Faculty of Science and Technology
Department of Computer Science

# UPC++ Partitioned Map

*Exploiting data locality in high performance computing using PGAS mechanisms*

—

**Aril Bernhard Ovesen**
*INF-3983 Capstone Project in Computer Science - December 2017*

## Contents

**Abstract**

This report presents UPC++ Partitioned Map, a distributed key/value store and map/reduce framework implemented in UPC++. PGAS communication functionality allows for fine grained remote access between any partition of the key/value store, and hierarchical separation of keys allows for more deterministic placement of data. Benchmarks show viable performance in comparison with other modern systems, and that there is a possibility for improved performance within general purpose computing through more advanced abstractions and configuration options.

# 1 Introduction

Parallelization frameworks, programming systems, and data abstractions are important components of high performance computing. While computing frameworks aim to allow programmers to express problems with semantics that allow for highest possible performance, few contemporary systems allow expression of data locality, especially the locality of arbitrary data structures that can be heterogeneously distributed [1]. A common premise is that the computation itself is the most expensive segment in a high performance computing pipeline. For example, the time complexity of an algorithm is expressed through the number of required arithmetic operations, ignoring the performance cost of data movement [2].

In computation frameworks built for supercomputers, exploiting data locality can reduce network usage, which is a bottleneck in data-intensive computing. Data intensive computing is of increasing importance as more data is generated through the Internet or instruments such as high resolution microscopes [3] or gene sequencers [4].

Conventional data abstractions are designed for compute-centric computing may not be suitable for a data-centric paradigm. There exists a perception that abstraction carry an inherent performance penalty, but abstractions can also be a powerful tool in high performance computing [5]. Computing at an impractically large scale can be reasoned about by programmers through abstracting sets of operations into tasks. If the tasks can be separated into independent operations, they can be distributed across compute nodes and executed in parallel. The task abstraction can be extended to exploit data locality by including information about which data the task requires. While this abstraction is more complex, it enables the creation of parallelization frameworks and data structures that are inherently distributed with regards to data locality, by applying the same logic that data is distributed by, to the distribution of computational tasks.

With these lessons in mind, a novel data structure for C++ was devised. In order to make assumptions about the distribution of data, it must be stored in an associative data structure, and in order to be usable in different computation contexts, the data structure should be abstract. Through this reasoning, the novel data structure, the UPC++ partitioned map, distributes data in a key/value store, namely a hash table. Parallelization is achieved through UPC++, a PGAS extension for C++ [6].

The goal of the UPC++ partitioned map is to provide a data structure that is inherently distributed and can be used for high performance computation, for data that can be key-indexed and tasks that require data look-ups. Hash digests of keys are used for distribution across cores or nodes. Because the hash function used in this system provides uniformly distributed digests, keys are made hierarchical to provide a technique of achieving similar distribution of items with relation to each other. This means that the distribution of key/value items is not necessarily decided by a single key, but the distribution scheme can be customized with several layers of keys to fit a particular computational problem.

## 1.1 Contributions

In summary, the following contributions are made: A novel design of a distributed key/value store utilizing PGAS functionality in UPC++ is described and implemented. A framework for distributed computations using Map/Reduce model is implemented alongside the key/value store, aiming to utilize the data locality used by storage for high performance and big data computing. An implementation of a modular distribution scheme that utilizes a hierarchical key structure to distribute data is also presented.

## 1.2 Outline

Section 2 describes fundamental PGAS concepts and languages, including UPC++. The goal of this section is to explain PGAS concepts to a degree that the design and motivation behind the implementation can be understood. It is not intended as an exhaustive resource, and is largely based on the UPC++ specification [7] and programmer's guide [8]. Section 3 describes the design of the system, including data storage, computation models, and distribution schemes. Section 4 describes some related systems that apply similar approaches to data storage and computing as the UPC++ partitioned map, and also describes the relation and differences between the systems in question. Section 5 describes how this system has been implemented using UPC++. Section 6 presents an evaluation of computation characteristics the system. Section 7 discusses potential changes and improvements to the design and implementation of the system, while Section 8 concludes this report.

# 2 Background

## 2.1 Partitioned Global Address Space

Traditionally, parallel programming (and the field of High Performance Computing) has been clustered in two language paradigms: *shared memory* and *message passing* [9]. Shared memory models provide an abstract memory space that is available for access across several concurrent compute threads. Message passing models, such as MPI [10], give a programmer control of the flow of execution by conceptually bundling most of the communication into a message abstraction. This approach can be used to achieve high performance; the programmer can reason about the cost of communication, and messages can be delivered between compute nodes to scale a program to large, multi-node systems.

Programming languages that achieve parallelism through a shared memory abstraction that provides memory access to several compute threads concurrently can be classified as Global Address Space (GAS) langauges. Communication between threads is achieved through reading and writing shared data, rather than explicit message passing [11]. Partitioned Global Address Space (PGAS) languages aim to provide a hybrid approach between previous parallelization paradigms, by utilizing the programmability advantages of the shared memory model, and the communication efficiency of message passing.

### 2.1.1 GASNet

GASNet is a communication API for GAS/PGAS languages that aims to provide a portable communication layer to systems that would otherwise be platform specific or network implementation dependent. Portability is achieved through translating PGAS programs to an intermediate C representation, which can be compiled into a platform specific application using the system's standard C compiler [11].

GASNet is distributed with several API implementations with support for different underlying network architectures, called *conduits* [12]. This means that systems built on GASNet can be implemented to utilize different communication interfaces, including, among others, MPI, shared memory, UDP/IP, and InfiniBand verbs.

## 2.2 Unified Parallel C

Unified Parallel C, referred to as UPC, is a superset of the C programming language for writing SPMD programs, utilizing a global address space paradigm for parallelism. A complete UPC specification was first standardized in 2001, and several UPC compilers have been under development since, including GNU UPC, which is an extension of GCC [13], and Berkeley UPC, which is dependent on GASNet [14].

The address space of a UPC program is partitioned into several logical fragments, with each concurrent thread being mapped to one fragment. With this model, each thread has affinity to its own individual memory region, which makes it possible to exploit data locality [15] and utilize parallelism with the same language constructs.

Handling data in a memory partition mapped to a single thread makes it implicit that data is being accessed by a thread that has affinity to the given partition. Memory can also be explicitly distributed across several threads, using shared global memory, which disregards any thread affinity and is logically separated from the aforementioned thread-mapped memory partitions.

The shared memory abstraction of UPC is provided regardless of underlying hardware structure. Partitioned memory segments can be mapped to individual compute nodes in a cluster, utilizing data locality by performing computation on private data close to the location where it is stored, while still providing a shared memory abstraction for communication and programmability. The original UPC language specification states that a driving principle behind UPC is that parallelism and remote memory access should not obfuscate the resulting program; programmers should themselves utilize the language constructs to design programs and data structures, instead of being provided with solutions to specialized tasks. Additionally, the specification claims that a shared memory programming model is attractive for users of distributed memory systems [16]. These two concepts can in part make up the motivation behind the UPC language.

