Communicating Process Architectures 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

A Service-oriented Scalable Dictionary in MPI

Sarwar ALAM [1], Humaira KAMAL and Alan WAGNER

Dept. of Computer Science, University of British Columbia, Canada

Abstract. In this paper we present a distributed, in-memory, message passing implementation of a dynamic ordered dictionary structure. The structure is based on a distributed fine-grain implementation of a skip list that can scale across a cluster of multicore machines. We present a service-oriented approach to the design of distributed data structures in MPI where the skip list elements are active processes that have control over the list operations. Our implementation makes use of the unique features of Fine-Grain MPI and introduces new algorithms and techniques to achieve scalable performance on a cluster of multicore machines. We introduce shortcuts, a mechanism that is used for service discovery, as an optimisation technique to trade-off consistency semantics with performance. Our implementation includes a novel skip list based range query operation. Range-queries are implemented in a way that parallelises the operation and takes advantage of the recursive properties of the skip list structure. We report the performance of the skip list on a medium sized cluster with two hundred cores and show that it achieves scalable performance.

Keywords. Fine-Grain MPI, performance, message-passing, multicore, range-query, skip list, concurrency, data structure, scalability, distributed, dictionary

Introduction

In [1] we introduced the notion of a service-oriented approach to adding distributed data structures to MPI. The lack of data structure libraries in MPI adds to the complexity of using message passing, requiring the programmers to write their own and explicitly manage the distribution of the data. This often leads to less optimised and less sophisticated implementations. In contrast to many MPI libraries that are implemented as a computational phase in an SPMD style program, a service oriented approach provides the data structure as a service using a separate set of processes. Message-passing is used to interface to the service and, as described in [1], leads to programs that are more cohesive and code that is less coupled with the application.

Our initial work highlighted our approach and some of the lessons learned using a simple linked list structure. The methodology started with an item per process design using synchronous messaging and then extended the design to multiple items per process and asynchronous messaging. The design addressed load-balancing issues and introduced shortcuts which led to several different consistency semantics. The linked list structure served to highlight our approach but, as expected, is not scalable and is more useful for synchronisation rather than as a key-value store.

The purpose of this paper is to extend our previous work to an important type of data structure, namely an ordered dictionary. Key-value stores (i.e., dictionaries) are commonly

[1]Corresponding Author: *Sarwar Alam, 201-2366 Main Mall, Vancouver, BC, V6T1Z4, Canada*. E-mail: sarwar@cs.ubc.ca.

used in cloud-based computing environments. Typically these have been designed for reliability and disk-based storage, however, more recently the demand to process real-time streaming data has lead to the use of in-memory key-value storage. A data structure that is often used to implement ordered dictionaries in sequential programs is a skip list [2] (see Section 1). Skip lists are a probabilistic data structure with an expected $O(\log N)$ cost for FIND, INSERT and DELETE. One notable advantage to skip lists over the usual hash-based approach used to implement key-value stores in distributed systems is that the keys in a skip list are ordered. Hashing provides a simple technique for distributing the data but is not as efficient for range queries which are better suited for ordered structures such as a sorted list. Our distributed skip list structure supports FIND, INSERT and DELETE of multi-dimensional key-value pairs. We also augment this with a range query operation that can take advantage of the ordering of the keys to output the result in a single operation. We consider range queries over multi-dimensional keys where one or more ranges can be specified. For example "4.[2,8].3.[1,3]" is a range query over four dimensions searching for records exactly matching the first and third dimensions in the range specified in the second and fourth dimensions.

There are several concurrent data structure implementations of skip lists built on top of a shared memory programing model using locks or lock-free techniques [3–5]. These implementations are targeted towards exploiting parallelism on multicore machines with support for memory consistency. However they do not easily scale outside a single machine and do not scale to larger clusters where communication is by message-passing and there is not the support for shared memory. A more comprehensive review of related work on skip lists is given in Section 4.

In the design of the skip list service we follow the same fine-grain methodology as that of a linked list. Initially, every key-value in the data structure is its own process and the operations are implemented using synchronous communication among processes making up the skip list. Later we will relax these conditions to both coarsen the implementation, by allowing processes to have multiple key-value pairs, and to allow asynchronous communication. As in our previous work, the design has the following advantages. First, it exposes a massive amount of concurrency that makes it easy to scale up or scale down. Second, because key-values belong to processes rather than tied to memory, it is easier to dynamically load-balance key-values among all of the cores on one or more machines. Third, there are numerous communication tuning parameters that can be introduced to match the implementation to the machine. Fourth, it makes easy to compose not only between OS-level processes but to compose in a fine-grain manner between function-level concurrency at the language level.

The service-oriented approach and fine-grain methodology needs an implementation of MPI that support the massive amounts of concurrency introduced in our design. As in [1], we implement our skip lists using FG-MPI [6, 7], a fine-grain implementation of MPI that extends the MPICH2 [8] middleware. FG-MPI makes it possible to have multiple MPI processes inside an OS process. It has its own integrated runtime scheduler, optimised communication between processes, and support for dynamically allocating processes. FG-MPI makes it possible to construct MPMD type of programs that scale to hundreds and thousands of processes across a cluster of multicore machines.

The focus of our work is on scalability with regards to the number of machines, irrespective of whether the distribution is needed for computation, memory, or the nature of the problem. Furthermore the focus of the design is on communication, rather than memory or file I/O. Assuming sufficient concurrency, communication cost is ultimately the limiting factor in the ability of a system to spread out across large numbers of machines. The ability to scale-up and scale-down is important to the design of systems to execute in diverse machine environments. Our distributed ordered dictionary system (i.e., skip list service) shows the design of a widely-used in-memory data structure (e.g., key-value store [9], database [10]) that can execute on single multicore machine or scale to large numbers of multicore machines.

