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. His insight to computer architecture and system has greatly broaden my knowledge and reshaped my understanding of the fields.

I would also like to thank my colleagues and friends in LSSG research group and the UpDown project - Andronicus, Tianshuo, Marziyeh, Jose, Jerry (Ruiqi), Jiya, Tony, Jerry (Jianru), Ziyi, and Ahsan - for feedback and collaboration that have been integral to the completion of this thesis, and most importantly for supporting me all way through my research.

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) are 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. Parallelism is managed via the keys with a modular programming interface: KVMSR enables programs to flexibly bind computation to compute resources based on keys for load balance and data locality.

In this thesis, we illustrate and evaluate the KVMSR modular interface 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 7,845x 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

In the past few decades, there has been a rise in "internet-scale" computation on big data, producing a growing need for computing systems that can process large-scale data efficiently. Among the emerging applications, graph computation is ubiquitous: complex relationships in various real-world scenarios are represented by graphs efficiently, for example, social networks, the world wide web, recommendation systems, bio-informatics, etc. [16, 20, 29, 15]. As the data keeps growing, real-world graphs can occupy gigabytes of memory and if this trend continues, real-time analysis of petabyte graphs will be the norm in the next decades. Therefore, high-efficient low-latency real-time processing of graphs continues to demand larger and more powerful machine presumably with millions of compute cores and petabytes of memory, significantly beyond the scale of current parallel machines (typically of hundreds or thousands cores and gigabytes of memory).

Graph computations exhibit high data parallelism at the vertex and/or edge level. The resulting fine-grained computation parallelism is exploited in varied graph processing software frameworks [24, 37, 8, 35, 38]. Despite the abundance, it is extremely irregular: real-world graphs often has skewed degree distribution (e.g., power-law), meaning that most vertices have a few neighbors except a couple outliers are connecting to magnitudes more of vertices [5].

Graph computation's data irregularity and fine-grained nature significantly restricts the scaling of its parallel performance. Without dedicated control, the extreme imbalance between the high-degree and low-degree vertices would result in some of the computing units being assigned magnitudes of work more than the rest of the machine, limiting the machine utilization and the overall performance. Furthermore, many graph algorithms exhibit memory-intensive computation: each vertex/edge involves limited computation on scattered data with limited data reuse (e.g., one addition for each edge in PageRank), leading to

poor data locality and costly communications and data movement. Compared to the regular parallel applications (e.g., dense matrix multiplication), load-balance such application and achieve scalable performance on a parallel machine is hard. As the scale of the computation and machine scales with the growing data (potentially to million-fold parallelism on petabytes data), the issues would become even more critical.

The two main stream parallel programming solutions are message-passing interface (MPI) and partitioned global address space models (PGAS). Despite the popularity, they do not support irregular applications well (e.g., skewed real-world graph analytics and graph pattern minings).

- MPI follows the single-program-multiple-data model (SPMD) model, that is data is partitioned across the machine and the parallel computation on each node is tied to the local data. Remote data are exchanged via costly messages. With scant support for global data structures, programmers must assemble the data required for each piece of parallel computation and align it to a static set of workers (ranks) to achieve high performance, extremely challenging for irregular data [4].

- PGAS model provides a global address space that aids in expressing parallel computation over the distributed global data structures (e.g., graphs) [30]. Nonetheless, PGAS lack effective support for fine-grained parallelism management which is important for data-irregular parallelism. For example, naively distributing graph's vertex-level irregular computation across the machine would result in load-imbalance, hindering the parallel efficiency.

Therefore, dealing with skewed computation is possible in MPI and PGAS, but with scant support from model it is extremely challenging to balance the irregular data and the result computation globally to achieve performance [1, 28].

A promising direction is MapReduce: many parallel applications have been expressed under this framework in the past several decades mostly due to the prosperity of func-

tional 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 tolerance and fail to exploit fine-grained parallelism efficiently with its coarse-grained approach [25, 31, 9].

1.1 The Problem

While graph computation has abundant data and computation parallelism, exploited by various software graph processing systems such as Pregel and Powergraph, none of these high-level systems provide programmer control over parallelism and locality. Lower-level parallel programming models (e.g., MPI, PGAS and MapReduce) enable programmer-controlled parallelism for performance but generally at coarse-grain, inefficient for graph computation's fine-grained irregularity.

In short, we aim to solve the challenge in achieving scalable performance of irregular applications, mainly graph applications, on future large-scale machines. Most importantly, we are targeting extreme-scaled machines with million-fold hardware parallelism and petabytes of memory. The proposed solution should achieve the following properties lacking in existing ones:

1. efficiently expresses and manages large-scale fine-grained parallelism in software
2. spreads the data across the petabyte memory for memory-level parallelism without complicating the application program

3. effectively binds the software parallelism onto the available compute resource under extreme irregularity

4. supports flexible tuning of data layout and computation binding (i.e., bullet points 2&3) to optimize performance at various machine and data scales.

1.2 Approach

We propose the key-value map-shuffle-reduce (KVMSR) framework for irregular computation parallelism on future large-scale parallel systems. These systems will use novel building blocks for fine-grained parallelism and scale to 30 million parallel computation elements [33]. The key elements of our approach are:

- User-defined key space for flexible management of fine-grained software parallelism, in contrast to MPI's static data-based management.

- MapReduce-like programming interface with support of rich-structured distributed data in a global address space, in contrast to the cloud MapReduce's rigid string-based key-value pairs.