## 2.3 UPC++

UPC++ is a PGAS extension to the C++ programming language. It is developed at University of California, Berkeley and saw the release of its 1.0 version in September 2017. UPC++ is implemented as a library extension, due to the complexity of developing a separate C++ compiler [17]. The library heavily utilizes C++ template programming [18] for PGAS mechanisms, which allows the creation of library functions that use generic variables that can be adapted to different types.

### 2.3.1 Programming model

UPC++ follows the SPMD model and like UPC utilizes threads with affinity to memory partitions. While UPC++ allows the use of C++ functionality such as objects and lambdas, it still aims to provide high performance and explicit

```
void main()
{
    upcxx::init();
    cout << "Hello world! " << upcxx::rank_me() << endl;
    upcxx::finalize();
}
```

```
Output:
Hello world! 0
Hello world! 2
Hello world! 3
Hello world! 1
```

**Listing 1:** "Hello World"-program in UPC++ and its output

communication by leaving much responsibility regarding concurrency and data movement to the programmer.

Each UPC++ partition has access to its own private memory segment, and is identified by its rank, an integer that can range from zero to the total number of partitions. Listing 1 demonstrates a simple program that is started with 4 threads, each printing their rank, illustrating the Single Program Multiple Data property of UPC++ programs. Ranks can communicate through a logical shared memory segment, by utilizing UPC++ communication functionality, including global pointers, remote procedure calls (RPCs), and distributed objects. The number of ranks is static during execution; the total rank count is specified when launching a program and cannot be modified during runtime.

Through communication or internal operations, a single thread may consist of several independent tasks that need access to computing resources for the program to continue its execution. In order to keep communication data movement predictable, a programmer will need to yield control of execution. This means that operations that require communication, such as RPC or global memory accesses, are separated from the main thread of execution and will not expend computing resources until explicitly granted permission to do so. Remote and asynchronous tasks are exposed to the caller through *future* objects, which will change their internal state when the task has completed. This allows the caller to perform operations dependent on the completion of remote tasks or retrieval of remote data, and effectively communicate between ranks.

The 1.0 release of UPC++ introduced the concept of *distributed objects*, which are objects defined by a generic type within UPC++ with the same identifier across all ranks. This means that data stored in the private memory segment of a partition can be referenced by a different rank in RPC calls. The data within the object is not implicitly distributed, but the name of the variable is. Local contents of the

```
int main(int argc, char *argv[])
{
    upcxx::init();

    auto myObj = upcxx::dist_object<myClass>(myClass());
    upcxx::barrier();

    if(upcxx::rank_me() == 1) {
        auto f = upcxx::rpc(2, [&]() {
            return (*myObj).myProperty;
        });

        f.wait();
        cout << f.result() << endl;
    }

    upcxx::barrier();
    upcxx::finalize();
}
```

**Listing 2:** Communication through RPC, referencing a distributed object of an arbitrary C++ class. Rank 1 retrieves and prints *myProperty* from rank 2's instance of *myObj*.

distributed objects can as such be explicitly distributed to any rank.

### 2.3.2 Progress

Outstanding asynchronous operations are not completed automatically, because the program consists of a static number of threads. In order to advance the internal state of UPC++ or complete RPC calls, computing resources are yielded from the main thread of execution via the *progress()* function. This concept puts the responsibility of frequently progressing UPC++ state on the programmer's shoulders. The goal of this is to provide visibility of resource usage and parallelism, the latter being important to achieve interoperability with libraries and software packages that would otherwise restrict the use of multi-threading. Being cautious about where to progress RPCs relieves the needs for concurrency control such as mutexes, and implies atomicity in partition-internal operations of basic data types. More complex data structures that are not designed for concurrent access can be safely referenced in RPCs across partitions as long as the *progress()* function is used only between atomic accesses. This is demonstrated further in the Implementation section of this report.

RPCs can be used for both modifying and copying data in the private segment of another rank's memory. The future objects returned by UPC++ RPCs can be used to indicate completion of the remote task, and communicate return values. A program that is dependent on data from a different rank, can halt advancement of the main thread by using the *future.wait()* function, during which progress is attempted until the given future indicates that its corresponding remote operation has completed. This is not possible to achieve in tasks executed by progressing – futures can only change state during progress, and progress-callbacks cannot themselves initiate further progress of other tasks. This concept provides a clear distinction between execution via the main thread of a rank, and execution via user-initiated progress of the same rank.

Listing 2 demonstrates communication between two ranks using RPCs, distributed objects, and futures. A distributed object of an arbitrary class is initiated on all ranks, and a RPC is invoked to retrieve a property of the instance of the distributed object *myObj* local to rank 2, back to rank 1. The RPC returns future $f$, which will hold a copy of the remote *myProperty*, which is available as *f.result()* when returning from *f.wait()*. A barrier is used to ensure that the identifier of the distributed object is available on all ranks before it is referenced in the following RPC.

### 2.3.3 Global pointers

Basic built-in C++ datatypes can be transmitted between partitions as RPC arguments or return values, by passing them as a copy. To communicate pointers to other partitions, the shared memory segment must be utilized. UPC++ can create a pointer to memory allocated in the shared memory segment, and copy local data to this location. The global pointer can be passed to other partitions through RPCs. Accessing global memory from a partition other than the one that initiated the pointer has to be done through asynchronous operations. UPC++ provides functions for remote puts and gets, which require internal progress on the partition that is hosting the shared memory in order to complete.

# 3 Design

This report presents a UPC++ partitioned map, a key/value store with multimap properties and hierarchical keys, in which the key-space is distributed across different partitions, communicating with PGAS mechanisms in UPC++. The goal is to provide an interface for parallel computing where the location of the computing process is decided by data locality.

## 3.1 Data storage

Hashing is actively utilized to determine where to store and search for data. At the partition level, key/value pairs are stored in a hash table, chaining keys that are hashed to the same entry together in linked lists. Each key can contain several values, which are stored in lists referenced by the entry with the corresponding key. Looking at each partition individually, the data structure is equivalent to a multi map hash table with separate chaining. The internal organization of data in the hash table is not important for the holistic functionality of the system, but the semantics of key/value operations are. Each key can map to several values, which means that inserting a new value with a pre-existing key will not overwrite or delete any data. Retrieving data corresponding to a key will return a list of all existing values mapped to that key, with the newest key first. This means that insertions will append new values to the beginning of the list of chained values.

