelism. To solve this problem, we present a key-based programming model, called key-value map-shuffle-reduce (KVMSR). The model enables programmers to express fine-grained parallelism across programmer-defined key-value sets. The parallel computation can then be optimized using KVMSR's modular control for load balance and data locality. KVMSR achieves this by expressing parallelism with respect to a global address space and providing modular control to flexibly bind computation to compute resources.

We define the KVMSR model and illustrate it with three programs, convolution filter, PageRank and BFS, to show its ability to separate computation expression from binding to computation location for high performance. We evaluate KVMSR on a novel fine-grained parallel architecture, called UpDown, supporting up to 4 billion fold hardware parallelism in the full system design. On an 8,192-way parallel compute system, KVMSR modular computation location control achieves up to 9,202x performance with static approaches and an increase of 3,136x to 4,258x speedup with dynamic approaches for computation location binding comparing to the single-thread CPU programs.

CHAPTER 1

INTRODUCTION

1.1 Introduction

In the past last few decades, there has been a rise of "internet-scale" computation on big data, producing a growing need for computing systems that process large-scale data efficiently. Among the emerging applications, graph computation is ubiquitous as graphs are used to represent complex relationships efficiently in various scenarios including social networks, world wide web, recommendation systems, bio-informatics etc. [16, 19, 29, 15]. Graph computations can exhibit high data-parallelism – exploited in varied software frameworks [23, 36, 8, 34, 37]. However parallelism alone does not translate to scalable. As the data keeps growing, graphs are now occupying gigabytes of memory and if this trend continues, real-time analysis of petabytes large graphs will be the norm in the next decades. Given the scale of the application, graph computation typically requires running on large-scale parallel systems with millions of compute cores and petabytes of memory for storing the data.

To achieve decent performance on large-scale parallel machines, application programmers need to generate sufficient parallelism to fill up the machine, effectively bind the parallelism to compute elements and manage parallelism accordingly to balance the load and better utilize the parallel compute resources. This is challenging because real-world graphs give rise to irregular parallelism and poor data locality, producing load imbalance, costly communications and data movement, and poor overall performance.

Main-stream parallel programming solutions do not support irregular applications well (e.g., real-world graph analytics and graph pattern minings). Among them, two of the most commonly used are message-passing interface (MPI) and partitioned global address space models (PGAS). MPI follows single-program-multiple-data model (SPMD) and its programs require partitioning the data across the machine so that the corresponding computation are parallelized. With scant support for global data structures, in order to achieve high performance on irregular parallelism, programmers must assemble the data required for each piece of parallel computation and align it to a static set of workers (ranks) [4]. On the other hands, PGAS model provides a global address space that aids in building the distributed global data structure. Despite its capability of expressing global data structure, PGAS still fails to serve these graph analytic applications well due to the graphs' irregular data and the resulting computation load-imbalance [30]. For both solutions, dealing with graphs is possible, but difficult because of irregular data and parallelism [1, 27].

A promising direction is MapReduce. Many applications have been expressed under this framework in the past several decades mostly due to the prosper of functional languages and cloud MapReduce. Its simple yet powerful interface allows parallel computation expressed in terms of independent map and reduce functions and parallelized on parallel compute resources. Despite the similarity, traditional functional programming languages and cloud MapReduce optimized for different paths and none could serve our purposes. Functional programming languages focus on expressing fine-grained parallelism in a shared memory machine. Thus, they cannot support these future large-scale parallel machines with petabytes of distributed memory. Cloud map-reduce systems add keys to organize the computation and are more scalable, but focus more on fault tolarance and fails to exploit fine-grained parallelism efficiently with its coarse-grained approach [24, 31, 9].

1.1.1 The Problem

1.1.2 Approach

We propose the key-value map-shuffle-reduce (KVMSR) framework to support irregular data and computation parallelism in future large-scale parallel systems. These systems will use novel building blocks for fine-grained parallelism and scale to 30 million parallel computation elements [32]. To optimize irregular programs, KVMSR combines rich-structured keys and global addressing of data and adds novel simple interfaces for programmers to directly control the binding of map and reduce tasks to compute resources. Hence, programmers can exploit the expressiveness of keys to flexibly manage parallelism and then optimize performance, in a modular fashion by controlling data locality and load balance.

1.1.3 Results and Key Contributions

We describe our KVMSR model, use program examples to illustrate it, and then show how the model's flexibility supports high performance for challenging irregular computations.

Specific contributions of this paper include:

- Design of KVMSR for expressing and managing fine-grained parallelism using keys. KVMSR expresses the binding of computation to compute resources independently from program computation and data structure.
- Examples of how the keys in KVMSR can be used to efficiently control computation binding, both statically and dynamically (using application, system, and data information), to achieve load balance and high performance on irregular graph applications
- Evaluation that shows on a 8,192-way parallel compute system, KVMSR modular computation location control achieves up to 2,317x performance with static approaches and an increase of 549x to 2,715x speedup with dynamic approaches for computation location binding.

The remainder of the thesis is organized as follows. Chapter **??** gives a brief overview of the background. Then, the KVMSR model is defined in Chapter 3, followed by two program examples described in Chapter 4. An implementation of the KVMSR model is evaluated in Chapter 5, to show its flexibility and performance benefits. Finally, Chapter 6 discusses related work in the field, and Chapter 7.1 summarizes the thesis and points out directions for future work.

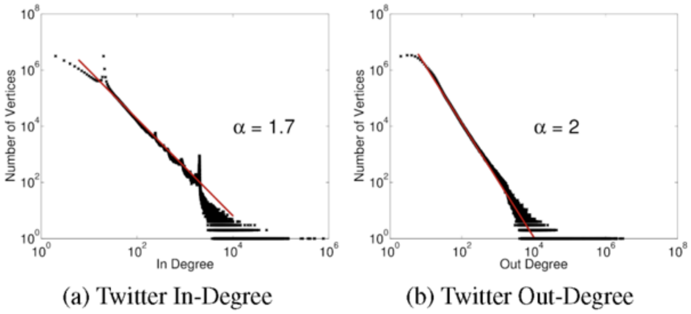
CHAPTER 2

BACKGROUND

This chapter presents a overview of the data-dependent irregular applications and its challenge on conventional parallel architectures and then focuses on one of the well-known programming paradigms for parallel application, MapReduce. It then describes the challenge introduces a fine-grained architecture targeting irregular parallelism and shows the potentials it unlocks for future parallel programming models.

2.1 Irregular Applications and Limited Performance Scaling

Irregular applications are hard to achieve performance on conventional coarse-grained parallel architectures due to their data-dependent imbalanced work distribution. One example of such an application is graph computation. Real-world graphs typically have skewed degree distribution [6, 2], that is a majority of vertices are of low degrees whereas a small portion of vertices have magnitudes more of neighbors. Figure 2.1 depicts the power-law degree distribution of Twitter follower network.



(a) Twitter In-Degree

(b) Twitter Out-Degree

Figure 2.1: The in and out degree distribution of the Twitter follower network

This graph property poses a challenge on efficiently parallelizing graph computation because graph algorithms are usually defined in a vertex/edge centric view where computation is expressed in terms of vertex/edge program. For example, a Pregel program requires defining a `Vertex` subclass and implementing the computation for a vertex in the `Compute()` method [23]. Listing 2.1 shows the PageRank implemented in Pregel. The `Compute()` method is executed on each vertex and the amount of work per invocation is determined by the number of neighbors for that vertex. The skewed degree distribution of real-world graphs results in a fraction of the vertices having magnitudes more work than the rest.

Listing 2.1: PageRank implemented in Pregel [23].

```
class PageRankVertex:
    public Vertex<double, void, double> {
    public:
        virtual void Compute(MessageIterator* msgs) {
            if (superstep() >= 1) {
                double sum = 0;
                for (; !msgs->Done(); msgs->Next())
                    sum += msgs->Value();
                *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
            }
            if (superstep() < 30) {
                const int64 n = GetOutEdgeIterator().size();
                SendMessageToAllNeighbors(GetValue() / n);
            } else {
                VoteToHalt();
            }
        }
};
```

The issue is more profound on conventional parallel machines, primarily because they rely on the single-program-multiple-data (SPMD) and message-based programming model to mitigate the high cost of communication and remote data access. That is, the graph is statically partitioned across the system and each node is assigned with a subset of the vertices and also responsible for executing the corresponding vertex programs. Remote data accesses are done with messages, which are also batched for performance. Graph computation's fined-grained parallelism fits poorly into this model. Mostly because of the skewed degree distribution of real-world graphs, part of the system will be assigned significantly more work than the rest, leading to imbalance load across the system and under utilizing the hardware parallelism.

2.2 Functional Language and Cloud Map-reduce

One of the parallel programming diagrams used widely is called MapReduce. It is known for its simple yet powerful interface for expressing parallel computation.

The idea of MapReduce originated from the functional programming domain: functional programming languages describes computations in terms of functions applied to sets/arrays (map) and combine the results (reduce) [25, 24, 7]. These constructs can be used to express parallelism and exploit it on multicore and larger NUMA shared-memory machines. However, these machines have limited scalability with the largest systems around 256 cores, significantly smaller than the machine scale we are targeting at.

