Efficiently Exploiting Irregular Parallelism Using Keys at Scale

Yuqing Wang[1], Andronicus Rajasukumar[1], Tianshuo Su[1], Marziyeh Nourian[1], Jose M Monsalve Diaz[2], Ahsan Pervaiz[1], Jerry Ding[1], Charles Colley[3], Wenyi Wang[1], Yanjing Li[1], David F. Gleich[3], Hank Hoffmann[1], and Andrew A. Chien[1,2]

[1] Department of Computer Science, University of Chicago, Chicago IL 60637, USA
{yqwang,andronicus,tssu,marziyehnourian,ahsanp,jrding,wenyiw, yanjingl,hankhoffmann,aachien}@uchicago.edu

[2] Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439, USA jmonsalvediaz@anl.gov

[3] Department of Computer Science, University of Purdue, IN 47907, USA
{ccolley,dgleich}@purdue.edu

Abstract. Motivated by the challenges of programming irregular applications for machines with million-fold parallelism, we present a key-based programming model, called key-value map-shuffle-reduce (KVMSR), that enables programmers to optimize fine-grained parallel programs. KVMSR expresses parallelism on a global address space and features modular interfaces to flexibly bind computation to available compute resources.
We define the KVMSR model and illustrate it with three programs, convolution filter, PageRank and BFS, to show its ability to separate computation expression from binding to computation location for high performance. On a 2,048-way parallel compute system, KVMSR modular computation location control achieves up to 1,732x performance with static approaches and an increase of 372x to 1,127x speedup with dynamic approaches for computation location binding.

Keywords: graph computing · parallel computing · fine-grained parallelism · scalable computing · high-performance computing · map-reduce

1 Introduction

Massive graph analytics applications of billions of vertices, arising from social network analysis and supply chain resilience, are increasingly important, and the state-of-the-art demands on the online responses for queries on such large graphs. Such applications will require systems with millions of compute elements and petabytes of memory. Programmers of such machines must generate sufficient parallelism to utilize compute resources, effectively bind the parallelism to compute elements and manage parallelism efficiently. This is challenging because real-world graphs give rise to irregular parallelism and poor data locality, producing load imbalance, costly communications and data movement, and poor overall performance.

Main-stream scalable, parallel programming solutions do not support irregular applications well (eg. real-world graph analytics). For example, message-passing models (e.g., MPI + domain decomposition) provide scant support for global data structures and exploiting irregular parallelism. To achieve high performance, programmers must assemble the data required for each piece of parallel computation and align it to a static set of workers (ranks) [4].

Even partitioned global address space models (PGAS) that provide a global address space that aids in building the distributed global data structure, fail to serve these graph analytic applications well [19]. In PGAS programs such as UPC++, good performance depends on careful decomposition and single program multiple data (SPMD) orchestration of data movement across multi-dimensional arrays. Dealing with graphs is possible, but difficult because of their irregularity in data and parallelism [2, 18].

A promising direction is MapReduce. Traditional functional programming languages have used map and reduce functions to express fine-grained parallelism, but are limited by the scaling of shared memory machines to ≈ 256, and thus cannot support these future machines. Cloud map-reduce systems add keys to organize the computation and are more scalable, but cannot exploit fine-grained parallelism [15, 20, 6].

We propose the key-value map-shuffle-reduce (KVMSR) framework to support irregular data and computation parallelism in future large-scale parallel systems. These systems will use novel building blocks for fine-grained parallelism and scale to 30 million parallel computation elements [21]. To optimize irregular programs, KVMSR combines rich-structured keys and global addressing of data and adds novel simple interfaces for programmers to directly control the binding of map and reduce tasks to compute resources. Hence, programmers can exploit the expressiveness of keys to flexibly manage parallelism and then optimize performance, in a modular fashion by controlling data locality and load balance.

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

– Design of KVMSR for expressing and managing fine-grained parallelism using keys. KVMSR expresses the binding of computation to compute resources independently from program computation and data structure.
– Examples of how the keys in KVMSR can be used to efficiently control computation binding, both statically and dynamically (using application, system, and data information), to achieve load balance and high performance on irregular graph applications
– Evaluation that shows on a 2,048-way parallel compute system, KVMSR modular computation location control achieves up to 1,732x performance with static approaches and an increase of 372x to 1,127x speedup with dynamic approaches for computation location binding.