Figure 1 illustrates the relation between key/value pairs and hash table entries. Hash tables on different partitions are logically independent of each other; PGAS mechanisms separate the execution threads and memory segments of different hash tables. The home partition of a key is deterministic, and an operation that

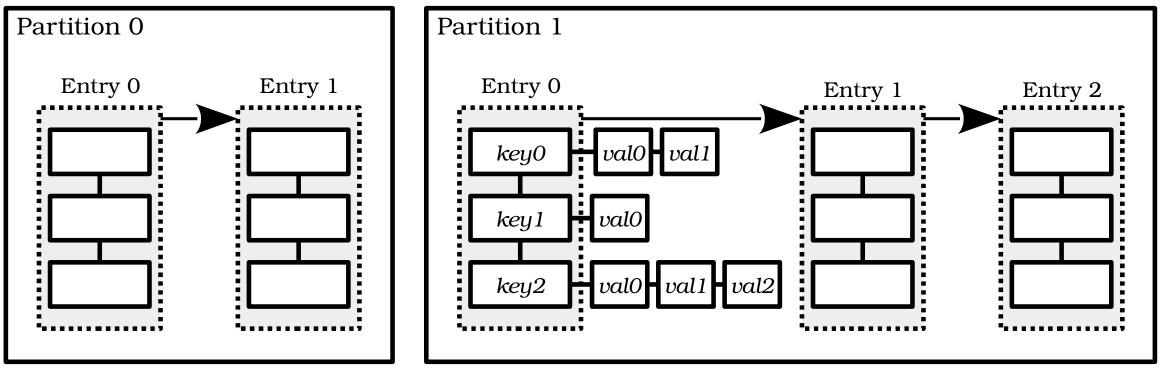

**Figure 1:** General overview of a partitioned hash table, its entries, and their chained multiple values.

is inserting or retrieving a key/value pair will never traverse the hash table of more than one partition in order to find a key or conclude that the key is not present.

## 3.2 Hierarchical keys

The partitioned map can utilize hierarchical keys to determine the desired partition to store a key in. Hierarchical keys is a concept in which a single key is treated as a concatenation of several sub-keys, and each sub-key influences which partition the entire key/value pair should be stored in. Each sub-key will split the remaining partition space into smaller groups, and determine which group of the partition space to use based on the value of the key, and the number of groups.

The act of determining a partition, given a key, is similar to searching through a tree, in which each sub-key determines the direction of search in one layer. The final layer of the tree is equal to the total number of partitions. The depth of such a tree is equal to the total number of sub-keys plus 1, and is illustrated in figure 2. This tree has a key-hierarchy of 3 levels, and a depth of 4.

This example illustrates a hierarchy structured like a binary tree, with each node in the tree fanning out to two new nodes. Further explained in the implementation section, the key hierarchy is modifiable per instance of the partitioned map, and should define sub-keys based on how the data is accessed. This is because the goal of using hierarchical keys is to achieve meaningful data locality relations between keys, while still attempting to exploit the data spread of distributing based on hashing. If the sub-keys are chosen arbitrarily, or the data access follows a pattern that has no relation to the hierarchical grouping, the system would likely

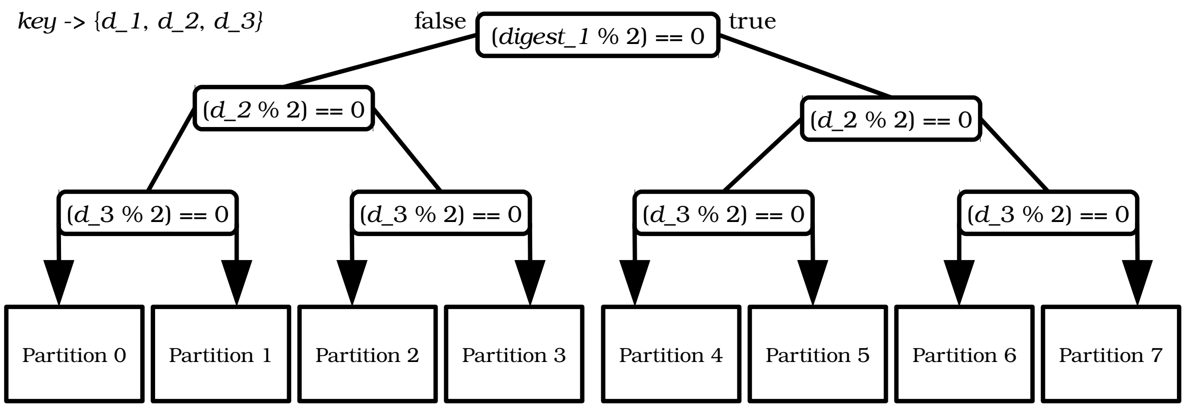

**Figure 2:** A key split into a hierarchy of height 4, with 8 partitions. This hierarchy is structued like a binary tree.

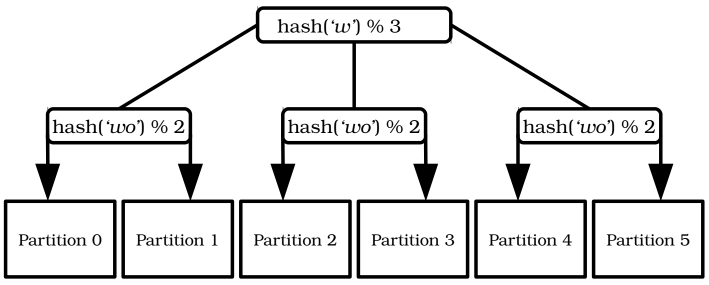

**Figure 3:** An example of a distribution scheme with a hierarchy of height 3, determining the location of the key '*word*' based on its first letters.

be better off using a single hash digest as distribution scheme. While illustrated as a tree, the hierarchical grouping of partitions is purely logical; the total number of hash tables is independent of the hierarchy. As mentioned in the previous section, individual hash tables bear no influence on each other.

Figure 3 shows a similar situation with a specific distribution scheme. A hierarchy of height 3 is used to distribute keys based on the first two letters in them. The first sub-key, a hash of the first letter, splits the partition space into three segments, each containing two partitions. The next sub-key, a hash of the first *two* letters, reduces this to one partition. With this hierarchical distribution scheme, it can be concluded that keys with the same first letter is placed in the same group of two partitions. All keys where the first two letters are identical will always be placed in the same partition. The relation between keys in the same partition with different first letters is effectively arbitrary, because it is depends on the hash digests and number of partitions.

## 3.3 Distributed computing

The motivation behind the design of the UPC++ partitioned map is to provide a data structure that can be utilized for both data storage and distributed computing, resulting in locality-aware computing.

The distribution of data that is already present through PGAS partitions can be utilized for distributed computing, by applying *collective operations* on the hash table entries. Collective operations are arbitrary functions with a fixed signature that are executed on every entry in the data structure. This execution model distributes the computation process with the same scheme as data insertions and retrievals; a collective operation is equivalent to a function that iteratively looks up every key in the a table, and has access to read or modify every value.