Later, in early 2000, cloud companies built a different map-reduce, designed for scale-out, to internet-scale computations [9, 10]. The key motivation was to exploit the natural and flexible expression of parallelism. These systems solved the important problems of reliability (map and reducers), but with the significant restriction of no shared data structures (across map or reduce functions). The cloud systems added keys, using them to both express computation function, and indirectly to control parallelism. However, these systems manage load balance automatically, depending on hashing and balanced sorts, eschewing programmer involvement. This works adequately because cloud MapReduce systems typically operate on coarse-grained tasks, running billions of instructions, many orders of magnitude larger than the 100 instruction fine-grained tasks we are pursuing.

MapReduce's simple and expressive interface can serve the first requirement posted in the above section and is a good candidate for expressing fine-grained software parallelism for graph computation. Despite the richness of domain, none of the existing frameworks provide a way for programmers to control the location of compute or data, i.e., the second requirements.

2.3 Fine-grained Parallel Hardware Architectures

Many novel architectures are proposed targeting challenge of efficient data-dependent irregular parallelism. The key characteristics of the architectures are: 1) abundant hardware parallelism consisting of millions of simple cores in contrast to the conventional complex out-of-order CPU cores with hierarchies of power-intensive caches, 2) nanosecond-level communication latency enabled by advanced interconnect techniques plus hardware support for fast messaging on low-diameter network in contrast to the software-based message solution on traditional long-latency interconnect, and 3) asynchronous DRAM accesses capable of generating high memory-level parallelism (MLP) and better utilizing the memory bandwidth in contrast to the synchronous load-store architecture with limited MLP.

The UpDown architecture is one example of novel architectures targeting irregular applications. Figure 2.2 presents a high-level view of the system design. An UpDown system consists of 16k nodes, each of which has 1 multi-core out-of-the-shelf CPU, 8 stacks of HBM memory, and 32 customized ASIC accelerators, called UpDown accelerator, as shown in figure 2.3. An UpDown accelerator consists of 64 MIMD cores, called lanes, and each lane has 64KB single-cycle-access software-managed scratchpad memory. A lane can sustain up to 128 concurrently active threads, each of which has independent hardware thread context (instruction pointer, general-purpose registers, etc.) Each lane has a FIFO event queue for incoming messages and an operand buffer for the message operands. The execution of a lane is scheduled based on the events in the queue.

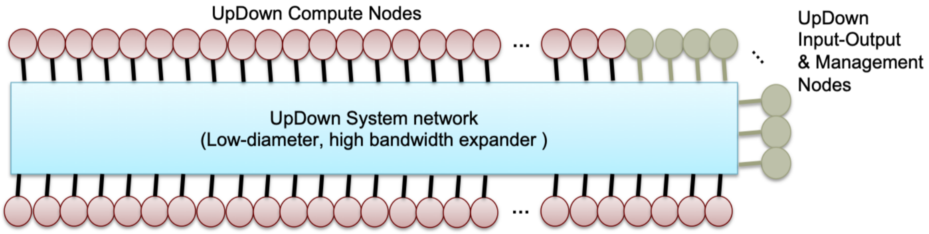

UpDown Compute Nodes
UpDown Input-Output & Management Nodes
UpDown System network
(Low-diameter, high bandwidth expander)
...

Figure 2.2: The UpDown System connected by the high performance system network.

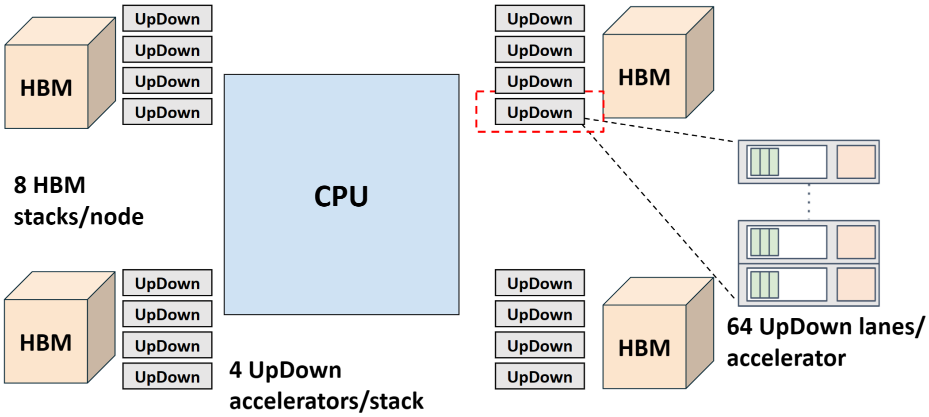


Figure 2.3: each node has 1 CPU, 32 UpDown accelerators with uniform access to 8 HBM stacks.

The UpDown accelerator is programmed using a RISC-V-like ISA with customized instructions for hardware-assisted event-driven programming. These instructions interact with the hardware structures customized for fast messaging described above and speedup event creation and event handling on UpDown. Hence, UpDown lane can create a event and send in as minimal as 3 instructions. On the receiver side, the message operands are directly mapped onto the register name space so that instructions can directly operate on the incoming message's operands, unlike on conventional machines, the data must be moved to a register before compute on it.

All the UpDown accelerators are connected via a diameter 3 topology called Polarstar network featuring low-diameter, low-latency, and high-bandwidth communication [18]. UpDown provides a name space called NetworkID (or nwid in short) for all the accelerators in a 16k node machine to send asynchronous messages directly to each other. Messages on UpDown are extremely fine-grained, typically between 4 to 12 words, including the destination UpDown lane nwid, the thread id to be activated, the event to be triggered at the destination lane, and the data operands. For a full-scale system with 16k nodes, 32 UpDown accelerators per node, 64 lanes per accelerator and up to 128 thread context per lane, it's capable of delivering 4,194,304,000 fold hardware parallelism.

Despite the promising performance potentials provided by the hardware design, the extremely fine-grained hardware parallelism poses a significant challenge on writing software programs capable of utilizing its potentials and delivering good performance. Most importantly, one need to solve two major problems: 1) *how to flexibly express and generate the fine-grained parallelism in software* and 2) *how to map the software parallelism onto the hardware with good performance and high utilization*?

CHAPTER 3

KVMSR PROGRAMMING MODEL

This chapter gives an overview of the KVMSR programming model. It first illustrates the objectives of the design and then provides an abstract view of the target parallel system. The rest of the chapter illustrates the KVMSR interface design, including defining data layout, expressing computation, and binding computation to processing units.

3.1 Objectives

Designing a programming model for irregular applications on fine-grained parallel machines requires understanding what a good parallel program should look like. Here, we identify three main aspects critical to parallel programs. As shown in Figure 3.1, successful parallelization of a program requires controlling three dimensions (namely, data layout, computation mapping, and parallelism binding) of the program, coordinating them to achieve good load balance and high machine utilization. The latter will lead to good parallel scalability and performance.

To achieve the goal, parallel applications must deal with several challenges including accurately expressing parallel computation and data structures, efficiently mapping computation to compute locations (e.g., cores or lanes), and data to memory locations (e.g., memory stacks or banks). as illustrated in Figure 3.1. While doing these correctly from a functional point of view is already challenging for regular HPC applications, doing so and also achieving scalable performance is even harder for irregular data-dependent applications. Therefore, the objective of a programming model is not limited to expressing the parallel computation correctly but also involves ease of tweaking the dimensions to balance the irregular data and computation across the system for performance. In other words, we are aiming to solve two problems: 1) *how to flexibly express the fine-grained parallel computation in software* and

2) *how to map the software parallelism onto the hardware with good performance and high utilization?*

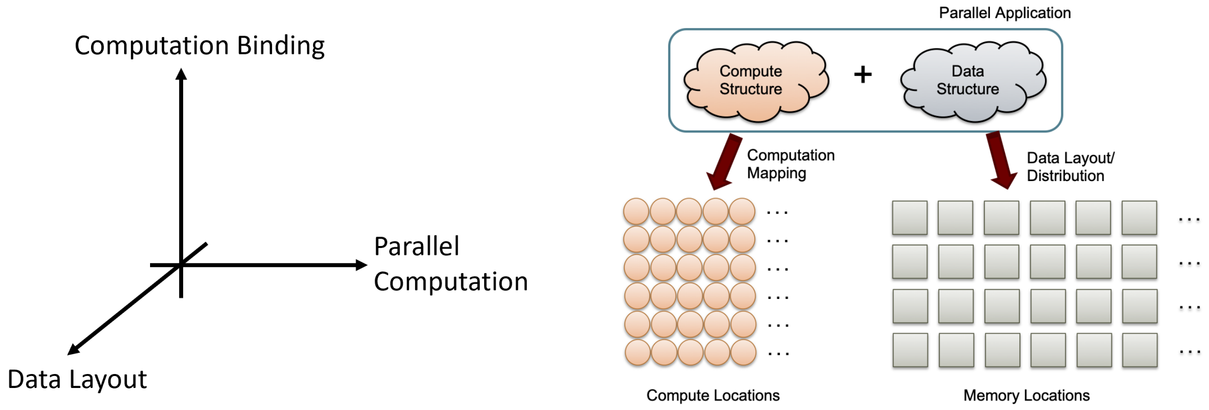


Figure 3.1: Parallel applications manage three dimensions to achieve performance (left). Computation and data need to be mapped to distributed capabilities in a balanced way for good performance.