This paper makes the following contributions:

- Provides further evidence of the practicability of a service-oriented approach to the design of data structures for MPI and the type of fine-grained methodology used in [1].
- Introduces a novel message-passing implementation of skip lists. Unlike concurrent skip list structures, the system is deterministic and in essence acts like a (leaky) pipeline with service requests flowing in and results flowing out. We introduce a range query operation that dynamically splits requests over the extent of the range, thereby parallelising the operation. We extend the operation to work with multiple key values per process.
- Shows that the skip list operations are atomic, and have a non-overtaking property that makes it possible to use shortcuts as an optimisation technique to trade off consistency semantics for performance. Unlike in a linked list, the shortcuts depend on the tower height and interact differently with the skip list structure.
- We achieved scalable performance on a medium size cluster of over 200 cores. Our skip list service can flexibly scale-up with added granularity and fine-grain concurrency on multicore and scale-out to take advantage of the parallelism in a cluster. Performance was the main focus of this work and an important part of the design was the variety of tuning parameters we introduced to adapt the skip list to the machine and the application. We can adjust the amount of asynchrony to better overlap communication with computation and overlap application requests with skip list operations. We can adjust the granularity to better hide communication delays and adjust tower heights, number of concurrent processes, and shortcuts to better trade-off parallelism with the expected number of operations.

In Section 1, we give a brief overview of skip lists and our design including a brief description of FG-MPI, the MPI middleware used in our implementation. In Section 2, we describe the implementation of the skip list operations and informally show deadlock-free, atomicity, and the non-overtaking property of the structure. Section 3 presents the experiments used for evaluating the system and related work and conclusions are discussed in Sections 4 and 5, respectively.

1. Overview

A skip list is an important dictionary data structure for storing and retrieving ordered data. Skip lists use probabilistic balancing which leads to simpler algorithms for insertion and deletion operations and an expected query time that is logarithmic to the length of the list [2].

As shown in Figure 1, a skip list can be viewed as multiple layers of linked-lists. The list at each layer, apart from the bottom one, is a sublist of the list below it. A node in a skip list is assigned a random height h and each node maintains a tower of h pointers to the next element in the linked-list at each of the layers. The number of nodes of a height h decreases exponentially with the value of h. Since there are fewer nodes of higher heights, a query operation is speeded up by skipping over a large number of nodes with shorter heights and moving down the layers until the search key is found, leading to an expected query time that is $O(\log n)$ for a list of length n. The search path of a query for key 38 is shown in Figure 1. Starting from the head node, the search path goes down the layers of its tower until it first encounters a successor node with a key less than or equal to 38. The search query is sent to this successor node where the same procedure repeats until key 38 is found.

We take a message-passing approach to the design of a skip list data structure. The starting point for the design is a "one data item per process" approach where we wrap a process

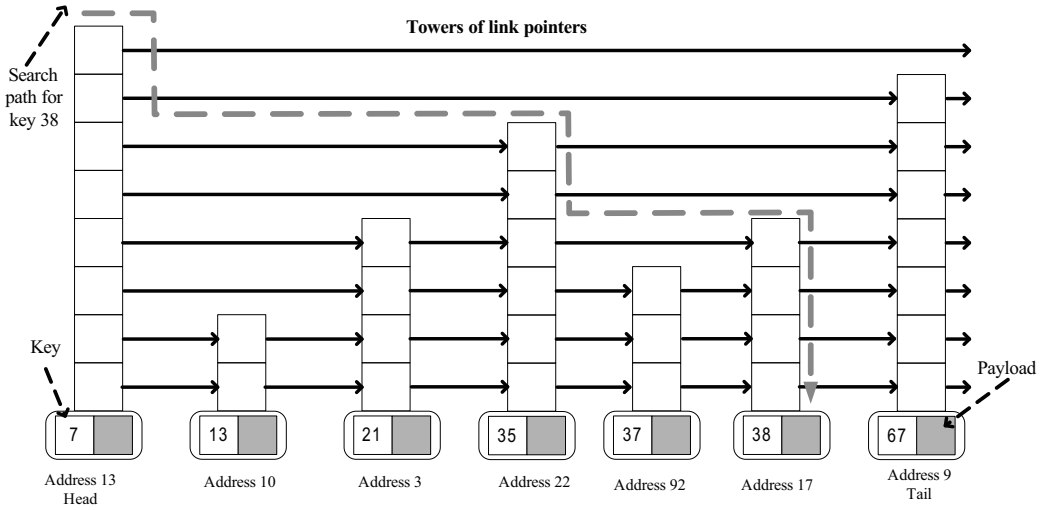


Figure 1. An example of a skip list data structure. Every node has a key-value, a payload and a tower of pointers to the successors in the linked-lists at different levels. The search path of a query for the key 38 is shown with thick dotted lines.

around every key-value pair. Later we extend the implementation to allow for multiple key-value pairs per process. An example of a small skip list is shown in Figure 2. As in a sequential implementation of a skip list [2], each node consists of a tower of pointers to other nodes along with the key-value pair that is stored at the node. Along level 0, the bottom level, we have an ordered linked-list and the list at level k is a subset of the list at level $k-1$. The root node in the skip list is configured to have the maximum tower height and there is a sentinel value used to denote the end of each level k list. As shown in Figure 2, each node in the skip list is an MPI process which is identified by its MPI process rank[1]. In our implementation MPI process rank is used in the same way a pointer to a memory address is used in the sequential implementation. Requests to perform an operation are messages that are passed from process to process along the list.