The remainder of the paper is organized as follows. We define the KVMSR model in Section 2, followed by two program examples described in Section 3.

An implementation of the KVMSR model is evaluated in Section 4, to show its flexibility and performance benefits. We discuss related work in Section 5, and summarize and point out directions for future work in Section 6.

2 KVMSR Programming Model

Successful parallel programming requires an application to control three dimensions (see Figure 1), coordinating them to achieve good parallel scalability and performance. Each parallel application must deal with several challenges that arise from parallel/distributed capabilities as shown in Figure 1, 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). 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 applications, especially the irregular ones.

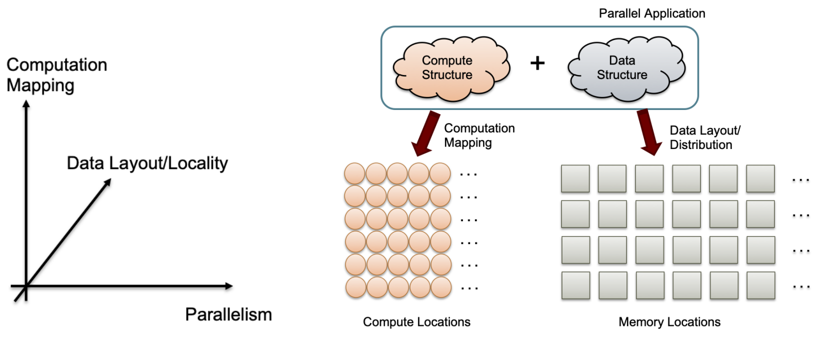

Fig. 1. Parallel applications manage three dimensions to achieve performance (left). Computation and data must be mapped to distributed capabilities.

The key-value map-shuffle-reduce (KVMSR) model supports a flexible expression of parallel computation and data structures, independently from the choice of computation binding and data distribution.

KVMSR employs *keys* as the critical abstraction for programmers to express parallel computation. Parallelism is, therefore, equal to the number of keys in the input, and as fine-grained as the corresponding map task. The reduce tasks are similar except that they are based on the intermediate keys generated by the map tasks. KVMSR's major innovation is to use the keys as the basis for programmers to bind computation to parallel computer resources. Most importantly, such binding is expressed independently from the data layout and program computation, resulting in a clean and modular interface. To support irregular applications, KVMSR provides a global address space so that a data item can be named uniformly from anywhere in the machine. KVMSR also exposes machine primitives that locate computation based on data location and dynamic computation load. 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 (control binding of `kv_map()` and `kv_reduce()` tasks to computation resources)
3. Global address space (uniform naming and global data layout)
4. Exposing machine primitives (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. For example, different dimensions of a program can be successively tuned to achieve high performance on the target system: First exploring computation binding to exploit parallelism, then colocating for data locality, and finally, load balancing based on dynamic information.

2.1 Machine Model

A parallel computer fundamentally includes two types of elements – compute and memory – tied together by an interconnect as illustrated in Figure 2. We assume the hardware provides a global address space, and the ability to send messages and move data across the interconnect with low overhead. High performance is achieved by sufficient parallelism and good load balance to utilize all of the compute elements efficiently.

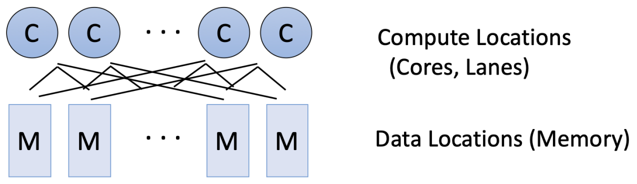

Fig. 2. A parallel machine fundamentally has compute and data locations connected by an interconnect.

One critical capability for irregular applications is to place computations based on the dynamic evolution of the program. For example, adapting to the computation load or the location of the data to be operated on. In all cases, static and dynamic, we assume that compute and memory elements locations are denoted {0:nr_memories}, and there may be finer resolution of compute locations, denoted lanes {0:nr_lanes}.