### 3.3.1 Map/Reduce model

Map/Reduce is a parallel programming model in which execution is split into a mapping phase, and a reduction phase. The core execution plan involves mapping data elements with some common factor together, and performing a reduction function on entire sets of data that has been mapped to the same identifier. A map function can be described as filter operation, while the reduce function performs a summary operation, with the purpose of reducing a set of data down to one element. The UPC++ partitioned map supports the map/reduce programming model by providing insertions as mapping functions, where the index key of a key/value pair decides the mapping. Collective operations can be used as reduce functions, where the chained values of one key represents one data-set to be summarized by reduction.

Collective functions used for reductions have are arbitrary functionality, and their signatures must follow a set pattern to support iterating through all values. The separation between reduce operations and regular collective operations is handled internally, and removing extraneous entries after a reduction has been performed is not the responsibility of the collective function passed to the partitioned map by a user.

### 3.3.2 Data locality

The same logic that determines the home partition of a key/value-pair can be used to distribute computational tasks. Because UPC++ creates SPMD programs, a design goal of the UPC++ partitioned map is to not require explicit task distribution. The identity of the parallel computing units are exposed through their ranks in the program, but parallelism should not necessarily be achieved through distributing computational task throughout the range of available ranks. The location of computation should rather be determined by the location of the data that is required by the task, assuming that this data is stored within the map. Structuring a program in this way is possible because the location of any given key/value pair is deterministic, and it is achieved through exposing the rank of the target partition to the SPMD program.

Following the principles of consistent hashing, using a uniformly distributed hash function should result in a near-uniform distribution of data [19]. This can be utilized to also achieve near-uniform distribution of tasks. The goal of structuring a program like this is to minimize the number of remote operations that a map needs to perform. It is possible to use the key hierarchy to reduce the number ranks executing a task down to any group that exists in the hierarchy (see the sub-trees in figure 2), and allocate a single task to multiple ranks.

Exploiting data locality in the manner described here is only possible when the computational task requires data that exists in the tree, and where a part of the key expressed in the key hierarchy is known. This means that choosing a distribution scheme that matches the data access pattern during computation is necessary in order to exploit data locality. It is for this reason that the layout of the key hierarchy is configurable, and alternative distribution schemes can be implemented and applied to the data structure.

# 4 Related work

## 4.1 Legion

Legion [1] is a runtime system and computation framework that aims to reduce the impact of data movement costs in high performance computing. It focuses on providing high performance and power efficiency through a data-centric programming model. This model involves providing abstractions for program data, logical regions, that are used to utilize locality during computation. Logical regions are program data stored in a structured table, which can be split into sub-regions, which means that data can indicate both locality (through its location in the table), and independence (through its partitioning into sub-tables). A design philosophy of Legion asserts that it is impossible to achieve optimal decomposition of data through a general algorithm, and that the programming model should express explicitly how both program data and algorithms are partitioned to achieve the desired parallelism [20]. The table of program data is purely logical, and can consist of multiple physical instances. This allows replication of read-only data and parallel execution of independent tasks.

The design of the UPC++ partitioned map is related to the Legion's programming model of explicit expression of data locality. When using UPC++ partitioned map as a computing framework, the partitioning of data should be explicitly expressed through the distribution scheme, so that partition-internal computation can be performed independently. A clear difference between the two systems is that Legion's logical regions can be physically manifested as multiple, replicated instances of itself, while the logical regions that are being expressed in the programming model are, as the name implies, purely logical. In the UPC++ map, the tables and data structures of different partitions can be logically expressed when they are not physically present, through UPC++ distributed object naming. The contents of the table, however, are only manifested in one partition, and their relation to hardware units of partitioning, such as cores, threads, or nodes, are explicit and static.

## 4.2 DynamoDB

DynamoDB is a key/value storage system, created and used by Amazon [21]. It provides three main abstractions for interacting with the database; tables, items, and attributes. Tables are collections if items, which are collections of attributes. While the attributes of an item is reminiscent of a database schema, Dynamo does not provide relational indexing. A primary key must exist among the attributes of an item, and indexing is performed by hashing this attribute. The primary key can consist of two attributes, which are referred to as the *partition key*, and the *sort key*. The physical location of the key/value pair is determined by the hash digest of the partition key, and the sort key is used to determine the its location within a partition. This means that all entries with identical partition keys will be found in the same partition, sorted after their sort key. In addition to this, queries can search for keys filtered by up to five additional *secondary keys* [22]. Because key/value pairs are deterministically determined by multiple keys, it is possible to perform general lookups that return more than one entry, if not all fields are included in the query.

The possibility of multi-indexed key/value-pair indexing is similar to the hierarchical keys presented in the UPC++ partitioned map. A core difference is that Dynamo's multi-layered indexing is used to refine a search *within an already decided partition*, while the multi-layered indexing presented in this report refines the search *down to a single partition*. In the UPC++ partitioned map, the location of a key/value pair within a partition is arbitrarily determined by the hash digest of the entire key, and the hierarchical separation into multiple keys have no partition-internal effect. Dynamo's queries are different in that they require a primary key to determine a target partition in which to search for the secondary keys at all. This presents two fundamentally different approaches to multi-layered lookups in a distributed key/value store.

## 4.3 DataMPI

DataMPI is an extension to MPI that aims to bridge high performance computing and big data computing, by modifying communication techniques of the former, to support the latter [23]. The authors of DataMPI observes that MPI communication techniques are optimized for many network interfaces, such as InfiniBand, and while they are utilized for high performance computation, they outperform current Hadoop communication protocols, which are often used in big data computation. DataMPI extends MPI by providing interfaces for key/value pair-based communication by incorporating key/value fields directly into the *send* and *receive* primitives of MPI.

Like DataMPI, the UPC++ partitioned map aims to utilize a generic parallel computation framework to perform key/value pair-based communication through remote operations and map/reduce operations. A key difference between the two systems is that DataMPI is an *extension* to a communication interface, while the UPC++ partitioned map is built *on top of* UPC++. The functionality that allows PGAS mechanisms to send and retrieve key/value pairs between hash tables is not directly built into the UPC++ library. The primitives used for key/value pair-based communication in DataMPI, on the other hand, is provided by the parallelization library itself.

## 4.4 RAMCloud

RAMCloud is a distributed key/value store system in which data is primarily handled in DRAM, as opposed to on disk [24]. The authors of RAMCloud assert that frameworks for batch computation of large amounts of data, such as Apache Spark, is not suitable for large scale web applications and random memory access. While RAMCloud is a storage system, and not designed for high performance computing, it is considered a related work to the UPC++ partitioned map because both systems provide a hash-based key/value store that can be distributed across multiple nodes, and particularly focuses on DRAM storage. Like Dynamo, RAMCloud uses consistent hashing to distribute keys, based on their hash digests and per-node unique IDs [19].