3.2 KVMSR Programming Model

To answer the two questions, we propose the key-value map-shuffle-reduce (KVMSR) model featuring a flexible expression of parallel computation with a customized choice of computation binding independent from data distribution.

KVMSR employs *keys* as the critical abstraction for programmers to express parallel computation. Parallelism is equal to the number of keys in the input key-value set and computation is as fine-grained as the map task corresponding to a key. The reduce tasks are similar except that the parallelism equals the total number of intermediate pairs generated by the map tasks. That is, a reduce task is generated for each intermediate key-value pair emitted by the map task.

KVMSR builds upon a global address space abstraction for data structures and supports customized data structures so long as they conform with the key-value set interface. Therefore, a data element can be addressed/named uniformly from anywhere in the machine. From the program's point of view, there is no distinction between accessing local data vs remote data, eliminating the programmer from manually managing the data movement.

KVMSR's major innovation is to use the keys as the basis for programmers to bind parallel computation to compute resources. Most importantly, such binding is expressed independently from the data layout and program computation and can be done statically or dynamically based on the runtime load status of the machine, resulting in a clean and modular interface. Such flexibility enables programmers to statically or dynamically load-balance their parallel programs without reorganizing the data or changing the map and/or reduce task implementation to accommodate the changes in binding.

Tersely, the key elements of KVMSR include:

1. Flexible and fine-grained parallelism, expressed as `kv_map()` and `kv_reduce()` tasks on keys

2. User-defined *key* spaces to control binding of `kv_map()` and `kv_reduce()` tasks to computation resources

3. Global address space for uniformly addressing and expressing of global data layout

4. Exposing machine primitives for exploring data location and dynamic computation load

The four features collectively enable high-level programming of applications with global data structures and independent control of a program's parallelism and computation binding. With a simple MapReduce-like interface, one can tune different dimensions of a parallel program to achieve high performance on the target system: exploring computation binding to exploit parallelism, colocating for data locality, and load balancing based on dynamic information.

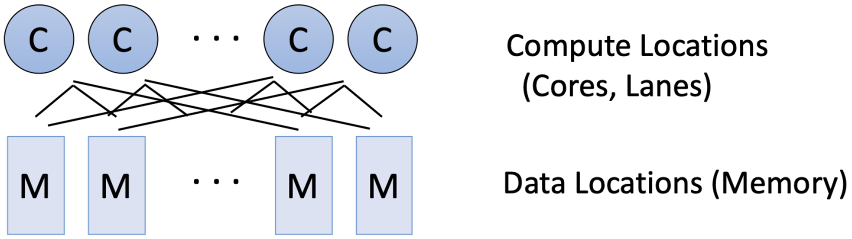


Figure 3.2: A parallel machine fundamentally has compute and data locations connected by an interconnect.

Machine Model

In our model, we assume that a parallel machine includes two types of elements – compute and memory – tied together by an interconnect as illustrated in Figure 3.2. We further assume the hardware provides a global address space, and the ability to send messages and move data across the interconnect with low overhead.

Here is a detailed description for each element:

1. **Processing units**: We assume that the machine consists of some number of MIMD processing cores, and each core has a unique ID associated. A core can send a direct message to any arbitrary core provided with the ID through the interconnect. For the rest of the thesis, we use *lane* to denote a MIMD core and assign the index from the namespace `{0:num_lanes}` to each lane, denoted as the `LaneID`.

2. **Memory**: The machine also has some number of memory locations, denoted as `{0:num_memories}`. Lanes are attached to memory locations (can be one-to-one or many-to-one relationships) but can directly access all the locations with a unified global address space. In addition, the machine provides mechanisms to identify the affinity based on the data address so that programmers can instruct the programs to take advantage of data locality. We will describe later why this can be achieved and is useful.

Given this machine model, high performance is achieved by sufficient parallelism and good load balance to utilize all of the compute elements efficiently. In the rest of the section, we will describe the KVMSR programming interface and illustrate how it can eliminate programmers' challenges on writing high-performance parallel programs on irregular data.

3.2.1 KVMSR Interface: Expressing Computation

At a high-level view, each KVMSR program contains two phases, map and reduce. The map phase runs on the input key value set and generates an intermediate key-value set. The intermediate key-value pairs will be shuffled to the reduce phase which executes in parallel and produces an output key-value set. The procedure is illustrated in pseudo-code in Listing 3.1.

Listing 3.1: Pseudo-code for KVMSR. Execute map tasks in parallel, generates an intermediate key-value set, shuffles, and executes reduce tasks in parallel to produce an output key-value set.

```
event kv_map_shuffle_reduce(KVSet input_set, KVSet output_set) {
    KVSet inter_set;
    for(KVPair kv : input_set)
        send(kv_map, kv.key, kv.values); // generate intermediate KVSet
    shuffle(inter_set);
    for(KVPair kv : inter_set)
        send(kv_reduce, kv.key, kv.values); // generate output KVSet
    send_reply(output_set); }
```

To write a KVMSR program, programmers need to define the parallel computation tasks in terms of `kv_map()` and `kv_reduce()` functions. The interface is illustrated in Listing 3.2. Similar to the conventional MapReduce framework's interface, both map and reduce functions take a key and a list of values associated with that key, compute on the input key-value pair as defined in the function bodies, and then emit one or more key-value pairs to the intermediate or output key-value set (output can be eliminated as well).

Listing 3.2: Function `kv_map()` and `kv_reduce()` interface.

```
event kv_map(Key key, Types values) {
    ... map code ...
    send(kv_map_emit, inter_key, inter_values); }

event kv_reduce(Key inter_key, Types inter_values) {
    ... reduce code ...
    send(kv_reduce_emit, out_key, out_values); }
```

As illustrated in Listing 3.1, the `kv_map()` functions are called in parallel for each key in the input key-value set, producing an intermediate key-value set. These are shuffled to bring together values for any single key. The `kv_reduce()` function is called in parallel on each intermediate key-value pair to produce the output set. While many uses are possible, `kv_map()` typically expresses independent parallel computation, and `kv_reduce()` merges values, handling any needed serialization in the program.

3.2.2 KVMSR Interface: Global Data Structures

KVMSR expects the input and output to follow the `KVSet` interface. Logically, each set contains an arbitrary number of key-value pairs, each includes a key and an arbitrary size of values. The keys are used to iterate over the entire `KVSet` (i.e., sequential access function `get_next()`) and retrieve a particular key-value pair from the `KVSet` (i.e., random access function `get_values()`). Listing 3.3 gives a brief skeleton of the interface for `KVSet`. Note that the skeleton uses pointers (`Key*` and `Values*`) for simplicity. Nonetheless, the interface does not constrain the data layout and underline data structure and it can be implemented as array, trees, table, queues etc. It is left to the programmers to define the access approach in the random and sequential access functions based on the data layout they choose for their programs.

Listing 3.3: Data structure `KVSet` interface.

```
struct KVSet{
    Key* key;  Values* values; }

event get_next(Key key) {
    ... calculate next key ...
    send_reply(key);
    }

event get_values(Key key) {
    ... calculate address ...
    send_reply(Value*); }
```

To write a KVMSR program, programmers first should extend the `KVSet` interface, implement the abstract functions for random and sequential access with keys for each customized subclass of `KVSet`, and then pass the input and output `KVSet` to the KVMSR entry point `kv_map_shuffle_reudce()`.

As described in Section 3.2, KVMSR assumes the machine supports a global address space with unified naming (i.e., addresses) to access all the data. Therefore, `KVSet` can be allocated locally in a memory location or stride across all the memory locations and anywhere in between. Regardless of the data layout, KVMSR can address the data within the global address space, greatly simplifying data management.

Global data structures also make KVMSR much more powerful than the conventional cloud MapReduce [9, 10], because in the latter map and reduce computation is restricted to the data from the input value and cannot access globally shared data. On the other hand, KVMSR's map and reduce functions can access arbitrary data during the computation with the global address. More importantly, it implies that KVMSR's input values can include pointers to data in the global shared address space, and with pointers, map and reduce functions can directly interact and/or manipulate pointer-based data structures, such as self-synchronizing data abstractions, etc. For example, one can define the map task to read data from a shared hash table using one of the input values as the table index and define the reduce function to write the output to a multi-producer-multi-consumer queue supporting atomic operations. In Section 4.1, we give an example to demonstrate the benefits of operating on global memory.

Such flexibility is the key to KVMSR's programmability and generality. The support of pointers enables programmers to describe computations on complex data structures. It also implies that programmers can take their existing C or C++ implementation of a serialized program and place it inside KVMSR's map/reduce functions to parallelize the computation automatically with minimal code changes.

3.2.3 KVMSR Interface: Parallelism Management

Managing parallelism is crucial to parallel programs' performance. Too much parallelism on a lane will overflow the lane resources (queues for example) slowing down the progression, whereas insufficient parallelism will leave the lane underutilized limiting the throughput. Traditional parallel programming frameworks usually expose primitives for controlling the thread/process limits, e.g., threads, ranks, etc. Similarly, KVMSR also allows programmers to assign a range of lanes to the program and specify the maximum number of concurrently running threads on each lane (bounded by the hardware parallelism) using the function `set_max_thread()`.