2.2 Expressing Computation

KVMSR computes on sets of key-value pairs. The values can include pointers to other data (or even synchronizing data abstractions) in the global shared address space. For example, independent map tasks could access a shared hash table supporting atomic operations, or an MPMC queue. As a result, KVMSR's map and reduce functions are extremely capable.

KVMSR's map phase generates an intermediate key-value set which is then shuffled to reduce. The reduce phase executes in parallel across the intermediate keys and produces an output key-value set. The function is illustrated in pseudo-code in Listing 1.1.

Listing 1.1. Pseudo-code for KVMSR. Execute map functions in parallel, generates an intermediate key-value set, shuffles, and computes reduce functions in parallel to produce an output key-value set.

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

2.3 Map and Reduce Functions

KVMSR programs define the computation tasks to be executed using the `kv_map()` and `kv_reduce()` functions. The interface is illustrated in Listing 1.2.

Listing 1.2. Function `kv_map()` and `kv_reduce()` interface.

```
void kv_map(Key key, Types values) {
    ... map code ...
    kv_emit(inter_key, inter_values);
    return; }

void kv_reduce(Key inter_key, Types inter_values) {
    ... reduce code ...
    return; }
```

As described above in Listing 1.1, the `kv_map()` functions are called in parallel for each key in the input 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 key-value pair of the intermediate key-value set 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 computation.

2.4 Global Data Structures

KVMSR assumes execution on a scalable parallel MIMD computer that provides global naming for data structures in memory. With unified names, calls to operate on arbitrary shared data abstractions can be made within the `kv_map()` or `kv_reduce()` function. For example, they can include reads and updates for a shared hash table, where atomic operations are provided by the hash table abstraction. Another common use is tree or queueing structures, where the structure implements both exclusion and ordering to support correctness.

This makes KVMSR much more powerful than cloud MapReduce systems [6, 7] where map and reduce computations cannot share global data. In Section 3.1, we give an example to demonstrate the benefits of operating on global memory.

Computation Location Naming KVMSR abstracts the physical compute elements into a list of computation locations, each of which is assigned an ID. Take a machine consisting of 256 lanes as an example, each lane is assigned an ID in the range {0:255}, and each computation task (i.e., either a `kv_map()` or `kv_reduce()`) will be bound to one of the lanes. We will further describe the binding in the section 2.5.

The naming scheme does not assume specific topology and/or hierarchy, but programmers can embed machine-specific information into the location name to reflect the nearness of compute units or the sharing of physical resources. In the rest of the paper, we use lanes as the basic units of compute, i.e., each lane is a compute location assigned with an ID of type `LaneID`.

2.5 Parallelism Management

Applications exploit the available parallel compute resources by binding tasks to computation locations. Whether one can efficiently do so is critical to the program's performance. To help write and optimize parallel programs, KVMSR provides two customizable functions for programmers to control the binding of computation to resources, i.e., compute locations.

Customizing Computation Location Binding In KVMSR, each key corresponds to a parallel computation task, and the function `get_map_loc()` and `get_reduce_loc()` use the key to determine a location on which the task will execute. The interface is illustrated in Listing 1.3.

Listing 1.3. Function `get_map_loc()` and `get_reduce_loc()` interface. Bind keys to compute locations.

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

Exploiting Parallel Compute Resource The computation of a compute location name can be done statically or decided dynamically as below.

– **Static** Simple hashing or static distribution techniques can be used to spread unpredictable task sizes and number of task computations across the machine. The binding can also be determined statically based on the data location, given it does not change during the program execution.
– **Dynamic** Locations can also be dynamically determined based on machine compute load.Applications can use machine-specific features, such as `get_less_busy_lane()`, to dynamically decide the binding.

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

3 Program Examples

We use two examples to show KVMSR's programming interface and highlight its efficiency in expressing irregular parallel programs. We assume a highly paralleled machine with thousands of lanes, each of which is assigned an ID.

3.1 Convolution Filter

We start with a simple example, the convolution filter, to illustrate the interface. The pixel data is stored in a global two-dimensional array. As shown in Listing 1.4, 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. Without further specification, the program places all computations on one lane, i.e., Lane 0.