# 5 Implementation

## 5.1 Data structure

Hash tables are implemented with the purpose of being initialized as UPC++ distributed objects. These objects are instances of the *DistributedKV* class, and in an SPMD context they represent the entire partitioned map. reference The class contains an array of table entries, in addition to properties for table expansion logic and distribution scheme. Both keys and values are implemented as *char* pointers, so that they serve a general purpose and can be cast to any arbitrary structure. Each hash table entry is stored as a separate *struct*, containing a key pointer, an integer indicating key length, and a pointer to the value struct. Value structs contain a pointer to the actual value, an integer indicating value length, and a pointer to other, chained values.

The distribution scheme is implemented as a separate class, which the DistributedKV class contains an object of. The *Distribution_Scheme* interface is used to provide alternative implementations, so that partitioning logic can be modified to suit a particular problem. The virtual methods of this interface is illustrated in listing 3. The *depth* parameter is used to indicate how many levels of the hierarchy that should be left. Consider the example in figure 3, where the hierarchy has 3 layers. Using a depth parameter of 0, the range of valid partitions would be reduced to a single partition. With a depth of 1, it would be narrowed down to 2 partitions, and a depth of 2 would not narrow the number of potential partitions down at all. Following the same example, if one wanted to execute a task on all partitions containing keys with a specific first letter, a depth of 1 should be used in the *in_partition* method of the distribution scheme.

## 5.2 Program structure

All UPC++ partitioned map functions follow one of two different execution models: *local*, and *anylocality*. The difference between functions of these two models is how they treat data stored in other partitions. Local operations will only affect the hash table belonging to the partition on which it was called, while anylocality-operations can potentially (but not necessarily) initiate RPCs.

Collective functions and reduce functions are local operations, because they will only perform operations on local data. In order to execute a function of this type on all data stored in a map, they must be executed in a part of the SPMD program that is not limited to a certain number of ranks, or invoked as a local operation on a remote partition, via RPC.

Two versions of key/value insertions and retrievals are implemented, one for each execution model. Local insertions will insert a key only if the distribution scheme

```
virtual int get_target_partition(char* key, int keylen,
                                              int hashed_key)
{
  /* Checks which partition 'key' should be placed in,
  starting from the top of the hierarchy
  Returns the rank of the target partition */
}

virtual bool in_partition(int n, int depth, char* key,
                                                int keylen)
{
  /* Returns true if rank 'n' can contain 'key'
  according to the hierarchy down to 'depth' */
  if(depth == 0)
    return (get_target_partition(key, keylen,
                                 hash(key, keylen)) == n);
}
```

**Listing 3:** The methods of the distribution scheme class that must be overriden to create a new valid distribution scheme

deems that it should be inserted in current rank. Anylocality insertions will behave in the same manner, but send the key/value pair to the correct partition using an RPC if it cannot be inserted locally. This is illustrated in listings 4 and 5, where example programs of respectively local and any-locality insertions are shown. In listing 4, each partition retrieves all keys that should be distributed, from an arbitrary function. Even if this program is executed with 4 partitions, only 8 keys will be inserted in total. Each rank in listing 5 retrieves only a subset of all keys and values. The example shows 2 key/value pairs per rank, which means that with 4 partitions, 8 total insertions to the table are made. This is possible in a case where some part of the program is filtered by the identity of each rank. In the case of listing 5, the function calls to *get_some_keys* and *get_some_vals* takes a rank as parameter, which can be used as a filter to make sure that each rank retrieves different data.

## 5.3 Concurrency control

No concurrency control mechanisms, such as mutexes and semaphores, have been implemented in this system. Because only one thread has affinity to each UPC++ partition, data can not be modified concurrently. Unless other parallelization methods (such as OpenMP) are explicitly integrated into the program, concurrent in private memory segments will not occur.

A related issue is that of atomicity. Progressing yields computing resources away

```
void main() {
  auto db = init_distrib_kv(initial_size, max_chaining);
  auto keys = get_all_keys();
  auto vals = get_all_vals();
  for(int i = 0; i < 8; i++)
    insert_if_local(db, keys[i], vals[i])
}
```

**Listing 4:** An example program, inserting 8 arbitrary keys and values. Each rank executes all functions in this program, but each key is only inserted once.

```
void main() {
  auto db = init_distrib_kv(initial_size, max_chaining);
  auto keys = get_some_keys(upcxx::rank_me());
  auto vals = get_some_vals(upcxx::rank_me());
  for(int i = 0; i < 2; i++)
    insert_anylocality(db, keys[i], vals[i])
  }
```

**Listing 5:** An example program, inserting 8 arbitrary keys and values, assuing a program with 4 partitions. Each rank will only insert 2 keys each.

from the main flow of execution, but can in many cases modify the same data as the main program. Examples include remote insertions, or hash table expansions as a result of such operations. This means that the progress function must be called in the main program only when no atomic operations are being performed. In other words, operations surrounding a *progress()* call is for all intents and purposes atomic, while yielding control from the main thread breaks atomicity.

In example programs implemented with UPC++, where anylocality-insertions were used to insert large amounts of data, progress was made after each insertion, effectively interweaving local outgoing insertions with remote incoming insertions. Progressing is important because incoming RPCs can consume memory.

## 5.4 Hashing

Hash digests of keys are created with CityHash [25], a family of hash functions with implementations by Google in C++. It is used in this implementation to provide 32 bit hashes of keys, and they are stored in the program as 32 bit integers.

```
uint32_t modulo(uint32_t b, uint32_t c) {
    return b % c;
}

uint32_t fastrange(uint32_t b, uint32_t c) {
    return ((uint64_t) b * (uint64_t) c) >> 32 ;
}

uint32_t bitwise_modulo(uint32_t b, uint32_t c) {
    return (b & (c-1)); //c = 2^n, n is a positive integer
}
```

**Listing 6:** Modulo and alternative implementations

## 5.5 Modulo

Previous explanations of the hash table and key hierarchy, including figures 2 and 3, use the modulo operator (%) as a way to deterministically derive a number $A$ from number $B$, where $A$ has a maximum value of $C$. An important property is that, if $B$ is uniformly distributed, $A$ should also be uniformly distributed. While modulo operators are easy to reason about, they can be replaced by more efficient implementations. Lemire [26] defines *fastrange* as a more efficient alternative, and a modulo equivalent bitwise operation exists in cases where $C$ is a power of two. Implementations of these operations are shown in listing 6.

## 5.6 Expansion

Key/value-pairs that are hashed to the same entry are stored in linked lists, a technique known as separate chaining [2]. Because the a key can map to multiple values, the value structs are also stored in linked lists. When any separate chain reaches a pre-determined limit, the table expands and re-inserts each entry. Expansion is performed by allocating a new array of entries of twice the size of the current array. The pointer to the current array is stored, and the entry-array property of the DistributedKV object is set to the new array. The old array is iterated through, and each entry is inserted into the new array, before deallocating the old array. Pointers to keys and values are not deallocated, but all structs used for organizing the data are. This operation is atomic; no progressing is performed during expansion.