Applications exploit the available parallel compute resources by binding tasks to computation locations and whether one can efficiently do so is critical to the program's performance. A static binding implied from the data location is sufficient if the program is regular and/or the parallelism is predictable but not for irregular applications with nondeterministic data-dependent irregularity. Therefore, to help write and optimize irregular parallel programs, KVMSR provides two customizable functions for programmers to control the binding of computation to resources, i.e., compute locations, enabling programmers to load-balance the system based on program knowledge and/or runtime information dynamically.

Customizing Computation Location Binding

As described above, KVMSR describes parallel computation in terms of map and reduce tasks, which are managed based on keys. Therefore, KVMSR provides two customizable functions `get_map_loc()` and `get_reduce_loc()`, both using the key to determine a location on which the task will be executed. The interface is illustrated in Listing 3.4.

Listing 3.4: Function `get_map_loc()` and `get_reduce_loc()` interface. Bind keys to compute locations.

```
event get_map_loc(Key key) {
    LaneID id = ...;
    send_reply(id); }
event get_reduce_loc(Key key) {
    LaneID id = ...;
    send_reply(id); }
```

With the `get_map_loc()` and `get_reduce_loc()` functions, the computation location binding for both phases can be statically specified or dynamically decided as below.

- **Static** Simple hashing or static distribution techniques can be used to spread unpredictable task sizes and number of task computations across the machine. The binding can also be determined statically based on the data location, given it does not change during the program execution.

- **Dynamic** Locations can also be dynamically determined based on machine compute load.Applications can use machine-specific features, such as `get_less_busy_lane()`, to acquire system load information and dynamically decide the binding to load-balance the system.

In summary, we introduced the KVMSR programming model and briefly explained the design objectives in this Chapter. Table 3.1 lists the key data abstraction, interface functions, and virtual functions to be implemented by programmers in KVMSR.

In the next Chapter, we will show how to utilize the power of customizing control of computation binding in KVMSR for parallelism management and computation load balancing.

Table 3.1: A summary of the KVMSR interface.

KVMSR Interface	Type	Description
KVSet	data structure	Input, intermediate, and output data structures should be defined following the KeyValueSet abstraction. User definition specifies how the data is laid out in the DRAM and the data type of key and value for each key-value pair.
event get_values(Key)	abstract function	Given a key from the `KVSet`'s key space, returns a pointer to its corresponding values (i.e., address).
event get_next(Key)	abstract function	Given a key from the `KVSet`'s key space, returns the next key in the `KVSet`.
event kv_map_shuffle_reduce()	interface function	Entry point of the KVMSR program.
event kv_map(Key, Values...)	abstract function	Defines the computation on input `KVSet`. Takes a key-value pair from input KVSet and may generate 1 or more key-value pairs to the intermediate `KVSet`.
event kv_reduce(Key, Values...)	abstract function	Defines the computation on intermediate `KVSet`. Takes a key-value pair from the intermediate `KVSet` and writes the result to the output `KVSet`.
event get_map_loc(Key)	abstract function	Given a key from the input `KVSet`'s key space, returns the lane for which the corresponding map task will be scheduled.
event get_reduce_loc(Key)	abstract function	Given a key from the intermediate `KVSet`'s key space, returns the lane for which the corresponding reduce task will be scheduled. All the intermediate key-value pairs with the same key will be scheduled on the same lane.

CHAPTER 4

EXAMPLES

In this Chapter, we describe two program examples to demonstrate KVMSR's modular programming interface and highlight its flexibility and efficiency in expressing and optimizing irregular parallel programs.

4.1 Convolution Filter

We start with a simple example, the convolution filter program, mainly to illustrate KVMSR's programming interface. The input and output pixel data are stored in global two-dimensional arrays. The KVMSR program applies a convolution filter on each sub-image of the same size and output another image.

4.1.1 Computation

The KVMSR pseudo code is shown in Listing 4.1. Each map function applies a 3x3 convolution filter to the sub-image centered at pixel $< x, y >$ and outputs the new value for that pixel. The outputs are then shuffled to the reduce function, which stores new values in the output image. Here, we use the center pixel's $< x, y >$ coordinates as the key.

Listing 4.1: Baseline KVMSR convolution filter program code

```
typedef struct { int x_idx, y_idx; } Key;
double input_image[M][N];


void kv_map(Key key, double[] values) {
    double[3][3] conv_kernel = load_filter();
    double[3][3] sub_img = to_matrix(values);
    double result = conv(sub_img, conv_kernel);
```

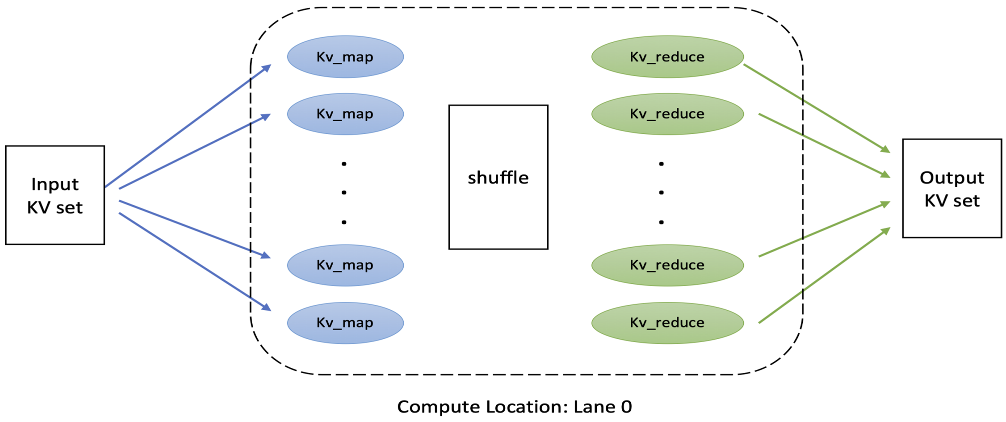


Figure 4.1: Computation binding for the baseline convolution filter program: placing all the tasks on a single lane.

```
    kv_emit(key, result);
    return; }


void kv_reduce(Key key, double value) {
    output_image[key.x_idx][key.y_idx] = value;
    return; }


LaneID get_map_loc(Key key) { return 0; }
LaneID get_reduce_loc(Key key) { return 0; }
```

4.1.2 Computation Binding

In Listing 4.2, we customize the `get_map_loc()` and `get_reduce_loc()` functions to distribute the computation tasks to the available lanes based on keys. With KVMSR's modularized interface, only the binding functions are changed, and the rest of the program, e.g., `kv_map()` or `kv_reduce()` computation and data layout, remains the same, as shown in Figure 4.2.

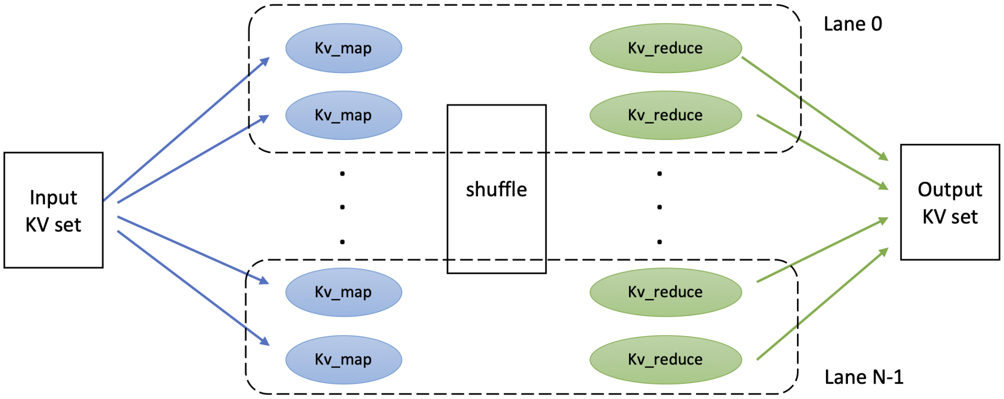


Figure 4.2: The statically distributed Convolution Filter program spreads computation over N lanes.

Listing 4.2: Parallel convolution filter program code with static computation location binding based on the key.

```
LaneID get_map_loc(Key key) {
    int idx = key.x_idx * input_img.dim[1] + key.y_idx;
    return idx % NUM_LANES;
}
LaneID get_reduce_loc(Key key) {
    int idx = key.x_idx * input_img.dim[1] + key.y_idx;
    return idx % NUM_LANES;
}
```

Using this simple example, we show that programmers can express program function in `kv_map()` or `kv_reduce()` task, conveniently use global data structures, and orthogonally control the computation location binding to parallelize the program across the compute resources.

4.2 PageRank

In a push-based PageRank program, each vertex reads its out-neighbors and sends the PageRank value along that edge. In the reduce stage, each vertex computes the average of incoming values from its in-neighbors in one pass and outputs the updated PageRank value [38] as shown in the KVMSR pseudo-code in Listing 4.3.