Listing 1.4. Baseline KVMSR convolution filter program code

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

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

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

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

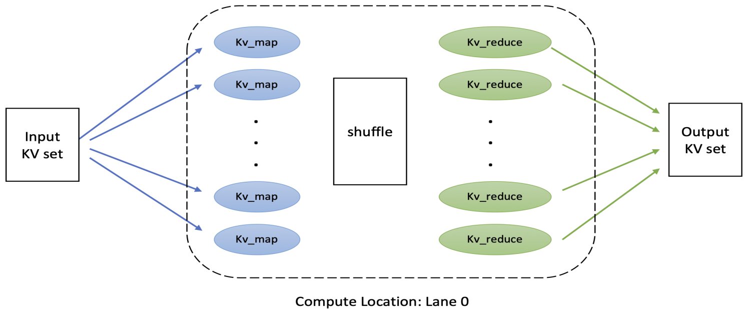

Fig. 3. Computation binding for the baseline convolution filter program: placing all the tasks on a single lane.

Serializing the computation on a single lane underutilizes the computation resources, producing poor good performance. So in Listing 1.5, we modify the `get_map_loc()` and `get_reduce_loc()` functions to distribute the 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()`, remains the same, as in Figure 4.

Listing 1.5. Parallel convolution filter program code with static computation location binding based on the key.

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

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

3.2 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 [23] as shown in the KVMSR pseudo-code (see Listing 1.6). If the `get_map_loc()` and `get_reduce_loc()` are omitted, by default, execution uses a single lane (sequential execution).

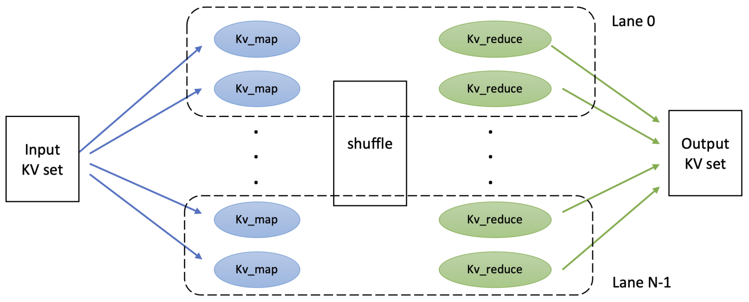

Fig. 4. The statically distributed Convolution Filter program spreads computation over N lanes.

Listing 1.6. Baseline PageRank Program Code

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

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

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

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 1.7 (the rest of the code remains the same and is omitted from the pseudo-code).

Listing 1.7. PageRank program with static computation binding based on key

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

Alternatively, one can also spread the computation based on the data locations. For example, in Listing 1.8, we use the function `get_location(data)` to find the location of data and statically bind computation to where it is located.

Listing 1.8. PageRank with static computation binding based on data location

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

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

To solve the issue raised by irregular data, we present another variant of PageRank which uses `get_less_busy_lane()` to dynamically identify lanes with less load and bind computation accordingly to spread the load. The resulting program is shown in Listing 1.9.

Listing 1.9. PageRank program with dynamic computation binding based on the computate load

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

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

4 Evaluation

We evaluate the KVMSR programming model on a simulated UChicago UpDown machine, a highly parallel machine designed for 30 million of multithreaded lanes, and evaluate the model performance using 2,048 lanes [21].

4.1 Performance Model

An UpDown machine has 2,048 lanes, i.e., compute locations, organized in clusters of 64 (one UpDown accelerator), 4 of these accelerators are associated with a memory stack, and there are 8 HBM2e stacks in a machine [21]. 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 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 2,048 lanes. 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.

4.2 Implementation and Experiments

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 follows Listing 1.4 and 1.6 in Section 3; 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 1.5, 1.8, and 1.9. We also implemented and evaluated synchronous BFS implemented with UDKVMSR. The program structure is similar to PageRank but with input and output from frontiers implemented using the parallel hash table and different computations on the data. The computation-to-compute-location binding functions for BFS are the same as PageRank's.

To show the expressiveness of KVMSR, we count the lines of code for describing program computation/function and the binding of computation to lanes. The results for each program and variations are presented in Table 1. To evaluate performance, we execute the programs on the UpDown simulator implemented using GEM5, reporting the runtime [21, 3]. The KVMSR programs exhibit fine-grained parallelism, indicated by the number of parallel tasks and the mean instructions per task. Pagerank is not only fine-grained but extremely irregular in its 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.