- Customizable computation binding functions using keys to allow fine-grained parallelism control such as alignment or computation affinity.

- Modular programming interfaces enabling orthogonal control of data layout, parallel computation, and the binding to the parallel computation resources.

1.3 Key Contributions

This thesis presents the KVMSR model and illustrates the design using selected program examples, namely convolution filter, PageRank, and BFS. We then demonstrate the model's

modular interface for flexibility support to performance-optimize challenging irregular computations.

Specific contributions of this thesis include:

- Design of KVMSR for expressing fine-grained parallelism using key-value abstraction and MapReduce-like interface at extreme scales.
- Design of parallelism management with keys and a simple and modular interface for specifying computation binding independently from program computation and data layout.
- Evaluation that shows on an 8,192-way parallel compute system, KVMSR achieves up to 7,845x performance speedup over x86 CPU core with static computation binding approaches and an increase of speedup with dynamic approaches for computation location binding.
- Analysis of a variety of parallel programs implemented in KVMSR and their usage of KVMSR's modular programming interface to express fine-grained computation and manage irregular parallelism for performance.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 gives a brief overview of the background. Then, the KVMSR model is defined in Chapter 3, followed by three program examples described in Chapter 4. An implementation of the KVMSR model is evaluated in Chapter 5, to show its flexibility and performance benefits. An analysis of a broader list of KVMSR programs and the programming practices is given in Chapter 6. Finally, Chapter 7 discusses related works in graph computation and parallel programming, and Chapter 8.1 summarizes the thesis and points out directions for future work.