Listing 4.3: Baseline PageRank Program Code

```
typedef int Key;
struct Vertex{ int degree; int neighbors[]; }

void kv_map(Key key, double value) {
    Vertex v = input_graph.get_vertex(key);
    double out_pr_value = value / v.degree;
    for (int i = 0; i < v.degree; i++)
        kv_emit(v.neighbors[i], out_pr_value);
    return; }

void kv_reduce(Key key, double value) {
    Vertex u = output_graph.get_vertex(key);
    u.value = one_pass_avg(u.value, value);
    return; }
```

4.2.1 Computation Binding

One way to parallelize the computation is to distribute the keys (in this case vertices) evenly across the lanes and assign tasks accordingly based on the keys. The resulting program is presented in Listing 4.4 (the rest of the code remains the same and is omitted from the pseudo-code).

Listing 4.4: PageRank program with the default static computation binding based on key.

```
LaneID get_map_loc(Key key){return key % NUM_LANES; }
LaneID get_reduce_loc(Key key){return key % NUM_LANES; }
```

Alternatively, one can also spread the computation based on the data locations. For example, in Listing 4.5, we use the function `get_location(data)` to find the location of data and statically bind computation to where it is located.

Listing 4.5: PageRank with static computation binding based on data location

```
LaneID get_map_loc(Key key){
    return get_location(input_graph.get_vertex(key)); }
LaneID get_reduce_loc(Key key){
    return get_location(output_graph.get_vertex(key)); }
```

Static bindings work well for PageRank if the graph is regular and every vertex has the same number of neighbors. However, most real-world graphs have skewed degree distributions. Lanes that are assigned high-degree vertices will have a magnitude more work than the rest, resulting in an imbalanced load across the machine and bad parallel performance.

To solve the issue raised by irregular data, we present another variant of PageRank which uses `get_less_busy_lane()` to dynamically identify lanes with less load and bind computation accordingly to spread the load. The resulting program is shown in Listing 4.6.

Listing 4.6: PageRank program with dynamic computation binding based on the computate load

```
LaneID get_map_loc(Key key) { return get_less_busy_lane(); }
LaneID get_reduce_loc(Key key) { return get_less_busy_lane(); }
```

Using PageRank, we illustrate several ways of using the `get_map_loc()` and `get_reduce_loc()` to statically or dynamically distribute unpredictable irregular computation across computation resources.

4.3 A Broader Range of KVMSR Applications

We have presented two KVMSR programs and the procedure to tune different dimensions of a parallel program (computation, data, and computation binding referring to section 3.1 for performance in the sections above. For the sake of understanding, the examples we pick are relatively easy with minimal computation done per key-value pair. We want to highlight in this section that KVMSR can support a wide range of challenging irregular computations.

KVMSR belongs to the larger project on developing the next-generation high-performance supercomputer targetting sparse and irregular computation, namely UpDown. By the time the thesis is written, we have a range of challenging applications developed entirely or partially on KVMSR for the UpDown project and many more are under development. The computation varies noticeably, ranging from graph neural network training and inference, sparse matrix multiplication, graph pattern matching, genetic sequencing, multi-hop reasoning, graph transformation, influence maximization, graph adjustment, etc. Most of the development was done within days. Table **??** summarizes the applications and their characteristic.

KVMSR is based on a MapReduce-like programming interface and extended with a modular interface for customizing computation binding on a parallel machine. The former provides KVMSR with programmability and generality demonstrated in cloud MapReduce and functional languages, whereas the latter enables flexible performance tuning of data-dependent irregular programs on large-scale parallel machines. As a result, KVMSR enables a wide range of programs running on a fine-grained machine with minimal programming effort.

In the next Chapter, we conduct a detailed evaluation of KVMSR's scalability and programmability on the UpDown machine.

CHAPTER 5

EVALUATION

In this Chapter, We evaluate the KVMSR programming model, mainly focusing on two properties: 1) programmability, modularity, and expressiveness of the model and 2) the performance scaling on a fine-grained parallel machine, the UChicago UpDown machine (described in Section 2). The performance numbers are collected from the gem5 simulator with UpDown accelerator extension.

5.1 Evaluation System Specification

Table 5.1 lists the hardware details of the UpDown system. Briefly speaking, an UpDown machine is designed to have up to 16k nodes, each of which has 2,048 lanes, i.e., compute locations, organized in clusters of 64 (one UpDown accelerator) . As for the memory, there are 8 HBM2e stacks attached to each node, in total 128GB of memory. The high degree of fine-grained parallelism is possible at low power on UpDown because the lanes have no data caches, only a small 64KB scratchpad memory.

Key performance attributes of the machine include a high degree of multithreading in each lane with 1 cycle thread creation and termination, massive fine-grained MIMD lanes each with 64KB scratchpad memory for fast access, and a globally addressed memory with low latency – 70ns within a stack and 150ns to remote stacks.

In our experiments, we simulate various numbers of parallel compute resources up to 8,192 lanes (i.e., 4 nodes). We utilize the machine's special feature to identify a lightly loaded lane and find a lane near a data for implementing the `get_less_busy_lane()` and `get_location(data)` function used in PageRank and BFS.

all computations in the baseline program run on a single lane (serialized execution). Parallel versions distribute the computation using custom binding functions shown in Listing

Table 5.1: UpDown System specification

Total number of nodes	16,000
Accelerators per node	32
Accelerator cores per node	2,048
Scratchpad memory per accelerator	4 MB
Scratchpad memory per core	64 KB
DRAM capacity per node	128 GB
DRAM bandwidth per node	8.8 TB/s
Intra-node DRAM access latency	150.24ns
Inter-node DRAM access latency	1,100 ns
Cross node network bandwidth	4 TBps/node
Cross node network latency	26.5 ns

4.2, 4.5, and 4.6.

We then execute the programs on the UpDown simulator implemented with GEM5, evaluate performance, and report the runtime [32, 3]. The KVMSR programs exhibit fine-grained parallelism, indicated by the number of parallel tasks and the mean instructions per task. Pagerank and BFS are not only fine-grained but extremely irregular in their task size, as demonstrated by the huge standard deviation. Traditional scalable programming models such as MPI and PGAS are unable to exploit such fine-grained parallelism, as their per-message communication overheads alone are thousands to millions of instructions.

5.2 Programmability, Modularlity and Model Expressiveness

KVMSR's modular interface allows computation location binding to be expressed separately from the program's parallel computation (function). In this section, we measure the programming effort (in lines of code) to tuning dimensions of the parallel programs to highlight modularity and programmability of KVMSR.

5.2.1 Implementation

We implement the KVMSR model on the UpDown machine, called UDKVMSR (UpDown KVMSR), and evaluate three KVMSR programs: convolution filter, PageRank, and BFS. The convolution filter and PageRank program follow Listing 4.1 and Listing 4.3 in Section 4.

In addition to the two program examples in Section 4, we further implemented the synchronous push-based BFS in UDKVMSR. The program structure is similar to PageRank but with input and output from frontiers implemented using the parallel hash table abstraction, updating the distance and parent information instead of PageRank values. BFS uses the same computation-to-compute-location binding functions as PageRank.

5.2.2 Metric

To show the expressiveness of KVMSR, we count the lines of code for describing program computation/function and the binding of computation to lanes. UpDown programs are written in a C-like language and compiled by the compiler, called UDWeave, into the UpDown assembly programs. Listing 5.1 gives an example of a UDWeave program. The lines of UDWeave code are comparable to those of the corresponding C program, except that memory accesses are achieved by asynchronous messages with UpDown intrinsic functions.

Listing 5.1: UDWeave code for reading the neighbors of a vertex from the BFS UDKVMSR program. Functions `evw_update_event()`, `send_event()`, and `send_dram_read()` are intrinsic functions supported by the UpDown hardware and will be compiled to UpDown ISA instructions.

```
event kv_map(long degree, long vid, long *neighbors, long dist, long* vertex_addr)
    // If the vertex is visited or has degree 0, skip it
    if ((dist >= 0 && dist <= iteration) || degree == 0)) {
        long evw = evw_update_event(CEVNT, kv_map_return);
        send_event(evw, vid, CEVNT);
        yield;
    }

    int count = 0;
    long* nlist_ptr = neighbors;
    long evw = evw_update_event(CEVNT, rd_nlist_return);

    while (count < degree) {
        send_dram_read(nlist_ptr, DRAM_MSG_SIZE, evw);
        count = count + DRAM_MSG_SIZE;
        nlist_ptr = nlist_ptr + DRAM_MSG_BSIZE;
    }
}
```

5.2.3 Result

Table 5.2: Code Size for each tuned version of KVMSR programs (Lines of Code).

Application	Function	Serialized Baseline	Static Binding on Key	Static Binding on Data Location	Dynamic binding on compute load
Convolution Filter	24	0	2	2	N/A
PageRank	58	0	2	2	6
BFS	185	0	2	2	6

In Table 5.2, we present the line of code counts for the programs and their variations. A program's code includes its computation/function portion (defined in `kv_map()` and `kv_reduce()`), and the computation binding portion (defined in `get_map_loc()` and `get_reduce_loc()`). The baseline does not specify any computation binding and execute

Table 5.3: Programs and Key Properties

Program (Dataset)	Data Size	Num Tasks	Data/Task	Mean Inst/Task	StdDev Inst/Task
Convolution Filter (8Kx8K Matrix, 3x3 filter)	512MB	67,076,100	72B	58	0
PageRank & BFS (RMAT graph scale 20, 2^{20} vertices)	292M	1,048,576	305B	154	43,395

everything on 1 lane, so 0 lines are required. The static bindings in convolution filter, PageRank, and BFS programs each add 2 lines of code to specify computation location from keys or data location using `get_location(data)`. One line in each of `get_map_loc()` and `get_reduce_loc()`. PageRank's and BFS's dynamic binding version based on the compute load (see Listing 4.6) adds 6 lines using the UpDown intrinsic function `get_less_busy_lane()`.