One difference between our process implementation and that of a sequential skip list is that at every level we store the minimum key of the next process for the level. For example, level 2 at process 0 stores the key 27, which is the key at process 12. The advantage to storing the minimum key at the predecessor is that requests do not need to query the next process to determine whether or not to forward the request. This significantly reduces the amount of messaging, but more importantly, it is now possible for requests to propagate asynchronously rather than the query-reply type of handshaking needed when the minimum value is not available. Although this adds to the number of keys we store, storing the minimum key is less significant once we allow for multiple keys per process.

This approach requires a system that can support large numbers of relatively small processes. Before further describing the design we discuss the main features of FG-MPI.

1.1. Fine-Grain MPI

MPI provides all of the basic message-passing routines needed for our design, however, it would not be practical with the current MPI middleware because most MPI implementations bind an MPI process to an OS process. As OS processes, they are too heavy-weight and it is not feasible with most systems to have the thousands and potentially millions of processes one might use in practice. This was a motivation for FG-MPI.

[1] All processes in an MPI program with N processes are assigned a rank from 0 to $N-1$. In MPI, process rank is used as the source and destination of messages.

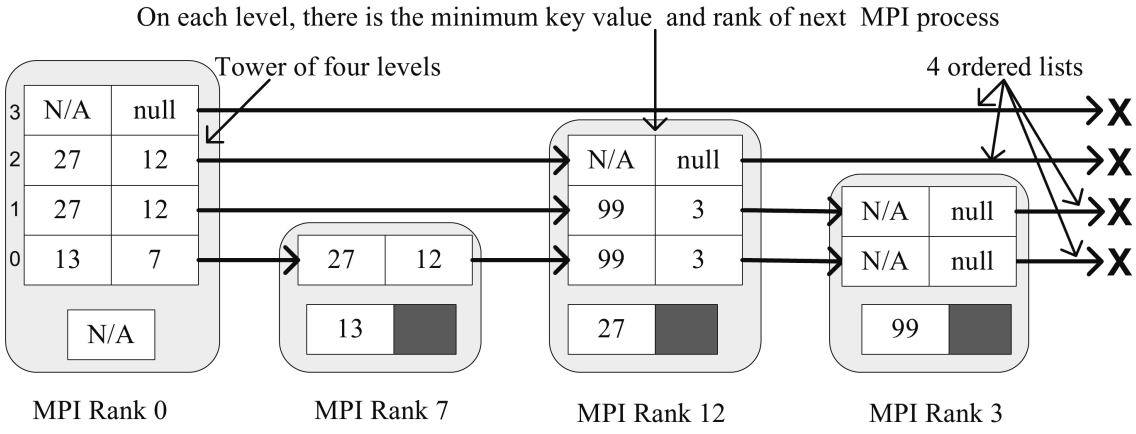


Figure 2. An example of a skip list where each list item is a process.

FG-MPI, which extends the MPICH2 middleware [8], decouples the notion of an MPI process from that of an OS process and makes it possible to have multiple MPI processes inside a single OS process. MPI processes inside an OS process execute concurrently in an interleaved manner as non-preemptive threads (coroutines). Because of the lightweight nature of coroutines it is possible to support thousands of processes inside a single OS process and millions of processes across cores and machines in a cluster environment.

An FG-MPI execution, $[P,O,M]$, can be described in terms of P, the number of MPI processes per OS process[2], O, the number of OS processes per machine and M, the number of machines. A typical MPI execution is of the form $[1,O,M]$ where N, the total number of MPI processes as given by the "-n" flag of MPI's `mpiexec` command, equals $O \times M$. In FG-MPI, a "-nfg" flag was added to `mpiexec` enabling one to specify $P > 1$, where $N = P \times O \times M$.

We take a service-oriented approach [1] to the design of the system where the skip list has a service interface that interacts with other processes through message passing. Each OS process is configured to have an application process, a manager process and one or more processes that are either free or a skip list process. Application processes send requests to the skip list service and receive back replies. The manager processes are part of the free process service and they service requests for the allocation and de-allocation of free processes.

At start-up, the skip list service consists of the skip list root process, application processes, and one manager process per OS process. All of the remaining processes are configured to be free processes. These free processes are all blocked on a receive call and FG-MPI's runtime scheduler [7] ensures that they remain on a blocked queue and do not add any overhead while blocked. Skip list processes make free nodes requests to the co-located manager process which cooperates with the other managers to find a free process which it can initialise by sending a message to turn the process into a skip list process. Unlike in a usual MPI environment, we can co-locate multiple processes together and take advantage of the FG-MPI scheduler to execute the co-located processes that can make progress. Effectively, by mapping P skip list processes to each OS process we increase the granularity of the OS process while still making it possible to interactively respond to requests. Normally this is not possible in MPI without resorting to threads and having the OS do the scheduling. Although the overhead in having P co-located processes is small [6], there will still be an advantage to having a coarser-grain implementation with MPI processes managing more than one key-value pair.

[2]We refer to these P MPI processes as *co-located* processes sharing a single address space. The terms *process*, *fine-grain process* and *MPI process* are used interchangeably. The term "OS process" is used to refer to an operating-system process.

1.2. Asynchronous Communication

MPI supports both synchronous and asynchronous communication. The standard send for small messages eagerly forwards messages to the destination and assumes the message can be buffered at the destination. There is no flow-control mechanism in MPI for small messages and it is possible to exhaust the buffer space [11]. In the skip list system we use a bounded buffer technique where we can send up to k eager sends before requiring a synchronous send (`MPI_Ssend`). The value of k is a parameter that can be set or adjusted automatically during execution [1]. The flow of messages in the skip list is deterministic and the value of k, assuming sufficient buffering, does not affect correctness. In the experiments in Section 3 we maximise this value, however, for the description of the system it will be easier to consider $k = 0$, the synchronous case.