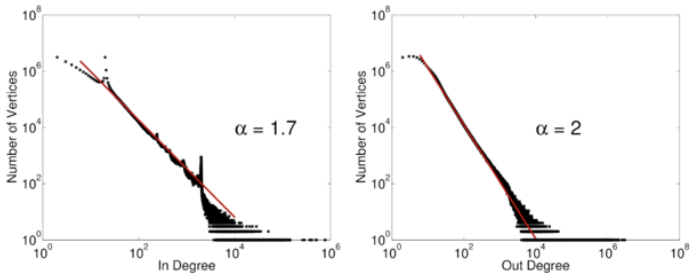
CHAPTER 2

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 [24]. 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 [24].

```
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 programs

are implemented in single-program-multiple-data (SPMD) and message-based programming model to mitigate the high cost of communication and remote data access. Typically, , the graph data is statically partitioned across the system and each node will be assigned with a subset of the vertices and responsible for executing the corresponding vertex programs. Remote data accesses are done via 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 underutilizing 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) [7, 26, 25]. 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 enables programmers to easily express massive parallelism in terms of map and reduce functions. Indeed, the cloud MapReduce framework was used by companies like Google, Facebook, Yahoo, and Amazon to process large graph data in the order of terabytes for a decade[32, 19]. Despite the richness of the domain, cloud MapReduce's coarse-grained work management performs poorly on skewed-distributed graph data. Most importantly, there is scarce support for optimizing the irregular computation performance such as user-defined computation-to-compute-resource binding to incorporate application knowledge to load-balance the system.

2.3 Fine-grained Parallel Hardware Architectures

Despite the critique of existing software frameworks' coarse-grained parallelism being inefficient for irregular applications, they were developed for machines in which fine-grained parallelism management is expensive. To be specific, conventional parallel machines constraints software in the following aspects: 1) expensive remote communication due to long latency and low bandwidth network, 2) no support for global address space, and 3) hardware parallelism limited to CPU and simultaneous multi-threading level (i.e., up to thousands of parallel threads). Under the above machine characteristics and limitations, programs whose computation is partitioned based on data location and whose parallel computation operates solely on local data with coarse-grained messages for exchanging remote data are efficient. In other words, the performance challenge of scalable irregular applications originates from the hardware architecture design. Therefore, many novel architectures are proposed to unfold new opportunities for 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 networks in contrast to the software-based message solution on traditional long-latency interconnect, 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, and 4) support of unified global address space eliminating the distinction of local-vs-remote data accesses and relief the programming effort.

In the thesis, we focus on one example of novel fine-grained architecture called the UpDown. 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 with 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.

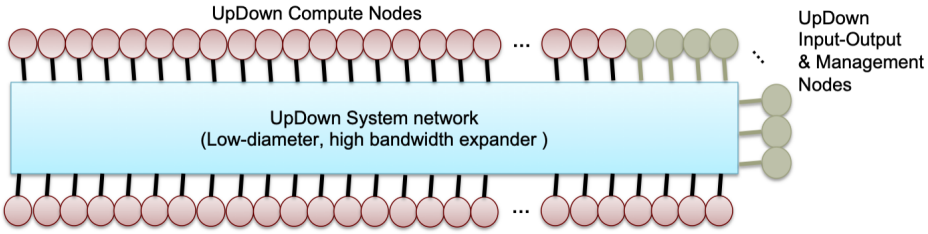


Figure 2.2: The UpDown System connected by the high-performance system network.

The UpDown accelerator is programmed using a RISC-V-like ISA with customized instructions for hardware-assisted event-driven programming. These instructions interact with

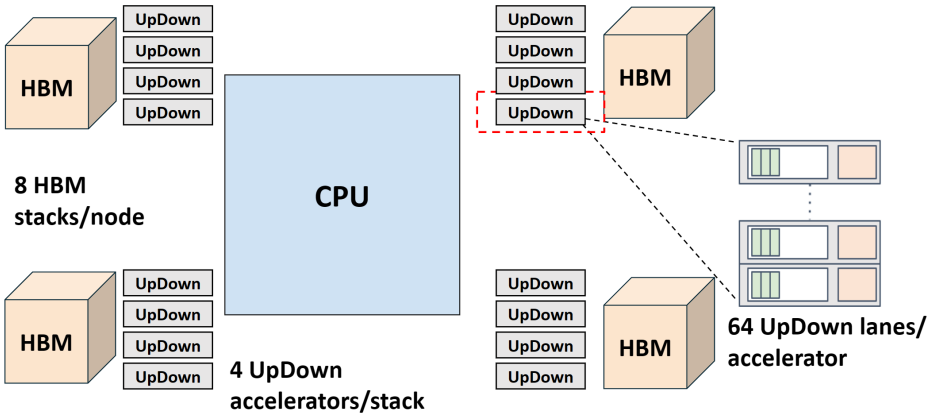


Figure 2.3: each node has 1 CPU, 32 UpDown accelerators with uniform access to 8 HBM stacks.

the hardware structures customized for fast messaging described above and speed up event creation and event handling on UpDown. Hence, an UpDown lane can create an event and send in as minimal as 3 instructions. On the receiver side, the message operands are directly mapped onto the register namespace 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 namespace 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. A full-scale UpDown system has 16k nodes, 32 UpDown accelerators per node, 64 lanes per accelerator, and up to 128 thread context per lane, capable of delivering 4,194,304,000 fold hardware parallelism.

Despite the promising performance potential, UpDown's extremely fine-grained hardware

parallelism poses a significant challenge in writing software programs capable of utilizing its potential and delivering good performance. Most importantly, one needs 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*? Addressing these challenges is the subject of our work.

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 the 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 computa-*

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

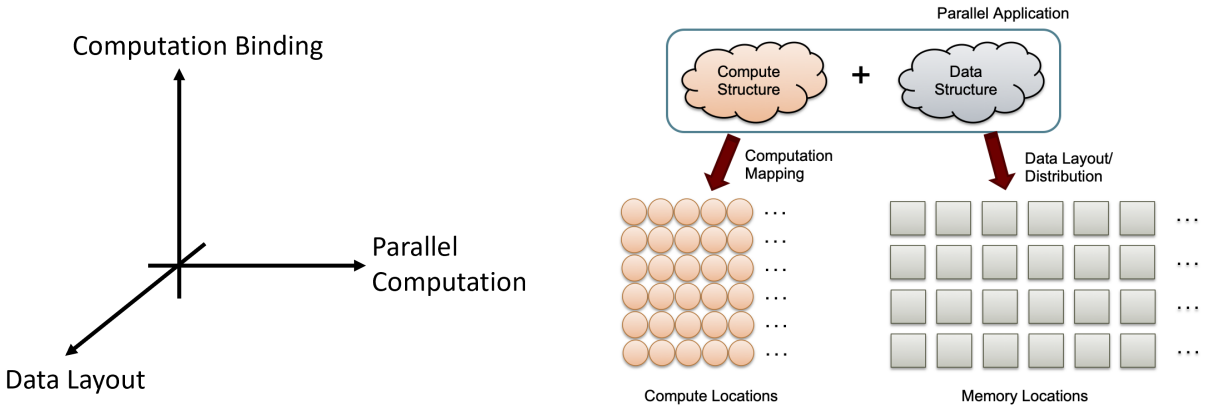


Figure 3.1: Parallel applications manage three dimensions to achieve performance (left). Computation and data need to be mapped to distributed capabilities in a balanced way for good performance.

3.2 KVMSR Programming Model

To answer the two questions proposed in Section 2, 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.

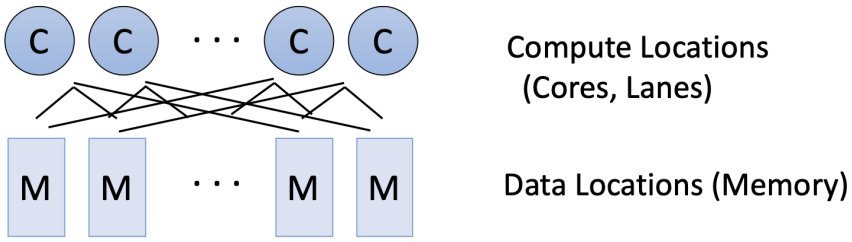


Figure 3.2: A parallel machine fundamentally has compute and data locations connected by an interconnect.

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.

```
thread KVMSR {
    LaneRange lanes;

    event map_shuffle_reduce(KVSet input_set, KVSet output_set,
            LaneRange lanes_op) {
        KVSet inter_set;
        lanes = lanes_op;

        for(KVPair kv : input_set)
            send(kv_map, kv.key, kv.values);
        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.

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

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

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.

```
thread UserKVSet: 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.