Table 5.2 highlights KVMSR's modular interface, allowing the definition of computation binding orthogonal to the program function, i.e., only the binding functions are modified leaving the main body of the program unchanged.

5.3 Benefits of Computation Location Control

In this section, we run the above UDKVMSR programs on the UpDown machine and collect the performance statistics from the gem5-based UpDown simulator.

5.3.1 Experiment Setup

Table 5.3 summarizes the datasets we used for the experiments. Convolution filer is running on a regular matrix with 8,192x8,192 (=67,108,864) entries and of size 512MB. PageRank and BFS both are running on the synthetic graph generated using the RMAT generator with parameters from Graph500 specification ($a = 0.57, b = c = .19$, and $d_{average} = 16$) [5, 13]. The generated graphs resemble real-world graphs and exhibit highly-skewed degree distributions.

We customized the GEM5 [3], a cycle-accurate hardware simulator, to simulate the UpDown machine. The simulator configurations follow the machine specification in Section 5.1 except that it only simulate up to 4 nodes, that is 8,192 lanes (MIMD cores) due to time and memory constraints. The baselines for our evaluation are the corresponidng C++ program for the same three applications (Convolution, PageRank and BFS) and running on a single core CPU. We collect the performance numbers also from the GEM5 simulator. The simulated system is a x86 Out of Order CPU (single core) with 64KB L1 cache, 256KB L2 cache and 8MB L3 cache at 2 GHz. The baseline C++ program is single-threaded, i.e., serializes the execution.

5.3.2 Result

We conducted two experiments to evaluate KVMSR's potential for performance scaling on thousands-fold parallelism and the impact of customized computation binding functions on parallel programs.

1. **Experiment I** We run the Convolution, PageRank and BFS UDKVMSR programs described in Section 4 with the static computation binding function on keys shown in Listing 4.2 for Convolution and Listing 4.4 for PageRank and BFS. We simulate the programs on the gem5 and measure their performance running on 64, 128, 256, 512, 1,024, and 2,048 UpDown lanes.

2. **Experiment II** We focus on PageRank and BFS, i.e., the data-dependent irregular programs, and comparing the performance of the same UDKVMSR programs except different choices of computation binding functions. We further scale up the machien scale to 8,192 lanes (4 UpDown nodes).

Performance Scaling of KVMSR Programs

Figure 5.1 presents the speedup of the UDKVMSR programs over the single-threaded CPU baseline programs for Convolution, PageRank and BFS respectively with the same static binding function based on key (i.e., statically divided the keys across the lanes).

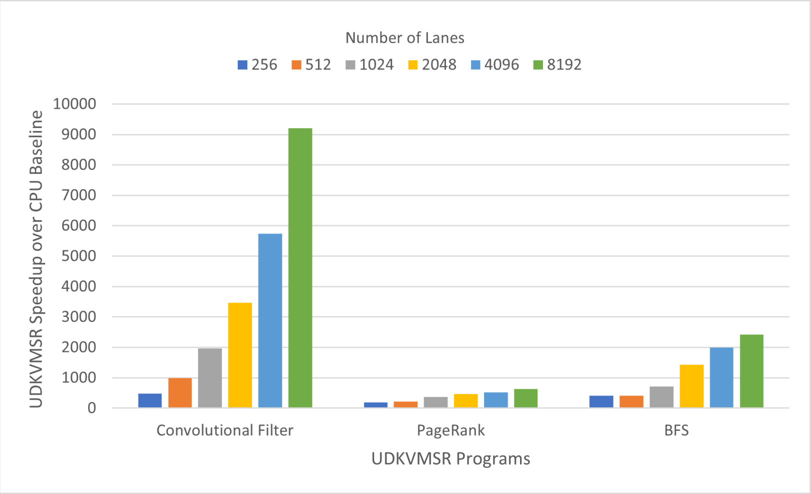


Figure 5.1: Speedup of the UDKVMSR Convolution, PageRank and BFS with static binding based on Keys over the single-thread CPU baseline

Compared to the single-thread C++ baseline program, the UDKVMSR convolution program gains 477x performance speedup on 256 lanes and 9,202x on 8,192 lanes program, achieving excellent parallel performance. On the other hand, the UDKVMSR PageRank and BFS program gained 195x performance speedup on 256 lanes and 635x on 8,192 lanes, and 408x performance speedup on 256 lanes and 2,423x on 8,192 lanes respectively, achieving moderate performance improvement from a 32x increase of hardware parallelism.

Taking a closer look at the performance scaling, we compare the 256-lane performance to that of the 8,192 lanes for the UDKVMSR programs. Convolution scales well, gaining 19.4x improvement for a 32 times increase in hardware parallelism. On the other hand, PageRank and BFS with the static computation binding fail to scale, gaining only 3.24x and 5.93x speedup respectively. This results from the data-dependent work unbalancing since

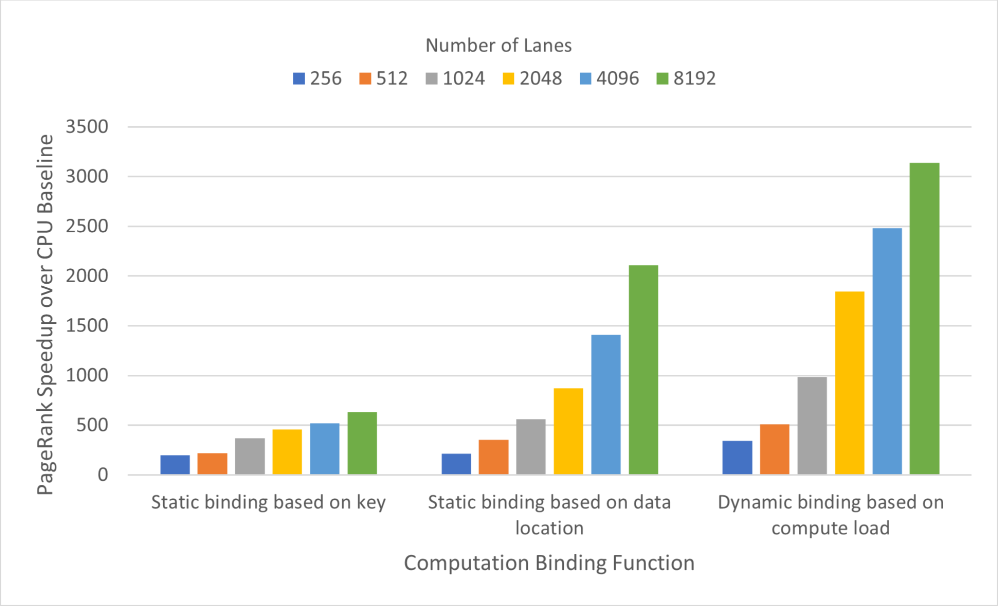


Figure 5.2: Speedup of PageRank with static binding based on Keys (left bars), static binding based on data location (middle bars), and dynamic binding based on compute load (right bars) over the single-thread CPU baseline.

both PageRank and BFS operate on the RMAT graph with skewed degree distribution. The naive binding approach which statically divides the tasks across the lanes leads to some unlucky lanes getting assigned high-degree vertices and having magnitudes of more work than the rest. The poor performance scaling is mostly a result of lanes pending for the long-run task to finish.

Managing Parallelism

As shown in the previous experiment, the two graph UDKVMSR programs with static computation binding based on key approach scale poorly as the hardware parallelism increases. The reasons are twofold:

1. The task size is determined by the highly-skewed input vertex's degree, resulting in irregular task size (see Table 5.3).
2. Both programs access memory intensively with limited computation associated with

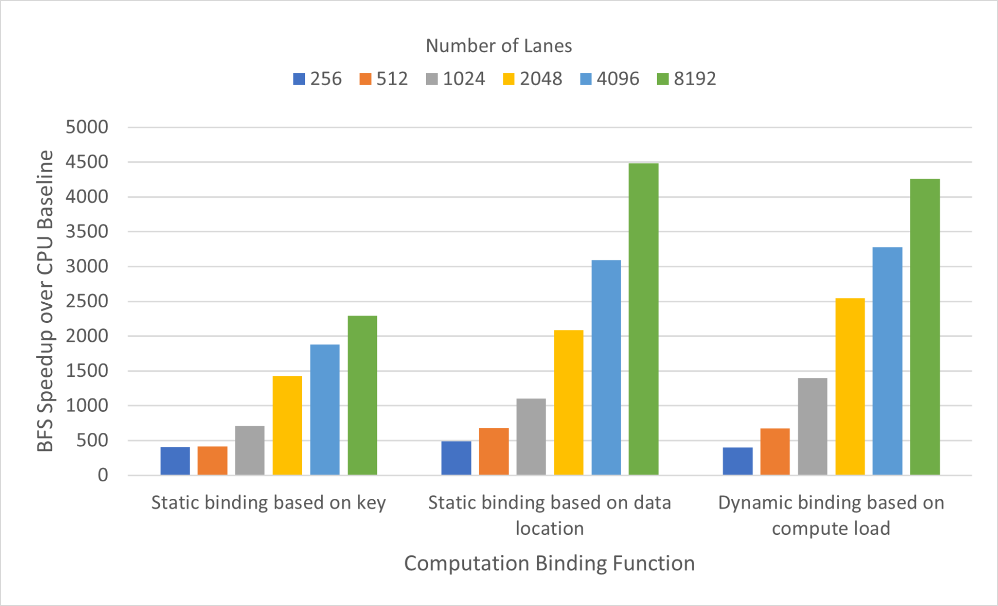


Figure 5.3: Speedup of BFS with static binding based on Keys (left bars), static binding based on data location (middle bars), and dynamic binding based on compute load (right bars) over the single-thread CPU baseline.

each data so DRAM access latency dominates.