In FG-MPI the receive queues for all of the co-located processes are shared and when an MPI routine is invoked by one process, the middleware progresses messages for all co-located processes potentially rescheduling them for execution. A consequence of the large amount of concurrency is there can be many more eager messages, which can add to the queueing delays for messages and extra work for the middleware. However, by having lots of processes and lots of small messages there is more fluidity in the flow of requests through the system. Increasing the degree of asynchrony makes it possible to have more active requests in the system and, for remote communication, overlap communication with computation, thereby keeping the OS process for each core busier.

1.3. Service Interface to the Skip List

We compose our skip list processes with application processes by allowing there to be one or more application process inside each OS process. The mapping of skip list and application processes is done at system start-up when functions are bound to MPI processes during FG-MPI initialisation.

Our skip list design builds on our previous work that investigated a similar interface to an ordered linked list service [1]. As in the linked list service, the application processes access the service by sending requests to a skip list process. Skip list processes reply directly back to the application process making the initial request. There may be additional communication as in the case of INSERT for large data values, where the data is retrieved from the application only after its position in the skip list is determined.

Three types of consistency semantics are supported according to how an application sends requests and receives back replies. We allow an application process to have a fixed number of outstanding requests. Because the process at the location of an item is the one that replies, replies are not guaranteed to come back in the order of the operations; replies by processes towards the end of the list are likely to return after a later request for a skip list item near the head of the skip list. Sequence numbers are used to re-order replies. The sequence number is carried along as part of the request and returned to the process inside the reply message. We use MPI's non-blocking receive command (`MPI_Irecv`) to pre-post receive buffers for the replies for each outstanding request. The pre-posted receive buffers act as a hold-back queue [12] which is managed by the application and can be used to re-order and return replies in the order of the operations.

All application processes know the rank of the head process and can send requests to the head. We show in Section 2.4 that requests are not able to overtake one another and thus requests are serviced according to the order they arrive at the head process. This totally orders the operations and is a linearisation point. However, the head process is an obvious bottleneck and we implemented *shortcuts*; a technique whereby requests can be sent to other skip list processes.

Shortcuts take advantage of the single address space of the co-located skip list processes to create a bulletin board type service where skip list processes can post keys and ranks for the application process to consider as alternative skip list locations to send the request. For example, if an application process is performing a `FIND(key x)`, it can look-up the key on the bulletin board to find the largest key smaller than `x`, and use the associated rank to jump into the skip list at that location. The non-overtaking property still holds once the request has been received by a skip list process.

Shortcuts remove the head as the bottleneck but at the expense of linearisability. However, with the help of sequence numbers, the application process can maintain sequential consistency by judicially using shortcuts only when they do not violate its own ordering. When consistency is not required, then both the application processes and/or skip list list processes can be configured to make full use of shortcuts. Consistency is with respect to the operations themselves and not the consistency of the dictionary structure as a whole. Also, irrespective of the consistency semantics, the list always remains properly ordered and the skip list operations themselves are correct.
The three semantics are:

Total Order :
All application processes send requests to the head.

Sequential Consistency :
Application processes judicially use shortcuts and a hold-back queue to ensure that the operations issued by the process are completed in order they were issued.

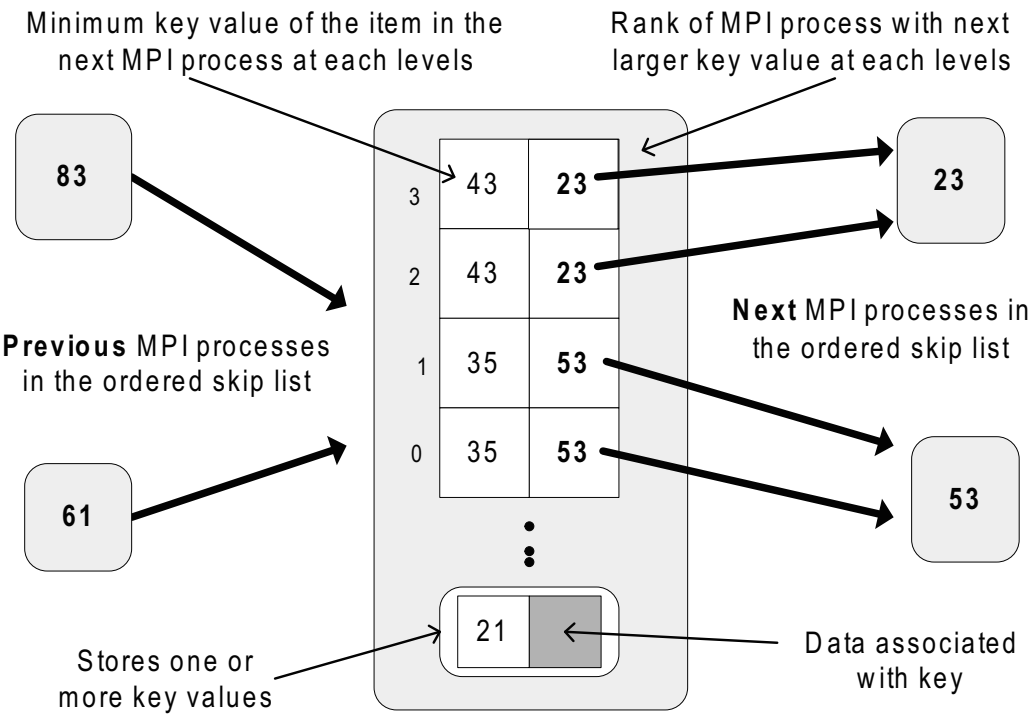
No Consistency :
Both the application processes and the skip list processes use shortcuts when possible and there is no guarantee about order.