```
thread UserKVMSR: KVMSR {
    LaneRange lanes;
    ....
    event get_map_loc(Key key) {
        // Pick a lane from lanes based on key
        LaneID id = ...;
        send_reply(id);
    }

    event get_reduce_loc(Key key) {
        // Pick a lane from lanes based on 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 unpre-

dictable 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 three 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 Express Parallel 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 pseudo code

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

    event get_next(Key key) {
        Key next_key = key.y_idx < N - 1 ?
            Key{key.x_idx, key.y_idx + 1} : Key{key.x_idx + 1, 0};
```

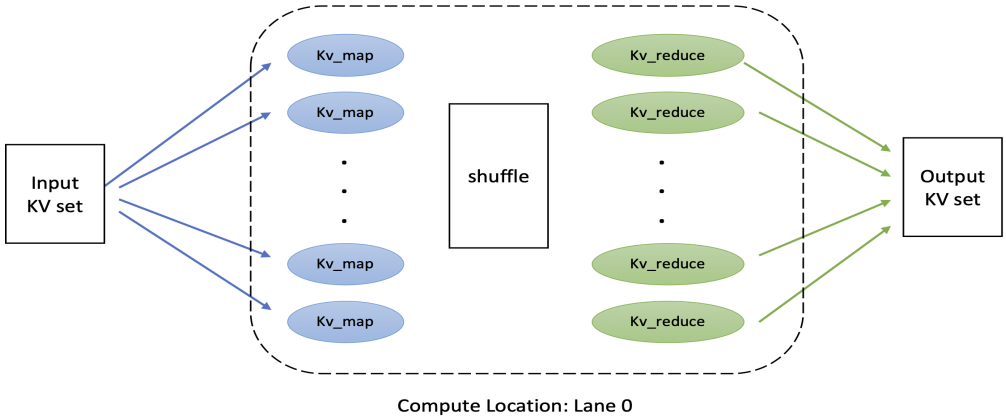


Figure 4.1: Computation binding for the baseline convolution filter program: placing all the tasks on a single lane.

```
        send_reply(next_key);
    }

    event get_values(Key key) {
        send_reply(input_image[key.x_idx][key.y_idx]);
    }
}

thread ConvFilterKVMSR: KVMSR {
    double output_image[M][N];

    event kv_map(Key key, double value) {
        double[3][3] conv_kernel = load_filter();
        for (int i = -1; i < 2; i++) {
            for (int j = -1; j < 2; j++) {
                if (key.x_idx + i < 0 || key.x_idx + i >= M ||
                    key.y_idx + j < 0 || key.y_idx + j >= N)
                    continue;
```

```
                Key inter_key = Key{key.x_idx + i, key.y_idx + j};
                kv_emit(inter_key, conv_kernel[i*-1][j*-1] * value);
            }
        }
        send_reply();
    }

    event kv_reduce(Key key, double value) {
        output_image[key.x_idx][key.y_idx] += value;
        send_reply();
    }
}
```

4.1.2 Instruct 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.

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

```
thread ConvFilterKVMSR: KVMSR {
    LaneRange lanes;
    ...
    LaneID get_map_loc(Key key) {
        int idx = key.x_idx * input_img.dim[1] + key.y_idx;
        return lanes.base_lid + idx % lanes.num_lanes;
```

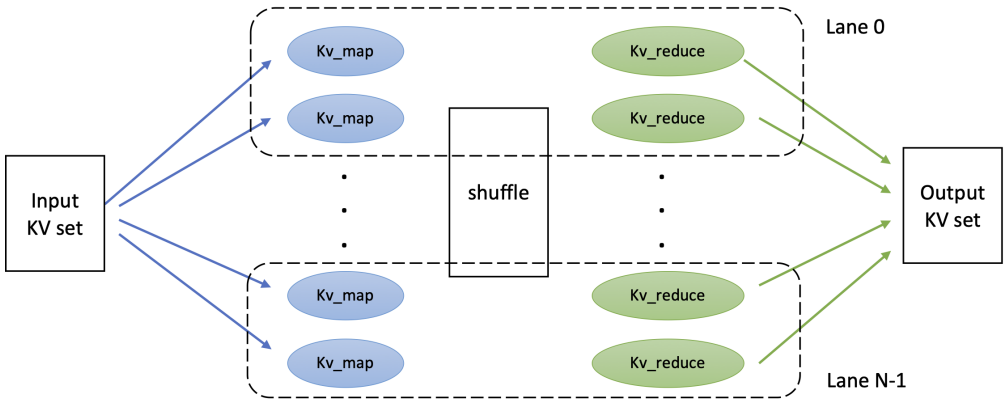


Figure 4.2: The statically distributed Convolution Filter program spreads computation over N lanes.

```
    }
    LaneID get_reduce_loc(Key key) {
        int idx = key.x_idx * input_img.dim[1] + key.y_idx;
        return lanes.base_lid + idx % lanes.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 [39] as shown in the KVMSR pseudo-code in Listing 4.3.

Listing 4.3: Baseline PageRank Program pseudo code

```
struct Vertex{
    int degree;
    double value;
    int neighbors[];
}

thread VlistKVSet: KVSet{
    typedef int Key;
    typedef Vertex Value;

    Vertex* vertices;

    event get_next(Key key) {
        send_reply(key + 1);
    }

    event get_values(Key key) {
        send_reply(vertices + key * sizeof(Vertex));
    }
}

thread PageRankKVMSR: KVMSR {
    double alpha;

    event kv_map(Key key,  Vertex vertex) {
        double out_pr_value = (vertex.value * alpha) / vertex.degree;
        for (int i = 0; i < vertex.degree; i++)
```

```
            kv_emit(vertex.neighbors[i], out_pr_value);
        send_reply();
    }

    event kv_reduce(Key key, double value) {
        Vertex vertex = output_kvset.get(key);
        vertex.value = one_pass_avg(vertex.value, value);
        send_reply();
    }
}
```