# 6 Evaluation

## 6.1 Set-up

For benchmarking, a word count program was implemented. This program takes a text file as input, and counts the occurrences of different unique words. This is implemented with a Map/reduce model, where the mapping function inserts each word in the file into the hash table, with the word as key and the integer '1' as value. The reduction function will sum these values, where one entry of the number '1' indicates one occurrence of the word.

Two word count implementations were made in UPC++; one using the partitioned map as described in sections 4 and 5, and one using a modified data structure. The modified UPC++ partitioned map is optimized to solve the word count problem. While the map is designed to serve for general purpose data storage and computation, the UPC++ word count implementation aims to use the same structure for data storage, but without any regard for generalizable abstractions. This means that the UPC++ word count does not store data as pointers, but rather C++ strings, so that they can be passed directly through RPCs, without any additional memory allocation or communication of shared-memory pointers. In addition, the UPC++ word count program does not strictly follow the map/reduce model; the problem is not solved in two phases, and the reduction function is applied every time a repeated word is inserted. This means that the linked list of values will never exceed one entry.

The goal of using two implementations of the same problem is to evaluate the effectiveness of the available abstractions and configuration options in the UPC++ partitioned map. The implementation of the partitioned map should be general enough so that it can be configured to perform on the same level as the strict word count implementation. This means that the potential gap between the performance of the two UPC++ applications evaluated could potentially be bridged through more advanced generalization.

UPC++ benchmarks are each ran with two different configurations with regards to file input, following the different program structures explained in the previous section. The 'local only reads' configuration reads the entire input file on each partition, only storing the data that is deemed to be distributed to its own local instance of the hash table. This has the advantage of minimal communication between partitions, with the disadvantage of more disk access, more modulo operations and more hashing. The 'any locality read' configuration splits the file into equally sized blocks, distributed between each partition, and data is distributed between them using UPC++ communication. This has the advantage of only reading the entire file once, but the disadvantage of performance penalties of cross-partition communication. Because of this trade-off, both variants has been included in the benchmark, for both of the word count programs.

```
time airfield 2009 air xhamia widespread flagicon also men
middle fc 2008 albania national cellpadding u club located
alternative align maps
```

**Listing 7:** Example of a 140 byte dataset generated by BigDataBench Wikipedia Text generator

The BigDataBench suite is used for datasets and other benchmark implementations. The BigDataBench text data generation tool aims to provide data sets with the same properties as real-world data [27]. The suite also includes implementations of various problems in different parallelization frameworks and programming languages. For this evaluation, word count implementations in Apache Spark and MPI are used. The BigDataBench Wikipedia text generator is used to generate a data set consisting of words from different Wikipedia articles. Such a data set is illustrated in listing 7.

## 6.2 Environments

Two different environments were used for testing. Firstly, a *single-node* server, consisting of 64 GB RAM, and Intel Xeon CPU E5-2650L v3 CPU running at 1.80GHz with 24 cores, each with 1 CPU thread, and 768KB, 3MB, and 30MB L1, L2, and L3 CPU caches, respectively. The second testing environment is the Abel computing cluster at the University of Oslo. The nodes of which are used in this evaluation consists of Intel E5-2670 CPUs running at 2.6 GHz with 16 cores and 64 GB of RAM. The single node installation of UPC++ uses the *smp* conduit, while the cluster installation uses the *ibv* InfiniBand conduit.

## 6.3 Single node benchmarks

Figure 4 shows the time it took to count all words in 5 gb, 10 gb, and 15 gb files on the single node environment, using 24 UPC++ partitions and 24 MPI threads. It is clear that the BigDataBench MPI implementation is significantly slower than both of the UPC++ variants, and the Apache Spark application. It is also observed that the UPC++ word count program outperforms the UPC++ map, with the any-locality variant of the UPC++ map performing slower than Apache Spark on 10 and 15 gb files. Every local-only variant of UPC++ programs outperforms any any-locality variant, and the any-locality variant of UPC++ word count outperforms Spark.

Figure 5 shows the time it took to execute a similar experiment, with the map stage and reduce stage compared against each other. UPC++ WordCount is not included in this experiment, because it as an optimization does not have separate map and reduce stages, and MPI WordCount is not included because finding

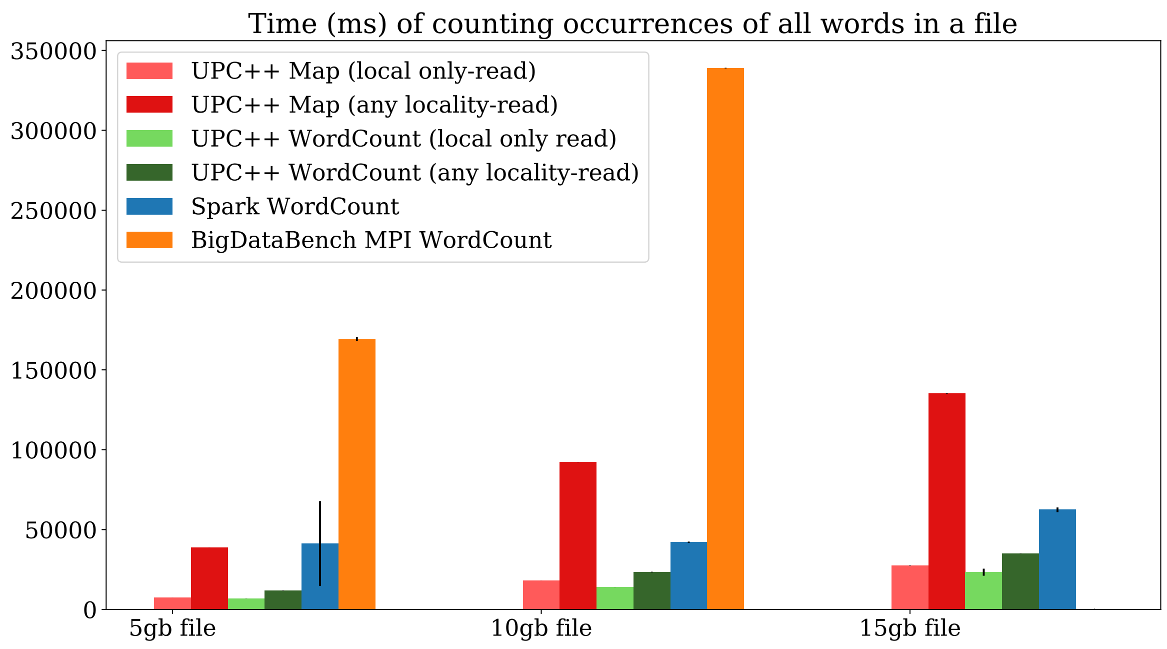

**Figure 4:** Time of performing word count on 5 gb, 10 gb and 15 gb files on the single node environment, executed with 24 partitions/threads.