In summary, these semantics are dependent on the following properties of our skip list implementation. First, the operations themselves must be atomic where one operation cannot interfere or corrupt another operation. Second, operations cannot jump over one another, even when shortcuts are used. Proof of these properties is discussed in Section 2.4. Section 3.1 compares the performance of the skip list with respect to these different semantics.

2. Skip list Design

As shown in Figure 3(a) each skip list process (i.e., node) is comprised of (i) a *tower* of `<key,rank>` pairs where each key is the minimum key (`min-key`) of the skip list process with that rank, (ii) key (or list of keys in the multi-key case), and (iii) the associated data. The height of the tower is probabilistically determined when a process joins the skip list and goes from level 0 to some level k less than a predefined maximum tower height. As shown in Figure 2 the skip list `ranks` on level i are a sublist of level $i-1$ and the `rank` on a level points to the next node in the list.

Figure 3(b) shows the message-driven structure of the process where processes block waiting for a request. Processes do not accept another request until it has completed sending all messages needed to locally complete the operation. Request messages contain the key information along with the process rank of the application node making the request. Request messages may alter nodes (INSERT, DELETE) and can mutate (RANGE-QUERY) as they traverse the skip list. Sections 2.1 and 2.2 describe the implementation of each of the four operations.



(a)

```
while (skiplist process) {
    receive (request r_i) from ANYPROCESS;
    switch (R) {
        FIND        : do FIND(r_i,level);
        INSERT      : do INSERT(r_i,process,level);
        DELETE      : do DELETE(r_i,level);
        RANGE-QUERY : do RANGE_QUERY(r_i,level);
        EXIT        : do TERMINATE();
    }
}
```

(b)

Figure 3. (a) Example of a skip list process with connections from previous and next processes, and (b) Message-driven main loop of the process.

2.1. FIND and RANGE-QUERY

The FIND operation is shown in Listing 1 where the process either replies to the client (Steps 3–6), forwards the request, or drops down to the next lower list. Since the minimum key (`min_key`) of the successors is stored at each node, it generates either one reply or one request.

The RANGE-QUERY routine (Listing 2) differs from FIND in that, when necessary, it splits the request (Steps 11–13). Every skip list process with data within the range specified in the query will reply to the application. In the multi-key per node case, the number of returned key-value pairs depends on the distribution of the data and may vary from one to the number of keys in the range. Each split partitions the range and each new request records the start and end of its part. The application re-assembles the parts and the query is complete when the application detects that the original range is complete. By design, skip lists adapt to the distribution of the data and similarly our range queries will split more in dense areas of the key space.

Listing 1: FIND Operation

```
1 FIND(request, level)
2 {
3     if ( level==0 &&  request_key < key[0] )
4         reply failure;
5     else if ( level==0 && request_key == local_key )
6         reply with data
7     else if ( request_key > key[level] )
8         forward request;
9     else
10        FIND(request,level-1);
11 }
```

Listing 2: RANGE-QUERY Operation

```
1 RANGE-QUERY(request, level)
2 {
3     // local_key: local process key-value
4     if ( level==0 &&  (key[0] > end_of_range) )
5         reply end_of_range;
6     else if ( level==0 && request_key in range )
7         reply with data
8     else if ( start_of_range > key[level] )
9         forward request;
10    else if ( (local_key && key[level]) in range )
11        split range into two parts;
12        forward last part of request;
13        RANGE-QUERY(first part of request,level-1);
14 }
```

2.2. INSERT and DELETE

The process to be inserted or deleted from the skip list needs to either link or unlink itself from the skip list as it traverses the list. The basic link and unlink operations are shown in Figure 4. For INSERT, given a tower of size k, as we traverse the list we need to fill in the levels of the tower when inserting it between two towers. For example, in Figure 4, process C is to be inserted at a location between A and B which requires A to send $[y,B]$ to C. For DELETE, the reverse occurs and we need to unlink the node to be deleted, C, from the list which will require A to send a message to C for its information about B. Unlike FIND and RANGE-QUERY, these operations make changes to the list while traversing the list.

The INSERT operation is shown in Listing 3. The operation differs in that there is a "new process" associated with the INSERT request. The height of the new process is immediately determined but we do not acquire the process until the first time it needs to link itself to the skip list. A process acquires a new free process by requesting one from its local manager, which finds the free process, returns its rank to the first process that then sends a message to initialise it with the new key and height.

We implemented a simple manager policy where the managers are arranged in a ring and requests that cannot be satisfied locally are forwarded to the next manager. If, after one cycle around the ring, no free process is found then the requesting process replies that the list is full (`list_full`) back to the application.

Listing 3: INSERT Operation

```
1 INSERT(request, newprocess, level)
2 {
3     if ( level <= request_height && request_key < key[level] )
4         link[key,level,newprocess];
5         if( level==0 )
6             reply success;
7         else
8             INSERT(request,newprocess,level-1)
9     else if( request_key == key[level] || request_key == local_key )
10        reply duplicate
11    else if( request_key > key[level] )
12        forward request;
13    else
14        INSERT(request,newprocess,level-1)
15 }
```

If duplicate entries are not allowed in the skip list, then an INSERT operation may fail due to the presence of a duplicate key. There are two cases to consider, which depend on the tower height of the new tower in comparison to the tower height of the existing duplicate node. First, if the height of the new tower is smaller or equal to the height of the duplicate we report failure when we first encounter the duplicate's tower, thereby leaving the list unchanged. Second, if the height of the new tower is larger than the height of the duplicate, the operation proceeds as usual linking in the higher levels. However, once it encounters the duplicate the new node sends unlink messages to the levels of the duplicate as they became visible. Eventually, once the new node is directly before the duplicate it initiates a DELETE, thereby deleting the duplicate, and again reports failure back to the application. One side effect of this is that duplicate operations will tend to raise tower heights, which affects the probabilistic guarantees when tower height is chosen by a random Bernoulli process.