4.2.1 Tuning Computation Binding: Static VS. Dynamic

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.

```
thread PageRankKVMSR: KVMSR {
    LaneRange lanes;
    ....
    LaneID get_map_loc(Key key) {
        LaneID id = lanes.base_lid + key % lanes.num_lanes;
        send_reply(id);
    }

    LaneID get_reduce_loc(Key key) {
        LaneID id = lanes.base_lid + key % lanes.num_lanes;
```

```
        send_reply(id);
    }
}
```

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

```
thread PageRankKVMSR: KVMSR {
    LaneRange lanes;
    ....
    LaneID get_map_loc(Key key) {
        LaneID id = lanes.get_location(input_set.get_values(key));
        send_reply(id);
    }

    LaneID get_reduce_loc(Key key) {
        LaneID id = lanes.get_location(input_set.get_values(key));
        send_reply(id);
    }
}
```

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

```
thread PageRankKVMSR: KVMSR {
    ....
    LaneID get_map_loc(Key key) {
        send_reply(get_less_busy_lane());
    }

    LaneID get_reduce_loc(Key key) {
        send_reply(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 BFS

We then present a more complicated example, BFS, which involves changing the `KeyValueSet` data structure and layout in DRAM as part of the optimization. At a high-level, each iteration of the BFS KVMSR program reads from the frontier containing the active vertices and conditionally update their neighbors if they are not visited. The program keeps iterating until the frontier becomes empty. Listing 4.7 shows the KVMSR program pseudo code for BFS. Similar to the PageRank example, map tasks in BFS operates on the vertex-level and involves traversal of the neighbor list except that the vertex comes from the frontier and updates are optional based on incoming vertex depth.

Listing 4.7: Baseline BFS program pseudo code.

```
struct Vertex{
    int vid; int degree;
    int neighbors[];
}

thread bfsKVMSR: KVMSR {
    GraphAbstraction graph;

    event kv_map(Key key, int value) {
        Vertex v = graph.get_vertex(key);
        int out_depth = value + 1;
        for (int i = 0; i < v.degree; i++)
            kv_emit(v.neighbors[i], out_depth);
        send_reply();
    }

    event kv_reduce(Key key, int value) {
        Vertex u = graph.get_vertex(key);
        if (u. depth > value) {
            u.depth = value;
            kv_emit(u, value);
        send_reply();
    }
}
```

4.3.1 Enable Rich-structured Key-Value Set

So far, the KVMSR examples we gave all take plain arrays as input and/output key-value sets. Listing **??** shows the pseudo code to take an one dimensional array as input key-value set. Implementing the BFS frontiers using fixed sized arrays is possible since the size is bounded by the number of vertices but as the graph data scales to gigabytes and terabytes, linear look up on a huge array with millions of entries to merge redundant insertions becomes extremely expensive. Furthermore, the KVMSR BFS program is designed to run on massively-paralleled machine where concurrent insertion/update is critical to scalable performance.

Listing 4.8: Array-based frontier BFS program pseudo code.

```
thread ArrayKVSet: KVSet{
    typedef int Key;
    Values* values;

    event get_next(Key key) {
        send_reply(key + 1);
    }

    event get_values(Key key) {
        Values* value_ptr = values + key * sizeof(Values);
        send_reply(value_ptr);
    }
}
```

Fortunately, the issue can be solved by a hash table distributed across the parallel machine allowing concurrent atomic updates and fast table look ups with constant overheads. KVMSR's key-value set abstraction enable adopting the rich-structured data structure like

hash table fairly simple. In this case, to enable hash table as KVMSR's input key-value set, it simply requires implementing the abstract iterating function `next(Key)` using the hash table's build-in iterator and wrapping the hash table `get()` function inside key-value set's random access function `get_values(Key)`. The pseudo code to enable hash table key-value set are depicted in Listing 4.9.

Listing 4.9: HashTable-based frontier BFS program pseudo code.

```
thread HashTableKVSet: KVSet{
    HashTable<Key, Values> hash_table;