Table 1. 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 16, 2^{16} vertices)	18MB	47,895	432B	116	17,383

4.3 Results

Programmability and Model Expressiveness KVMSR interface allows computation location binding to be expressed separately from the program's parallel computation(function).

Table 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	84	0	2	2	N/A
PageRank	58	0	2	2	6
BFS	185	0	2	2	6

In Table 2, we present line counts for program function code (`kv_map()` and `kv_reduce()`), and computation binding code (`get_map_loc()` and `get_reduce_loc()`), several versions. The baseline does not specify any computation binding, 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 1.9) adds 6 lines using the machine's special feature `get_less_busy_lane()`. The result in Table 2 highlights KVMSR's modular interface, allowing orthogonal definition of program function and computation binding, i.e., only relevant functions are modified leaving the rest of the program untouched.

Benefits of Computation Location Control To evaluate the performance benefits of the KVMSR model, we run the parallel programs above on the UpDown machine and measure their performance on 1-2,048 lanes. The baseline runs on 1 lane and parallel versions distribute the computation across 256-2,048 lanes.

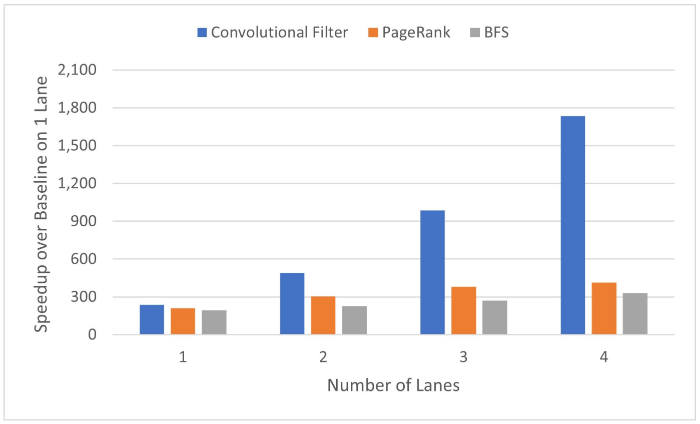

Fig. 5. Speedup of parallel versions of Convolution, PageRank and BFS with static binding based on Keys (speedup vs serialized baseline)

The baseline serialized programs fail to utilize the parallel resources, producing poor performance. Spreading computation across parallel lanes (see Figure 5) produces significant speedups for both applications. Ranging from 213x with 256 lanes to 825x with 2,048 lanes. Convolution scales well, gaining 7.2x for an 8-fold hardware increase from 256 to 2,048 lanes. PageRank and BFS scale less well, gaining only 1.97x and 1.71x respectively, due to irregularity in parallelism from the skewed graphs.

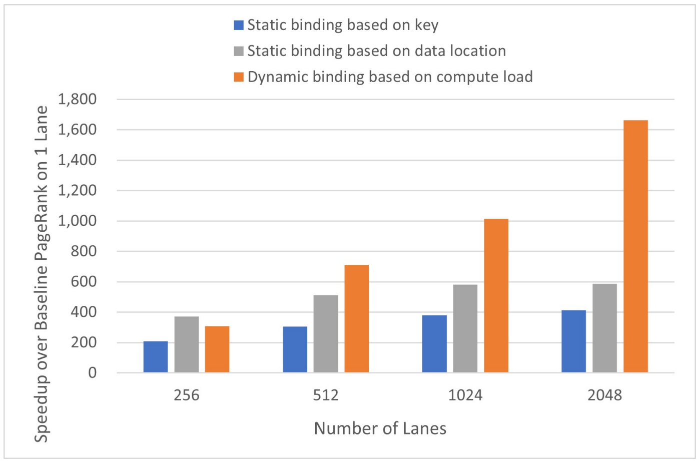

Fig. 6. Speedup of PageRank with static binding based on Keys, based on data location, and dynamic binding based on compute load over the serialized baseline (speedup vs. serialized baseline).