The DELETE operation is shown in Listing 4. This operation does the reverse of the insert operation. It is also the only operation that needs to perform an `unlink` operation which requires a process to send back information to a predecessor in the list.

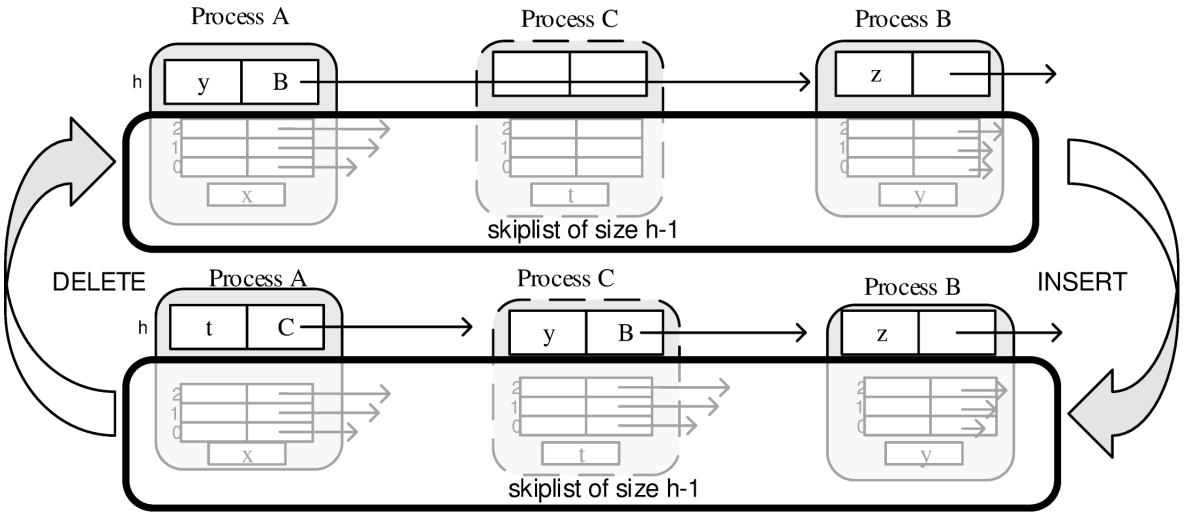


Figure 4. From top to bottom the figure shows process C being linked or unlinked at Level *h* of the skip list which is required for INSERT and DELETE, respectively.

Listing 4: DELETE Operation

```
1 DELETE(request, level)
2 {
3     if ( level==0 &&  request_key < key[0] )
4         reply failure;
5     else if ( request_key == key[level] )
6         unlink[key,level];
7         if( level==0 )
8             reply success;
9         else
10            DELETE(request,level-1)
11    else if( request_key > key[level] )
12        forward request;
13    else
14        DELETE(request,level-1)
15 }
```

2.3. Freedom from Deadlock

Ensuring correctness is a major challenge since a skip list may easily consist of hundreds and thousands of processes. The freedom from deadlock inside the skip list follows from the following two properties:

(a) requests are handled atomically, and
(b) requests are always in the forward direction.

All processes start by receiving a request and each of the listings end by either making a new request to a forward node and/or replying back to the application. It is possible for two processes to communicate back and forth between themselves while performing a request (e.g. `(un)link`), but initial requests occur only in the forward direction.

2.4. Order of Operations

The deterministic nature of our implementation makes it possible to reason about the order of the operations. In this section we will show that operations cannot overtake one another inside the list. This condition, together with shortcuts, make it possible to implement the total order, sequential consistency, and no consistency semantics described in Section 1.3.

We will describe an operation R as a sequence of request messages which hop along the list with each request updating the local information and generating a potentially new request to another process. Let R equal to $r_1, r_2, \ldots r_k$ be the list of request messages received by the processes.

As given by the listings in the previous section, requests are atomic and all messages between processes can be realised using synchronous communication. One subtlety that arises with MPI communication is that messages need to remain ordered when we extend it to asynchronous messaging. In the link and unlink operations, where the process sending messages on level l (the level l source process) can change, we use a synchronous send (`MPI_Ssend`) to send the *last* message from that source. This ensures that all buffered messages are flushed before a process can receive a message from a new level l source process. As a result, when the source process on a level changes, the messages from these two separate sources are not re-ordered by the MPI middleware. Therefore requests remain ordered and it remains only to show the operations themselves remain ordered. We show this by arguing inductively over the height of towers starting with the ordered list on level zero.

Consider a list of processes where process i can only send messages to process $i+1$, the next process in the list. Suppose each process stores a key-value pair such that if $i < j$ then the key at P_i is less than the key at P_j. Thus the list of processes is essentially the distributed counterpart to an ordered linked list data structure. In the ordered list there are FIND, INSERT, DELETE and RANGE-QUERY operations and furthermore assume operation requests are sent to the process at the head of the list and all communication is synchronous.