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

    event get_values(Key key) {
        send_reply(hash_table.get(key));
    }
}
```

The BFS KVMSR program gives an example of adopting advanced data strctures in terms of key-value abstraction, and highlights KVMSR's flexible support of data layouts and concurrent data abstractions in a global shared memory machine.

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 a massively-paralleled fine-grained 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 [33, 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 load_neighbors(long degree, long vid, long *neighbors, long dist) {
    // 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

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

`get_reduce_loc()`). The baseline does not specify any computation binding and execute 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 thousand-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 compare the performance of the same UDKVMSR programs except for different choices of computation binding functions. We further scale up the machine 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 the key (i.e., statically divided the keys across the lanes).

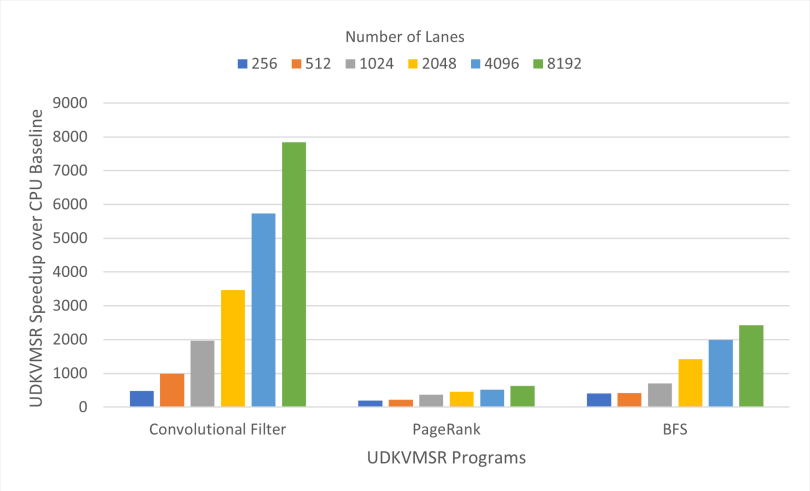


Figure 5.1: Speedup of the UDKVMSR Convolution, PageRank and BFS with static binding based on Keys over the single-thread CPU baseline

Compared to the single-thread C++ baseline program, the UDKVMSR convolution program gains 477x performance speedup on 256 lanes and 7,845x on 8,192 lanes program, achieving good 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 16.8x 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

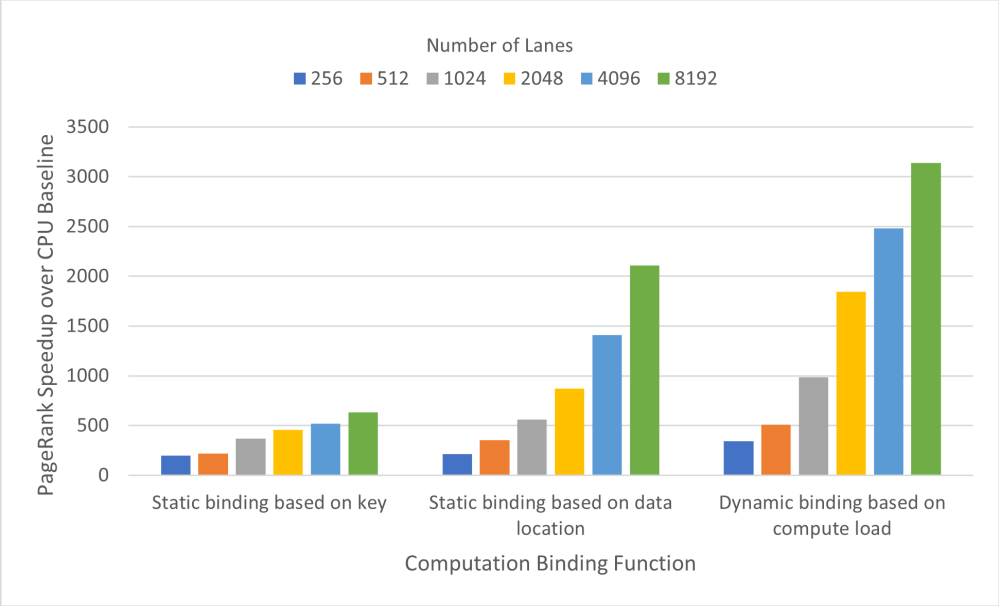


Figure 5.2: Speedup of PageRank with static binding based on Keys (left bars), static binding based on data location (middle bars), and dynamic binding based on compute load (right bars) over the single-thread CPU baseline.

both programs take the RMAT graph with skewed degree distribution as input. 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

We show that the two graph UDKVMSR programs with static computation binding based on key approach scale poorly as the hardware parallelism increases in the previous subsection. The main reasons are:

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

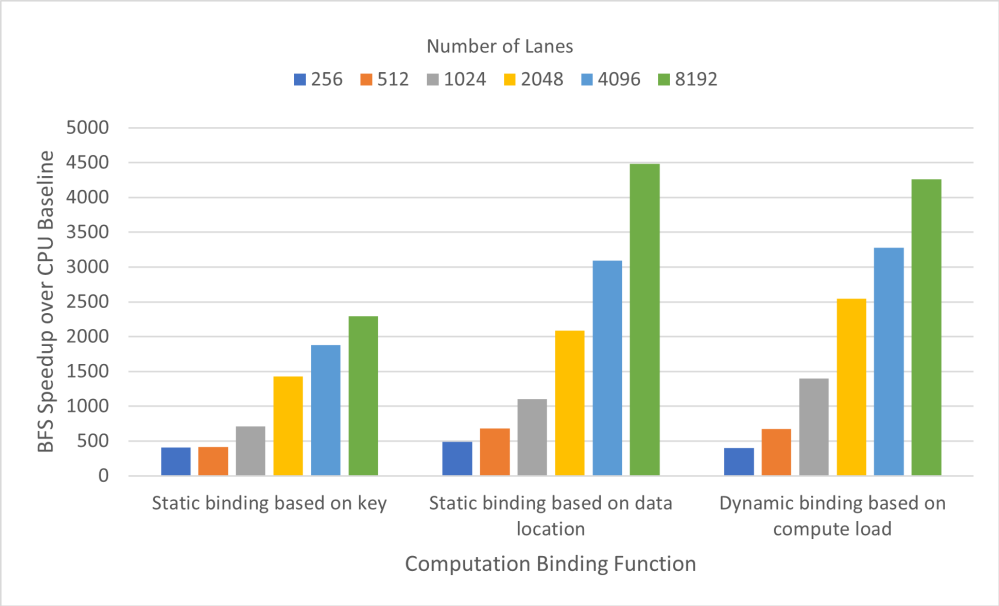


Figure 5.3: Speedup of BFS with static binding based on Keys (left bars), static binding based on data location (middle bars), and dynamic binding based on compute load (right bars) over the single-thread CPU baseline.

each data so DRAM access latency dominates the performance.

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 evenly 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 approach. 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 the 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 dominate 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

BROADER USAGE AND INSIGHT

We have presented in the Chapter 4 three KVMSR programs, including a demonstration of using KVMSR's modular interface to tune various dimensions of a parallel program (computation, data, and computation binding described in Section 3.1) for performance. For the sake of understanding, the examples we pick are relatively easy: most map/reduce function involves a couple of integer/floating point arithmetic instructions for each key-value pair. We want to highlight in this Chapter that KVMSR has supported a much wider range of challenging irregular computations with massive fine-grained parallelism. We would also like to share the lessons we learned from the implementation practise.

6.1 Programmability and Flexibility

KVMSR belongs to the larger project on developing the next-generation high-performance supercomputer targetting sparse and irregular computation. At the time of this writing, the UpDown project has completed 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 and performance tuning in weeks due to KVMSR's simple and modular interface. Table 6.1 summarizes the application computation and their usage of KVMSR.

Table 6.1: Description of programs implemented in the KVMSR programming model and run on the UpDown system.

Program	Key-Value Set	Computation
SpMV	**Input&output**: Sparse matrix representation. Each key has two values: 1) Row start offset, and 2) row end offset	The map function fetches the sparse matrix values indexed by the KVMSR key, then gets the corresponding vector values, computes the row entry in the resulting vector, and store to the output matrix.
GNN	**Input&output**: The vertex store inside a graph abstraction implemented using the hash table. Keys are Vertex IDs and the key-value pair is the `Vertex` in the vertex store.	The map function gets all the neighbors of the vertex (key) and their feature vectors, performs aggregation, and then writes back the resulting vector.
Multihop Ingestion	**Input&output**: Hash table. Keys are the hash table key.	The map function filters entries based on relation. If the relation matches, insert head/tail into another SHT in reduce.

Program	Key-Value Set	Computation
Influence Maximization I: Compute Random Reverse Reachable Set (RRR Set)	**Input**: An array of random seeds. Keys are the seed vertex IDs to start the random BFS. **Output**: Hash table. The keys contain two values: the vertex ID and the RRR set ID.	The map function runs a serialized random BFS on a globally shared graph and inserts the vertices belonging to the resulting BFS tree into a hash table. Note that the graph is stored in a graph abstraction implemented using the hash table and is not part of the input key-value set. Calls to query the graph abstraction are made inside the map function (e.g., get a vertex's degree and neighbor vertices, etc.)