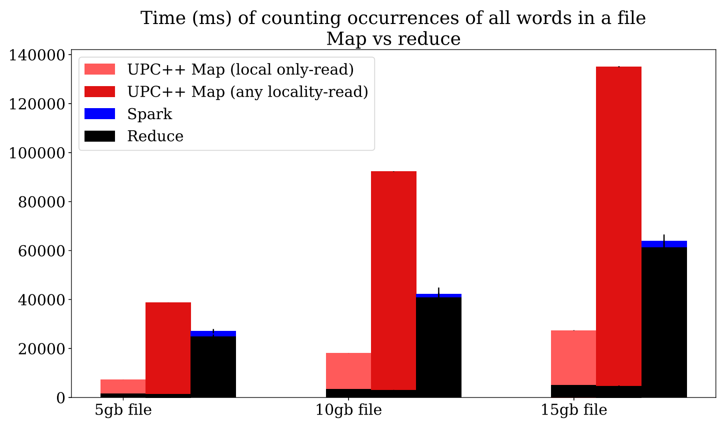

**Figure 5:** Time of performing word count on 5 gb, 10 gb and 15 gb files on the single node environment, executed with 24 partitions/threads, comparing mapping time with reduction time.

the correct timings would involve manually examining the source code of the benchmark to insert the appropriate timers, and re-compiling the BigDataBench-provided binaries from source.

It is clear from these results that the reduce-phase in Spark makes up for a significantly larger portion of the execution time of the program, than in the UPC++ variants. This can be a result of the Spark distributed dataset not shuffling (distributing) data to the appropriate partitions before it is necessary to complete an operation [28], in this case the reduce function. As such, this experiment does not provide a fair comparison of the reduction operation itself in the map/reduce model. Still, the results illustrate that reading from file (mapping) is more efficiently implemented in Spark than it is in the UPC++ programs. With the reduction phase in Spark involving shuffling, this hypothesis makes sense, because the mapping in the UPC++ programs involves their own equivalent shuffle operations, which makes up large portions of their execution time. Despite these significant variations in implementation, it is heavily suggested that mapping with the any-locality configuration is inferior to local-only mapping and the Spark program. The local-only variant of UPC++ provides the most efficient mapping phase, while the reduction phases of the UPC++ programs are equal.

## 6.4 Cluster benchmarks

Figure 6 shows results from running the same benchmark on the cluster. The UPC++ partitioned map anylocality benchmark was not able to complete in this environment, due to issues with GASNet[1]. Cluster benchmarks show similar results as those on the single node environment, with UPC++ WordCount anylocality now performing slower than Spark. It can be assumed that the cost of communication increases when the program is distributed across multiple compute nodes, and anylocality variants, as opposed to local variants, may perform RPC calls after as much as every insertion.

Local insertions provide the best performance, and it is suggested that anylocality insertions should only be used in cases where the cost of retrieving input, for example from disk or network, outweighs cost of communication.

[1]Because the UPC++ partitioned map, as opposed to UPC++ WordCount, require a 2 RPC protocol to perform anylocality insertions, GASNet was unable to complete all RPCs, giving the message that it cannot "exceed 65536 explicit events", a phrase that is not referenced in its documentation. It is possible to restrict the number of RPCs by waiting for completion of insertions on the sender's side, which would in the worst case make insertions sequential across the system.

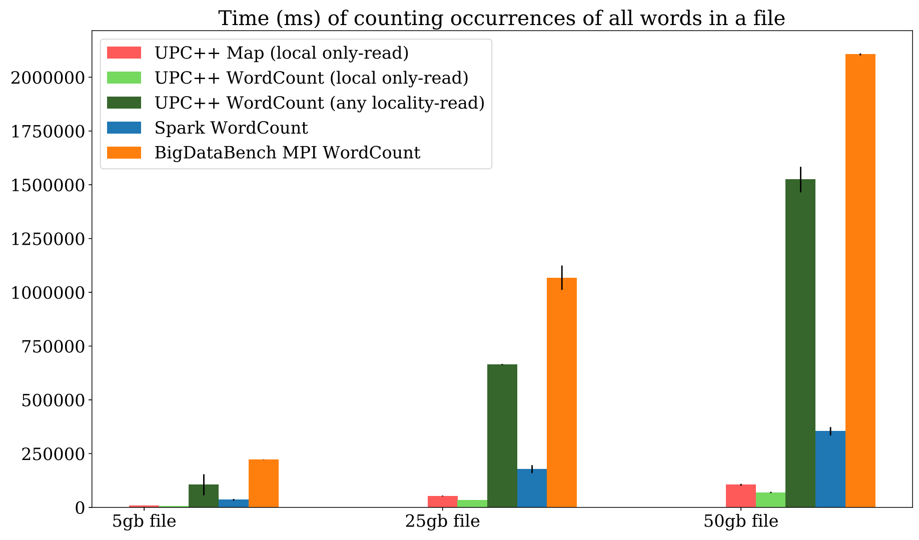

**Figure 6:** Time of performing word count on 5 gb, 25 gb and 50 gb files on the cluster environment, distributed across 8 compute nodes.

## 6.5 Modulo

The modulo alternatives discussed in the previous section were implemented and evaluated in the same word count program. Figure 7 shows that the implementation with fastrange provides the fastest execution time. Bitwise modulo proves slower than using the regular modulo operator.

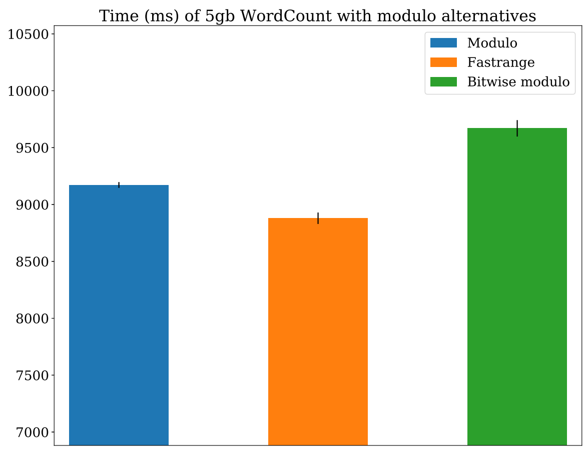

**Figure 7:** Time of performing word count on 5 gb on the single node environment, with different modulo alternatives for determining partitions and table indexes.