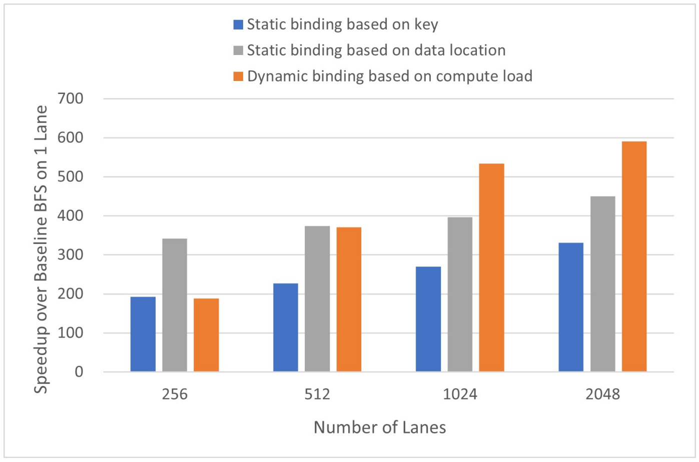

Fig. 7. Speedup of BFS with static binding based on Keys, based on data location, and dynamic binding based on compute load over the serialized baseline (speedup vs. serialized baseline).

Static computation binding based on keys is insufficient to distribute PageRank and BFS's computation efficiently across the machine, due to their irregularity in task size (see Table 1). Both program's task size is determined by the vertex's degree; highly skewed in an RMAT graph. Therefore, we explored other binding functions and showed their performance scaling in Figure 6 and 7.

The first alternative is binding computation to locations close to the data. With improved data locality, PageRank and BFS achieve an average of 1.6x performance improvement compared to the static binding based on keys. However, the performance scaling from 256 lanes to 2,048 lanes barely changed. The imbalanced load across the lanes significantly limits the performance scaling as the hardware parallelism increases.

We've shown that static binding functions lack the run-time information to balance load effectively. In our dynamic binding approach, tasks are distributed based on the computation load of lanes, placing computation tasks dynamically on less busy lanes. This dramatically improves PageRank scaling: about 4 times the speedup for 2,048 lanes from 413x to 1,663x. The improvement on BFS is less, from 331x to 590x, since the BFS's parallelism (i.e., frontier size) is spread across the iterations. Nonetheless, the performance is still better than the static approaches.

In summary, KVMSR's flexible and modular interface allows this dramatic change of computation location binding with a few lines of code.

5 Discussion and Related Work

5.1 Functional Language and Cloud Map-reduce

Functional languages allowed functions to be applied to sets/arrays (map) and combine the results (reduce) [16, 15, 5]. Originated for expressive power, 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. Our studies are for 2,048 compute locations, and a full UpDown design has over 30M compute locations.

Cloud companies built a different map-reduce, designed for scale-out, to internet-scale computations [6, 7]. 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.

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.

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

A popular model for scalable parallelism (and high-performance computing – HPC), is message passing. Typically, the single-program multiple data (SPMD) divides data across separate processes with private address space [10]. 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 [18]. If work is dynamically generated and tied to such structures, e.g., irregular work and parallelism, programming is even more challenging [13]. If the data or computation is irregular, this produces complex programming and communication (see high-performance implementations of irregular and graph applications [18, 17]).

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 [19, 2, 13]. PGAS programs provide the convenience of global naming, easing the programming of complex data structures and irregular parallelism. However, this convenience does not alter the underlying performance challenge, as to achieve speedup work must be aligned and balanced across the address spaces. This is because ultimately the computation is done by cores which can only access data in a single private address space.

5.3 Linda & Tuple Space

Linda is another well-established model in the parallel programming world, focusing on coordinating communication between processes [8]. 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.

5.4 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 [11, 22]. 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 [14, 1]. 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 benefitting from the increased memory to compute larger problems [12, 9]. KVSMSR targets general irregular algorithms and data, a much larger class of applications.

5.5 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.

6 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 with `get_map_loc()` and `get_reduce_loc()` functions. We have demonstrated the expressive power and flexibility of the

KVMSR programming interface and presented the promising initial performance it can achieve on two irregular parallel programs.

Directions for future work include evaluating KVMSR at larger machines (millions of lanes) and application scales (millions of vertices). Experiments with a broader range of applications would also be insightful. A particularly interesting direction is to explore UpDown's novel machine mechanisms to dynamically choose computation location using the application and runtime information.