Influence Maximization II: Find the vertex with maximized influence	**Input**: The hash table from phase I. Each key has the vertex ID and the RRR set ID containing the vertex. **Output**: A histogram implemented with an array. The values are the number of occurrences of a vertex in all the RRR sets.	The map function generates a count of 1 for the vertex ID field in the input key. The resulting key-value pair is shuffled to reduce. The reduce function sums all the counts corresponding to the same vertex and writes to the output histogram. After the KVMSR program, traverse the histogram to find the vertex with the maximized counts.

Program	Key-Value Set	Computation
Influence Maximization III: Eliminate RRR sets containing the maximized influencer.	**Input**: The hash table from phase I storing all the RRR sets. **Output**: An array of flags indicating whether a RRR set is active. The key is implicitly the index of the array, corresponding to a RRR set ID.	The map function checks if the vertex id of the input key equals the max influencer's ID. If so, set the corresponding RRR set to not active.
Adjust Graph I: Remove influencers.	**Input**: An array of max influencers found by the Influence Maximization program.	The map function removes the max influencer (input key) from the graph and its neighbors by calling the graph abstraction to set the edge attribute to 0. Note that the graph is stored in the global shared memory and implemented using the graph abstraction.
Extract Subgraph (Vertex)	**Input&Output**: The vertex store of a graph abstraction implemented using the hash table. Keys are Vertex ID and the key-value pair is the `Vertex` in the vertex store.	The map function conditionally inserts the input vertex (input key-value pair) to another graph's vertex store based on the vertex attribute.

Program	Key-Value Set	Computation
Extract Subgraph (Edge)	**Input&Output**: The edge store of a graph abstraction implemented using the hash table. Keys are Edge ID and the key-value pair is the `Edge` in the edge store.	The map function conditionally inserts the input edge (input key-value pair) to another graph's edge store based on whether the source and destination vertex is selected in the **Extract Subgraph (Vertex)** program.
Generating Genomic Contigs	**Input**: Array of file offsets for a text file containing the genome sequence. **Output**: A histogram implemented with hash table. The key is the 32-kmer to be counted and the values are the number of occurrences of a 32-kmer in the text file.	The map function reads the from text file based on the input file offsets (key-value pair) and generates a count of 1 for each 32-kmer in the subsequent read. The counts will be shuffled to the reduce function and sum to the total count for that kmer. Then, the reduce outputs the counts to the output histogram.
Building macro-node graphs	**Input**: Hash table from the **Generating Genomic Contigs** program **Output**: A graph abstraction implemented in hash table. The graph to be constructed containing the macro nodes.	The map function gets an entry from the input hash table, generate two nodes and an arbitrary number of edges and insert them into the output graph.

Program	Key-Value Set	Computation
Splitting Vertex in a skewed graph	**Input&Output**: *version 1* - array-based vertex list *version 2* - graph abstraction implemented in hash table	The map function gets a vertex from input key-value set, split the vertex into sub-vertices if the original vertex's degree is greater than a split threshold, and randomly assign the neighbors to split vertices. Then, the reduce function inserts the new-generated vertices if split otherwise the original vertex with its neighbors into the output output key-value set.

6.2 Insight and Lessons Learnt

When implementing the programs listed in the table 6.1, we found the following properties of KVMSR greatly aid the programming effort:

- **Express computation in a MapReduce-like interface**: For most programs, it is natural to break the parallel computation into map phase which computes on independent data and the reduce phase where values are exchanged and merged. Having the MapReduce interface for expressing the computation allows programmers to finish implementation in days. Also, many programs already have the same or similar variations implemented using the cloud MapReduce frameworks, so the transition to a program on the new fine-grained architecture is fairly smooth. One example is the **Generating Genomic Contigs** program, which is a variation of the well-known word-counting MapReduce program. The computation in the **Influence Maximization II**: Find the vertex with maximized influence program is also similar.

- **Modular Interface design**: We found that a number of programs above either share similar input/output data or has the two versions of the same program with different

input/output. KVMSR's modular interface greatly eases the effort to build a program or variants of the old program via only changing one part of the program, i.e., data layout/structure or computation, unlike traditional approaches requiring rewriting all the closely coupled parts of the program.

- **Global memory**: It is highlighted in the table that a few programs interact with an additional globally shared data abstraction in the map and/or reduce function which is not part of the input/output key-value set of the KVMSR program, or instance, the **Influence Maximization I**: Compute Random Reverse Reachable Set (RRR Set) program and the **Adjust graph** program. This is possible because KVMSR operates on a global shared memory and can directly address anywhere in the system provided with a valid global address. Without it, the program would need to be further broken into more pieces with significantly more intermediate states to keep track.