To study the impact of computation binding on irregular parallel programs performance, we designed the second experiment: changed the computation binding functions and measured the programs' performance and scaling. We again normalize the performance of BFS and PageRank programs running on various sizes of the machines but this time for different computation binding functions, including both static and dynamic. For better comparison, the left bars in Figure 5.2 and 5.3 shows the same data as the PageRank and BFS performance in Figure 5.1, where both programs use the static binding function based key to evely divided the keys to lanes.

The first alternative we explored is binding computation to locations close to the data. The data is depicted by the middle bars in Figure 5.2 and 5.3. With improved data locality, PageRank and BFS achieve an average of 2.6x performance improvement compared to the static binding based on keys. In the case of 4 nodes (8,192 lanes), the performance of

PageRank improves by 3.2x due to reduced memory access latency. BFS scales less but still improves by 1.96x over the naive binding appraoch. However, static binding functions lack the run-time information to distribute the computation evenly, so in the case of less than 1 node ($\leq$ 2,048 lanes), load imbalancing across the lanes significantly limits the performance, leaving room for improvement.

The other binding approach we examined is dynamically binding tasks to on less busy lanes based on the run time system load status. The data is shown in the right bars in Figure 5.2 and 5.3. The additional runtime information dramatically improves PageRank's performance: about 4.9 times the speedup for 2,048 lanes from 343x to 3,136x. The improvement on BFS is less compared to the static binding based on data location approach, i.e., from 339x to 4,258x, since the BFS's parallelism (i.e., frontier size) is spread across the iterations and data accesses dominates the latency.

In summary, KVMSR's flexible and modular interface allows this dramatic change of computation location binding with a few lines of code, bringing huge performance improvements to the programs.

CHAPTER 6

RELATED WORK AND DISCUSSION

In this Chapter, we review the related work on parallel programming models and frameworks and graph processing systems.

6.1 Functional Language and Cloud Map-Reduce

Functional languages allowed functions to be applied to sets/arrays (map) and combine the results (reduce) [25, 24, 7]. Originated for expressive power, these constructs can be used to express parallelism and exploit it on multi-core and large NUMA shared-memory machines. However, these machines have limited scalability with the largest systems around 256 cores. Our studies are for 8,192 compute locations, and a full system design has over 30M compute locations, which have several magnitudes more hardware parallelism than any existing multi-core machines.

Later, cloud companies built a different map-reduce, designed for scale-out/internet-scale computations and focus on fault tolerant [9, 10]. The key motivation was to exploit the natural and flexible expression of parallelism and effectively map the parallelism onto scale-out distributed systems. These systems solved the important problems of reliability (map and reducers), but with the significant restriction of no shared data structures (across map or reduce functions). Compared to the functional languages predecessors, the cloud systems added string keys, using them to express computation function and to control parallelism indirectly and at coarse-grained. However, these systems manage load balance automatically, depending on hashing and balanced sorts, eschewing programmer involvement. This works adequately for cloud system because their MapReduce typically operate on coarse-grained tasks, running billions of instructions, many orders of magnitude larger than the 100 instruction fine-grained tasks we are pursuing in UDKVMSR.

None of these functional or cloud map-reduce frameworks provide any way for programmers to control the location of compute or data. KVMSR uses keys to control and manage the parallelism. Users can direct the computation location mapping, balance the load across the system, and synchronize data reduction all with keys. Such control is the core contribution of KVMSR.

6.2 Message Passing (MPI) & Partitioned Global Address (PGAS)

A popular model for scalable parallelism (and high-performance computing – HPC), is message passing. Typically, the single-program multiple data (SPMD) divides data across separate processes with private address space [14]. Each process computes on local, private data, and in the pure message-passing model, all remote (global) data is accessed via explicit messages.

The message-passing model makes programming complex distributed structures tricky (e.g., trees, graphs, hierarchical data). For computations using such structures for algorithmic efficiency, programming with distributed data and computation is challenging. The model provides no support for global naming, so different names must be used (typically software-interpreted) for any global data structures. As a result, programs using sophisticated pointer-based structures are difficult to express in this model [27]. If work is dynamically generated and tied to such structures, e.g., irregular work and parallelism, programming is even more challenging [21]. If the data or computation is irregular, this produces complex programming and communication (see high-performance implementations of irregular and graph applications [27, 26]).

An important extension of the message-passing model adds a partitioned global address space (PGAS), federating the local process address spaces as in Global Arrays, UPC++, and ADLB [30, 1, 21]. PGAS programs provide the convenience of global naming, easing the programming of complex data structures and irregular parallelism. However, this convenience does not alter the underlying performance challenge, as to achieve speedup work must be aligned and balanced across the address spaces. This is because ultimately the computation is done by cores which can only access data in a single private address space.

6.3 Linda & Tuple Space

Linda is another well-established model in the parallel programming world, focusing on coordinating communication between processes [11]. The key concept in Linda is its tuple space abstraction, a data repository shared between processes where each process can independently generate and/or take elements (i.e., tuples) from it. The resulting advantage is communication orthogonality, meaning that processes involved in communication are decoupled in both time and space dimensions.

The logical view of KVMSR's key space, to some extent, resembles Linda's tuple space. Despite the similarity, tuple space focuses on concurrent access, production, and modification of shared tuples. On the other hand, the key space in KVMSR is mainly for efficiently managing parallel computation on key-value pairs. One can bundle keys together and partition the key space in different granularities for KVMSR to exploit parallelism at various levels. This level of management is not a focus for tuple space, where communication is at the granularity of each tuple.

6.4 Scalable Graph Processing Systems

While many graph-processing systems have been constructed, many of them focus on efficiency and do not scale to large numbers of parallel nodes [17, 35]. Of those designed to scale, those based on map-reduce are designed to scale, but suffer from massive inefficiency as each vertex and edge operation can cost a TCP message in a cloud computing cluster

[22, 33]. The two implications are that high performance requires the use of datacenter scale resources (10,000 nodes to outperform a 128-node SMP) and because they are built on map-reduce, load balance is performed by the system, and programmer input is not possible. At the lower efficiency and coarse-grained execution of these systems, sampling with balanced sorting gives adequate balance. Customized graph computing systems have been developed that include a custom programming model, vertex-centric and iterative, and achieve moderate scalability on conventional hardware (16x on either 16 or 64 nodes), largely benefiting from the increased memory to compute larger problems [20, 12]. KVSMSR targets general irregular algorithms and data, a much larger class of applications.

6.5 Discussion

In general, flexibility and modularity in program structure are considered a virtue in languages and programming models – and application software architecture. In message-passing programs, code expression locks in data layout/locality choices, and consequently computation mapping (freezing the data mapping). In PGAS programs, this problem is lessened, but achieving good performance requires data movement and work management to align with the parallel compute structure.

By design, KVMSR uses a global address space to enable computation to be expressed independently of performance tuning. Thus computation binding to compute resources can be done flexibly with keys. This supports rapid exploration to find good choices, enabling adaptation to different data properties, hardware properties, or even dynamical runtime states.

CHAPTER 7

SUMMARY AND FUTURE WORK

7.1 Summary

This thesis presents the architecture of the UpDown accelerator, as an event-driven programmable processor, with a notion of how it fits into the larger UpDown system. The key ideas demonstrated are summarized below.

- The novel UpDown ISA and architecture mechanisms in the UpDown accelerator help achieve our goal of software customization in the memory hierarchy. While this customization can be applied across applications in general, we demonstrated this on a set of irregular graph processing applications and SpMV using encoded memory

- The key features of the UpDown accelerator, namely low-latency events, short thread activations, split transaction messaging, and flexible software programmability enable it to achieve high thread creation rates (43Bi/s) and potentially saturate the available memory bandwidth (381GB/s) of a single HBM2e DRAM stack.

- We demonstrated in our evaluation, that the UpDown features enable fine-grained parallelism and software customization, achieving performance scale-up as a function of parallelism in large-scale irregular workloads like graph processing. We showed speedups of up to 631x and geomean 282x over a single x86 core. Additionally, we showed the low area (1.25%) and power (17%)of the UpDown (compared to [28]) which allows cost-efficient scaling of the system.