# 7 Future work

## 7.1 Improvements

As mentioned in the previous section, the gap in performance between UPC++ partitioned map and UPC++ WordCount that was exposed by benchmarks should be bridged. In order to achieve this, functionality of UPC++ WordCount can be mimicked and implemented as configuration options in the functionality of UPC++ partitioned map. The largest improvement, as shown by the difference in anylocality performance, is to remove the communication overhead of sending global pointers. Char pointers were implemented as the base data type for keys and values in order to implement a general purpose system. Pointers come with additional overhead because communicating them to remote partitions involves memory allocation, copying, and finally deallocation. Passing keys and values in RPCs by copy is possible with other data types, and can be achieved through template programming.

With this approach, each object of the DistributedKV class would be constructed with two type declarations, one key type and one value type. This would make the UPC++ partitioned map closer in functionality with UPC++ WordCount, where keys are C++ standard library strings, and values are integers. This implementation should still allow for a configuration in which global pointers can be used in communication as before.

Support for overlapping map/reduce phases would increase performance in cases where it is possible to omit storing a mapped value in the table, and instead perform the reduction operation directly. The advantage of this approach is that less values are chained, and will not consume memory, and the iteration through the linked list of values during the reduction phase is no longer needed. This approach would only be valid for reduction functions that resemble commutative operations, and would be implemented as a configuration option in addition to current behavior. Commutative operations are operations where the order in which they are performed is irrelevant, such as addition and multiplication of scalars [29].

An improvement of the project a whole is evaluating the effectiveness of utilizing hierarchical keys for data locality. This can be achieved through benchmarks with computation where communication between specific partitions is required.

## 7.2 Expansion

Network topography can be exposed in C programs through the *netloc* software package [30]. If hierarchical keys can be used to expose data locality between partitions, a further improvement can be made on heterogeneous systems by taking network topology into account. Data with relation to each other can be placed closer together in the hierarchy, and as a result, closer together in the network, if the hierarchy is designed around physical network layout.

Distribution schemes can be modified, but the interface assumes that a hashed key exists. An extension to the current use-cases of the UPC++ Partitioned map is to allow distribution of keys with arbitrary logic that does not necessarily involve hashing. For example, keys that are numerically distributed can be partitioned based on their magnitude, similar to Spark's range partitioner [31], for which no hashing is necessary.

## 8 Conclusion

UPC++, a newly released PGAS library for C++, was used to implement a novel data structure for parallel computing, the UPC++ partitioned map. This data structure allow for locality-aware parallel programming by partitioning data in a distributed hash table and executing tasks in threads with affinity to the appropriate data. UPC++ partitioned map supports the map/reduce model for parallel computation, and can provide advanced partitioning logic through hierarchical keys; key/value store indexes that are multi-dimensional and determine data partitioning in multiple user-defined steps. Benchmarking show viable performance with map/reduce programs on single-node and heterogeneous systems, with proposals of several improvements to increase perforamance by increasing configurability. What remains to be done is implementations of programs utilizing hierarchical keys, and performance evaluations of those.

## References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2012.

[2] R. Sedgewick, *Algorithms in C.* Addison-Wesley, 2009.

[3] D. Sage, H. Kirshner, T. Pengo, N. Stuurman, J. Min, S. Manley, and M. Unser, "Quantitative evaluation of software packages for single-molecule localization microscopy," *Nature Methods*, vol. 12, no. 8, p. 717–724, 2015.

[4] Z. Guo, G. Fox, and M. Zhou, "Investigation of Data Locality in MapReduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, (Washington, DC, USA), pp. 419–426, IEEE Computer Society, 2012.

[5] B. C. Mccandless and A. Lumsdaine, "The role of abstraction in high-performance computing," *Lecture Notes in Computer Science Scientific Computing in Object-Oriented Parallel Environments*, p. 203–210, 1997.

[6] "berkeleylab / upcxx / wiki - Home - Bitbucket." `https://bitbucket.org/berkeleylab/upcxx/wiki/Home`. [Online; accessed 26-November-2017].

[7] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, K. Ibrahim, M. Jacquelin, A. Kamil, B. Lelbach, and B. van Straalen, "UPC++ Specification v1.0, Draft 4," 9 2017.

[8] J. Bachan, S. Baden, D. Bonachea, P. Hargrove, S. Hofmeyr, K. Ibrahim, M. Jacquelin, A. Kamil, B. Lelbach, and B. van Straalen, "UPC++ Programmer's Guide, v1.0-2017.9," 2017.

[9] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned Global Address Space Languages," *ACM Comput. Surv.*, vol. 47, pp. 62:1–62:27, May 2015.

[10] "Message Passing Interface." `https://computing.llnl.gov/tutorials/mpi/`. [Online; accessed 27-November-2017].

[11] D. Bonachea and J. Jeong, "GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages," 2002. [Online; accessed 23-November-2017].

[12] D. Bonachea and P. Hargrove, "GASNet Specification, v1.8.1."

[13] "GNU UPC." `http://www.gccupc.org`. [Online; accessed 23-November-2017].

[14] "Berkeley UPC." `http://upc.lbl.gov`. [Online; accessed 23-November-2017].

[15] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An Evaluation of Global

Address Space Languages," *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP 05*, 2005.

[16] W. W. Carlson, D. E. Culler, and E. Brooks, "Introduction to UPC and Language Specification," 1999.

[17] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++ : A PGAS Extension for C++," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[18] "Function template, cppreference.com." `http://en.cppreference.com/w/cpp/language/function_template`. [Online; accessed 23-November-2017].

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, (New York, NY, USA), pp. 654–663, ACM, 1997.

[20] "Legion overview - The Legion Programming System." `http://legion.stanford.edu/overview/index.html`. [Online; accessed 10-December-2017].

[21] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo," *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP 07*, 2007.

[22] "DynamoDB Core Components - Amazon DynamoDB." `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html`. [Online; accessed 10-December-2017].

[23] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: Extending mpi to hadoop-like big data computing," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[24] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud Storage System," *ACM Trans. Comput. Syst.*, vol. 33, pp. 7:1–7:55, Aug. 2015.

[25] "HitHub - google/cityhash." `https://github.com/google/cityhash`. [Online; accessed 10-December-2017].

[26] "A fast alternative to the modulo reduction - Daniel Lemire's blog." `https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/`. [Online; accessed 10-December-2017].

[27] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, and et al., "Bigdatabench: A big data benchmark suite from inter-

net services," *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[28] "Spark Programming Guide - Spark 2.2.0 Documentation." `https://spark.apache.org/docs/latest/rdd-programming-guide.html#shuffle-operations`. [Online; accessed 4-December-2017].

[29] K. H. Rosen and K. Krithivasan, *Discrete mathematics and its applications.* McGraw-Hill Education, 2015.

[30] "Portable Network Locality (netloc)." `https://www.open-mpi.org/projects/netloc/`. [Online; accessed 10-December-2017].

[31] "RangePartitioner." `https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/RangePartitioner.html`. [Online; accessed 10-December-2017].