- **Flexible parallelism management and control**: The above parallel programs often have million-fold parallelism in software and are running on a machine with thousands of independent MIMD processing units and magnitudes of more threads. KVMSR's modular interface greatly eases the challenge for programmers to tune the parallelism and manage the computation binding for performance.

CHAPTER 7

RELATED WORK AND DISCUSSION

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

7.1 Functional Language and Cloud Map-Reduce

Functional languages allowed functions to be applied to sets/arrays (map) and combine the results (reduce) [26, 25, 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.

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

Message passing interface is a popular model for scalable parallelism (and high-performance computing – HPC). 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 [28]. If work is dynamically generated and tied to such structures, e.g., irregular work and parallelism, programming is even more challenging [22]. If the data or computation is irregular, this produces complex programming and communication (see high-performance implementations of irregular and graph applications [28, 27]).

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, 22]. PGAS programs provide the convenience of global naming, easing the pro-

gramming 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.

7.2.1 MapRedudce in PGAS

Some previous works in the early 2010s tried to implement the MapReduce

7.3 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.

7.4 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, 36]. 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 [23, 34]. 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 [21, 12]. KVSMSR targets general irregular algorithms and data, a much larger class of applications.

7.5 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 8

SUMMARY AND FUTURE WORK

8.1 Summary and Future Work

We have presented a key-based MapReduce-like programming model, KVMSR, for optimizing the execution of irregular parallel programs on large-scale parallel systems. The model enables the expression and management of fine-grained parallelism with keys, global naming of data structures and computation locations, and customized control of compute location binding orthogonal to the expression of data layout and computation itself. We have demonstrated the expressiveness, modularity, and flexibility of the KVMSR programming interface and presented the promising initial performance it can achieve on three irregular parallel programs.

Several research problems still remain to be explored surrounding the KVMSR programming model. Some of them are discussed below.

1. Evaluation of the KVMSR on a larger machine setup, e.g., millions of parallel execution units. Also, explore KVMSR's opportunity on other state-of-art or theoretical parallel machines.

2. A more rigorous experiments on a broader range of regular and irregular applications list in Chapter 6 and on larger and/or more skewed datasets. This is aim to verify the scalable performance achieved by KVMSR shown in this thesis can be generalize to a variety of applications.

3. Exploration of other dynamic load-balancing approaches to choose computation location using the application and/or runtime information and potentially with UpDown's novel machine mechanisms on a broader applications.

4. Ongoing work on a dynamic global fine-grained load-balancer extension for KVMSR. The is an extension of the static and locally-dynamic load-balancing mechanism the illustrate in this thesis.

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] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. "Filtering: a method for solving graph problems in MapReduce". In: *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 85–94. ISBN: 9781450307437. DOI: 10.1145/1989493.1989505. URL: https://doi-org.proxy.uchicago.edu/10.1145/1989493.1989505.

[20] 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.

[21] 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.

[22] 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.

[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/10.1145/1807167.1807184.

[24] 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.

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

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

[27] 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.

[28] 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.

[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/s00361445034 2480. 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] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. "Scalable big graph processing in MapReduce". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 827–838. ISBN: 9781450323765. DOI: 10.1145/2588555.2593661. URL: https://doi-org.proxy.uchicago.edu/10.1145/2588555.2593661.

[33] 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.

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

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

[36] 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.

[37] 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).

[38] 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.

[39] 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.