- Finally, the UpDown ISA is also generic enough that it is not constrained to any particular set of applications or data structures, as shown by its use across a variety of workloads and data representations. This is in contrast to the typical domain specialized architectures (DSAs) being proposed in literature that have hardwired mechanisms targeting a specific class or applications and data structures.

7.2 Future Work

Several research problems still remain to be explored surrounding the UpDown accelerator. Some of them are discussed below.

1. The evaluation of the UpDown accelerator vs a single-threaded x86 core provides valuable insights into the efficacy of the UpDown ISA and architecture mechanisms. There is scope to expand this evaluation to a broader class of architectures.

2. The UpDown cluster showed significant performance scaling going from 1 to 4 UpDowns. However, multiple aspects of the larger UpDown system such as the interconnect architecture within a cluster, node and between nodes, scalability of the UpDown mechanisms beyond a node, interaction with multiple DRAM stacks, remain to be explored and evaluated.

3. The integration of an accelerator into a heterogeneous system is an interesting research problem in general. In the UpDown system, we integrate the accelerator into the memory hierarchy between the Top cores and DRAM. There are interesting research questions around how UpDown can communicate notifications and data to the Top core and its caches to keep up with the high memory bandwidths that we have measured. There are also interesting questions around the memory consistency and coherence models as the scale of the system grows.

4. The high level programming model for the UpDown system is also an open research question. This thesis has focused on the architecture mechanisms and used hand-coded assembly kernels to showcase performance benefits. The precise compiler infrastructure, programming model and runtime system are open problems to study.

5. Finally, while we have studied graph analytics and SpMV as representative applications to evaluate the UpDown architecture, studying a broader class of applications like graph neural networks, data mining etc. are of interest to prove further the generality of UpDown mechanisms for irregular applications.

REFS

[1] John Bachan, Dan Bonachea, Paul H. Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B. Baden. “The UPC++ PGAS Library for Exascale Computing”. In: *Proceedings of the Second Annual PGAS Applications Workshop*. PAW17. Denver, CO, USA: Association for Computing Machinery, 2017. ISBN: 9781450351232. DOI: 10.1145/3144779.3169108. URL: https://doi.org/10.1145/3144779.3169108.

[2] Albert-László Barabási and Réka Albert. “Emergence of Scaling in Random Networks”. In: *Science* 286.5439 (Oct. 1999), pp. 509–512. DOI: 10.1126/science.286.5439.509.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi-org.proxy.uchicago.edu/10.1145/2024716.2024718.

[4] Rupak Biswas, Leonid Oliker, and Hongzhang Shan. “Parallel computing strategies for irregular algorithms”. In: *Annual review of scalable computing* 5 (2003), p. 1.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. “R-MAT: A recursive model for graph mining”. In: vol. 6. Apr. 2004. DOI: 10.1137/1.9781611972740.43.

[6] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. “Power-Law Distributions in Empirical Data”. In: *SIAM Review* 51.4 (2009), pp. 661–703. DOI: 10.1137/070710111. URL: http://link.aip.org/link/?SIR/51/661/1.

[7] *The C++ Reference Manual.* available from https://en.cppreference.com/w/.

[8] Timothy A. Davis. “Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra”. In: *ACM Trans. Math. Softw.* 45.4 (Dec. 2019). ISSN: 0098-3500. DOI: 10.1145/3322125. URL: https://doi.org/10.1145/3322125.

[9] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[10] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. “Efficient Big Data Processing in Hadoop MapReduce”. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 2014–2015. ISSN: 2150-8097. DOI: 10.14778/2367502.2367562. URL: https://doi-org.proxy.uchicago.edu/10.14778/2367502.2367562.

[11] David Gelernter. “Generative Communication in Linda”. In: *ACM Trans. Program. Lang. Syst.* 7.1 (Jan. 1985), pp. 80–112. ISSN: 0164-0925. DOI: 10.1145/2363.2433. URL: https://doi-org.proxy.uchicago.edu/10.1145/2363.2433.

[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 17–30. ISBN: 978-1-931971-96-6. URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez`.

[13] *Graph 500 Results*. `https://graph500.org/`.

[14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN: 0262527391.

[15] Tomaž Hočevar and Janez Demšar. "A combinatorial approach to graphlet counting". In: *Bioinformatics* 30.4 (Dec. 2014), pp. 559–565. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/btt717`. eprint: `https://academic.oup.com/bioinformatics/article-pdf/30/4/559/48917199/bioinformatics\_30\_4\_559.pdf`. URL: `https://doi.org/10.1093/bioinformatics/btt717`.

[16] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a Social Network or a News Media?" In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. Raleigh, North Carolina, USA: Association for Computing Machinery, 2010, pp. 591–600. ISBN: 9781605587998. DOI: `10.1145/1772690.1772751`. URL: `https://doi.org/10.1145/1772690.1772751`.

[17] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. "GraphChi: Large-Scale Graph Computation on Just a PC". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 31–46. ISBN: 9781931971966.

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

[19] G. Linden, B. Smith, and J. York. "Amazon.com recommendations: item-to-item collaborative filtering". In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80. DOI: `10.1109/MIC.2003.1167344`.

[20] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. "GraphLab: A New Framework for Parallel Machine Learning". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. UAI'10. Catalina Island, CA: AUAI Press, 2010, pp. 340–349. ISBN: 9780974903965.

[21] Ewing Lusk, Ralph Butler, and Steven C Pieper. "Evolution of a Minimal Parallel Programming Model". In: *Int. J. High Perform. Comput. Appl.* 32.1 (Jan. 2018), pp. 4–13. ISSN: 1094-3420. DOI: `10.1177/1094342017703448`. URL: `https://doi.org/10.1177/1094342017703448`.

[22] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 135–146. ISBN: 9781450300322. DOI: 10.1145/1807167.1807184. URL: https://doi.org/10.1145/1807167.1807184.

[23] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 135–146. ISBN: 9781450300322. DOI: 10.1145/1807167.1807184. URL: https://doi-org.proxy.uchicago.edu/10.1145/1807167.1807184.

[24] Simon Marlow et al. "Haskell 2010 language report". In: *Available online http://www. haskell. org/(May 2011)* (2010).

[25] John McCarthy, Paul W Abrahams, Daniel J Edwards, Timothy P Hart, and Michael I Levin. *LISP 1.5 programmer's manual*. MIT press, 1962.

[26] Marco Minutoli, Maurizio Drocco, Mahantesh Halappanavar, Antonino Tumeo, and Ananth Kalyanaraman. "CuRipples: Influence Maximization on Multi-GPU Systems". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392750. URL: https://doi.org/10.1145/3392717.3392750.

[27] Marco Minutoli, Mahantesh Halappanavar, Ananth Kalyanaraman, Arun Sathanur, Ryan Mcclure, and Jason McDermott. "Fast and Scalable Implementations of Influence Maximization Algorithms". In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–12. DOI: 10.1109/CLUSTER.2019.8890991.

[28] Hassan Mujtaba. "Intel Sapphire Rapids '4th Gen Xeon' CPU Delidded By Der8auer, Unveils Extreme Core Count Die With 56 Golden Cove Cores". In: *WCCFTech* (2022). URL: https://wccf.tech/189cd.

[29] M. E. J. Newman. "The Structure and Function of Complex Networks". In: *SIAM Review* 45.2 (Jan. 2003), pp. 167–256. ISSN: 1095-7200. DOI: 10.1137/s003614450342480. URL: http://dx.doi.org/10.1137/S003614450342480.

[30] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. "Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers". In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing '94. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 340–349. ISBN: 0818666056.

[31] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala language specification*. 2004.

[32] Andronicus Rajasukumar. *UPDOWN: AN INTELLIGENT DATA MOVEMENT ARCHITECTURE FOR LARGE SCALE GRAPH PROCESSING*. Tech. rep. TR-2023-03, Available from `https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2023-03`. University of Chicago, Computer Science, 2023.

[33] *Scaling Apache Giraph to a Trillion Edges*. `https://engineering.fb.com/2013/08/14/core-data/scaling-apache-giraph-to-a-trillion-edges/`. 2013.

[34] Julian Shun and Guy E Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: (), p. 12.

[35] Julian Shun and Guy E. Blelloch. "Ligra: A Lightweight Graph Processing Framework for Shared Memory". In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 135–146. ISSN: 0362-1340. DOI: `10.1145/2517327.2442530`. URL: `https://doi.org/10.1145/2517327.2442530`.

[36] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. "GraphMat: high performance graph analytics made productive". In: *Proceedings of the VLDB Endowment* 8.11 (July 2015), pp. 1214–1225. ISSN: 2150-8097. DOI: `10.14778/2809974.2809983`. URL: `https://dl.acm.org/doi/10.14778/2809974.2809983` (visited on 04/03/2022).

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. "Gunrock: a high-performance graph processing library on the GPU". In: *SIGPLAN Not.* 51.8 (Feb. 2016). ISSN: 0362-1340. DOI: `10.1145/3016078.2851145`. URL: `https://doi.org/10.1145/3016078.2851145`.

[38] B. P. Welford. "Note on a Method for Calculating Corrected Sums of Squares and Products". In: *Technometrics* 4.3 (1962), pp. 419–420. DOI: `10.1080/00401706.1962.10490022`. eprint: `https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022`. URL: `https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022`.