Given two operations R^a and R^b, and two requests $r_i^a \in R^a$ and $r_j^b \in R^b$, let $[r_i^a \rhd r_j^b]@P$ denote that both r_i^a and r_j^b are executed by process P, and receive r_i^a executes after receive r_j^b in process P. Furthermore let $\mathscr{R}$ be a multiset of operations and $\mathscr{P}$ be a process list such that for all for all requests r in an operation R of $\mathscr{R}$, r is executed by a process from $\mathscr{P}$.

Definition 2.1 *For all r in some R of $\mathscr{R}$ on process list $\mathscr{P}$, define $\succ$ as:*

$$r_i \succ r_j \iff \begin{cases} r_i, r_j \in R & \text{for some } R \text{ and } i > j \\ [r_i \rhd r_j]@P & \text{for some } P \in \mathscr{P} \end{cases}$$

This is causality relation, capturing the notion of time, where either the receive r_i happens after r_j in a given operation or r_i happens after r_j at a given process P. Causality relations are irreflexive partial orders [13]. By design, operation requests are synchronous, atomic and only occur in the forward direction. Fix a multiset of operations $\mathscr{R}$ and a suitable set of processes $\mathscr{P}$. Unless otherwise stated, all occurrences of $\succ$ use this $\mathscr{R}$ and $\mathscr{P}$.

The purpose of the following lemma is to show that in a simple list one operation cannot overtake a second operation that is ahead of it in the list. Given that there can be many operations distributedly finding, adding and deleting nodes at the same time we provide a formal proof of the statement using the space/time model and terminology from Charron-Bost, Mattern and Tel [14]. A "crown" is a crossing that is a forbidden structure in a time/space diagram for a realistic synchronous computation (i.e., a computation realisable by synchronous messaging). We show that overtaking induces a crown, contradicting the fact that the messaging is synchronous.

Lemma 2.2 *Let R^a and R^b, be two operations on process list P. If $[r_i^a \rhd r_j^b]@P$ where $r_i^a \in R^a$ and $r_j^b \in R^b$, then for all k, such that r_{i+k}^a, r_{j+k}^b exist, $r_{i+k}^a \succ r_{j+k}^b$.*

Proof Assume to the contrary, take the first occurrence such that $r_{i+k}^a \succ r_{j+k}^b$ but $r_{j+k+1}^b \succ r_{i+k+1}^a$. Given that we have a linear process list where process i can only send to process $i+1$, the next request can only be to the next process in the list. By assumption r_i^a and r_j^b both occur in process P, it follows from the previous statement that r_{i+k}^a and r_{j+k}^b occur in the same process. As well, r_{i+k+1}^a and r_{j+k+1}^b both must occur in the next process in the list. The relations $r_{i+k}^a \succ r_{j+k}^b$ and $r_{j+k+1}^b \succ r_{i+k+1}^a$ form a crown, which implies that the messaging cannot be realisable as a synchronous communication, which contradicts that the communication is synchronous.

Lemma 2.2 captures the property that operations in the list are performed by processes in order and cannot jump ahead of one another. This property is also useful for reasoning about shortcuts (Section 2.5) where operations may start at processes other than the head. It follows from Lemma 2.2 that irrespective of where the operation starts, when an operation is behind another operation, the second operation remains behind the first one. In a skip list we have the following theorem.

Theorem 2.3 *Given a collection of operations on process list P. If $[r_i^a \rhd r_j^b]@P$, then either (a) $[r_{i+1}^a \rhd r_{j+1}^b]@P$ or (b) for all $m,n > 1$, r_{i+m}^a and r_{j+n}^b do not visit the same process.*

Proof Skip lists have the following two nice recursive properties. A skip list of height h can be viewed as a skip list of height $h-1$ with the addition of an ordered list, which is a sub list of the list at height $h-1$ in the lower skip list. The collection of skip list processes between two processes P_i and P_j also form a sub skip list.

The proof is by induction on the maximum height of towers in the skip list. A skip list with maximum height of one is an ordered list and the result follows from Lemma 2.2. Assume true for a skip list of maximum height $h-1$ and now consider a skip list of height h. Consider two operations r_i^a and r_j^b at some process P and furthermore assume that as part of the execution they both examine level h to determine whether one or both of r_i^a and r_j^b send a request to the next process (i.e. hop forward) at level h or decrement the level and traverse the lower skip list. If both r_i^a and r_j^b traverse the lower skip list, then by induction the hypothesis holds. Now consider the case when both hop forward to the next process at level h. Since the list at level h is an ordered list, it follows that $[r_{i+1}^a \rhd r_{j+1}^b]@P$ by Lemma 2.2. Finally, in the case when one descends into the lower skip list and the second request hops ahead on level h, this can occur only because the target key for one operation is greater than the key stored at level h at P while the target key for the second operation is less than key stored at the level. As a result, for all $m,n>1$, r_{i+m}^a and r_{j+n}^b never visit the same process because the remaining requests in the operation can only reside in the processes between the current process and processes before the one pointed to by level h, whereas the requests for the operation hopping forward will be to processes equal to or beyond the process specified in level h.

In summary we have (a) operations are atomic and cannot overtake one another, (b) once unlinked, requests behind a delete process can proceed and hop ahead, (c) for insert, there is a delay since we do not allow it to accept requests until its tower is complete (inserts are the most expensive operation in terms of its potential to delay other requests), and (d) the skip list remains deadlock free since requests are only sent in the forward direction.

2.5. Shortcuts

As a result of Theorem 2.3, which shows that operations do not overtake one another, it follows that if the application sends all requests to the head, then the operations are totally ordered according to the order they are received by the head node. Again, as was the case in the single linked list, the head can become a bottleneck. As in [1], we indroduce a shortcut mechanism whereby application processes can take a shortcut and join the skip list closer to the key's target location.