Acknowledgements

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0082. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

References

1. Scaling apache giraph to a trillion edges (2013), https://engineering.fb.com/2013/08/14/core-data/scaling-apache-giraph-to-a-trillion-edges/
2. Bachan, J., Bonachea, D., Hargrove, P.H., Hofmeyr, S., Jacquelin, M., Kamil, A., van Straalen, B., Baden, S.B.: The upc++ pgas library for exascale computing. In: Proceedings of the Second Annual PGAS Applications Workshop. PAW17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3144779.3169108, https://doi.org/10.1145/3144779.3169108
3. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (aug 2011). https://doi.org/10.1145/2024716.2024718, https://doi-org.proxy.uchicago.edu/10.1145/2024716.2024718
4. Biswas, R., Oliker, L., Shan, H.: Parallel computing strategies for irregular algorithms. Annual review of scalable computing **5**, 1 (2003)
5. The c++ reference manual, available from https://en.cppreference.com/w/
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (jan 2008). https://doi.org/10.1145/1327452.1327492, https://doi.org/10.1145/1327452.1327492
7. Dittrich, J., Quiané-Ruiz, J.A.: Efficient big data processing in hadoop mapreduce. Proc. VLDB Endow. **5**(12), 2014–2015 (aug 2012). https://doi.org/10.14778/2367502.2367562, https://doi-org.proxy.uchicago.edu/10.14778/2367502.2367562

8. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (jan 1985). https://doi.org/10.1145/2363.2433, https://doi-org.proxy.uchicago.edu/10.1145/2363.2433
9. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed Graph-Parallel computation on natural graphs. In: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 17–30. USENIX Association, Hollywood, CA (Oct 2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez
10. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press (2014)
11. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. p. 31–46. OSDI'12, USENIX Association, USA (2012)
12. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.: Graphlab: A new framework for parallel machine learning. In: Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence. p. 340–349. UAI'10, AUAI Press, Arlington, Virginia, USA (2010)
13. Lusk, E., Butler, R., Pieper, S.C.: Evolution of a minimal parallel programming model. Int. J. High Perform. Comput. Appl. **32**(1), 4–13 (jan 2018). https://doi.org/10.1177/1094342017703448, https://doi.org/10.1177/1094342017703448
14. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. p. 135–146. SIGMOD '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1807167.1807184, https://doi.org/10.1145/1807167.1807184
15. Marlow, S., et al.: Haskell 2010 language report. Available online http://www.haskell. org/(May 2011) (2010)
16. McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P., Levin, M.I.: LISP 1.5 programmer's manual. MIT press (1962)
17. Minutoli, M., Drocco, M., Halappanavar, M., Tumeo, A., Kalyanaraman, A.: Curipples: Influence maximization on multi-gpu systems. In: Proceedings of the 34th ACM International Conference on Supercomputing. ICS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3392717.3392750, https://doi.org/10.1145/3392717.3392750
18. Minutoli, M., Halappanavar, M., Kalyanaraman, A., Sathanur, A., Mcclure, R., McDermott, J.: Fast and scalable implementations of influence maximization algorithms. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–12 (2019). https://doi.org/10.1109/CLUSTER.2019.8890991
19. Nieplocha, J., Harrison, R.J., Littlefield, R.J.: Global arrays: A portable "shared-memory" programming model for distributed memory computers. In: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing. p. 340–349. Supercomputing '94, IEEE Computer Society Press, Washington, DC, USA (1994)
20. Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: The scala language specification (2004)
21. Rajasukumar, A.: Updown: An intelligent data movement architecture for large scale graph processing. Tech. rep., University

of Chicago, Computer Science (2023), tR-2023-03, Available from https://newtraell.cs.uchicago.edu/research/publications/techreports/TR-2023-03

22. Shun, J., Blelloch, G.E.: Ligra: A lightweight graph processing framework for shared memory. SIGPLAN Not. **48**(8), 135–146 (feb 2013). https://doi.org/10.1145/2517327.2442530, https://doi.org/10.1145/2517327.2442530
23. Welford, B.P.: Note on a method for calculating corrected sums of squares and products. Technometrics **4**(3), 419–420 (1962). https://doi.org/10.1080/00401706.1962.10490022, https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022