In implementing shortcuts we depart from pure message passing and take advantage of the properties of co-located processes in FG-MPI. In FG-MPI, all co-located skip list processes share the same address space and because processes are scheduled non-premptively it is easy to atomically coordinate access to a shared bulletin board structure for skip list processes to query for shortcuts.

Inside each OS process we maintain a local bulletin board structure. Skip list nodes post their keys, rank and tower height to the bulletin board. All co-located application processes can read from the bulletin board as a potential hint for faster access to the skip list. The shortcut is only a hint since the node pointed to by the shortcut may have been freed or may have been freed and reallocated. In the case of a free process, the free process sends a failure reply when contacted by the application. In the case the node had been reallocated, then we either detect a larger key and fail or detect a smaller key and follow the shortcut. On failure, we notify the application process, which can send the request to the head to ensure it will succeed.

Unlike the simple linked list, there are other ways in which shortcuts can fail. For an INSERT or DELETE the tower height of the shortcut needs to be higher than the tower to

be inserted or deleted. Otherwise it is not possible to update the towers between nodes that have connections on levels above the shortcut. We can detect these situations report failure whenever a shortcut leads to a tower of greater height than that of the tower to be inserted or deleted.

Rather than report failure as soon as we encounter a larger tower, we could allow the process to climb to a higher level and continue. This violates the condition of Theorem 2.3 and introduces a race condition between the operation taking the shortcut and other operations. However, the first operation to reach the tower, wins and once ordered remain ordered for the rest of the operation.

By using shortcuts, sequence numbers, and a hold-back queue the application has the mechanisms needed to implement a variety of consistency semantics depending on the application purpose. Sequential consistency can be maintained using shortcuts without climbing and finally when climbing is permitted there are additional opportunities for using shortcuts but there is no longer any consistency guarantees. When consistency is not required we can extend the use of shortcuts to allow the skip list nodes themselves to use shortcuts.

2.6. Multiple Key per Process

In this section we describe the extensions to FIND, RANGE-QUERY, INSERT and DELETE operations to handle the case when a list process may store multiple key-value pairs. We call the maximum number of items that a skip list process can hold the *granularity*, G, of the skip list service. Both G and P, the number of co-located MPI processes, are helpful in reducing the idle time in the service. Increasing G reduces the amount of messaging and therefore the amount of work done in performing requests. However, to maximise the opportunity for parallelism, in a $[P,O,M]$ execution we need to distribute the skip list to the $O \times M$ processing cores.

The extension to the FIND operation is straightforward. The local set of keys are searched before forwarding the FIND request to a neighbour process. The RANGE-QUERY being an extended form of the FIND operation, which is split as it traverses the list, uses the same technique and searches the local set of keys prior to forwarding the request. A consequence of increasing the granularity is that an application process may receive variable sized replies from list processes in response to a RANGE-QUERY. To allow the application process to properly assemble the replies, two messages are sent to it by the list processes; the first containing the size of the result and the second the result itself.

The extension to the INSERT operation, however, is more interesting and required the addition of a new SPLIT operation. For INSERT, since each node now stores a contiguous range of the keys, there is a choice between adding the key to an existing node (see Figure 5) or creating a new node (i.e. process) to store the key. Obviously, we cannot continuously insert keys into existing nodes and there has to be a mechanism to balance the load and to split an overloaded node. To this end we define a threshold value G, the granularity, whereby we split a node once the number of items in the node exceeds G.

There are two cases to consider for splitting an overloaded node (see Figures 6(b) and 6(c)). In both cases we determine a tower height for the new node to be inserted. If the tower height is less than or equal to the tower height of the node to be split, then we obtain a new node who will insert itself in the list directly after the existing node. The existing node has all the information and data needed to properly initialise and split the load between the two nodes. And, as in the other operations, communication is in the forward direction.

If the tower height is greater than the height of the new to be inserted, then the new node needs to be inserted directly ahead of the existing node in the list, however, this violates our condition for operations to traverse the list in the forward direction. Note as well, this case is essential because it is the mechanism whereby nodes with towers of higher heights

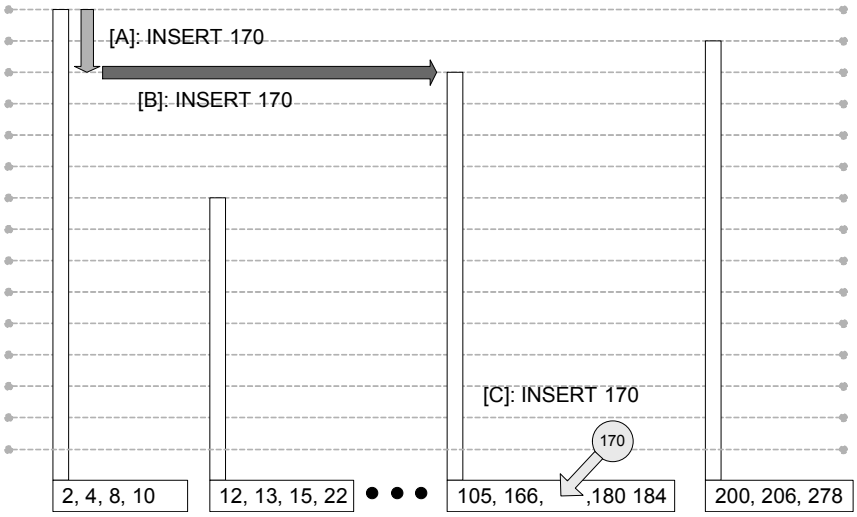


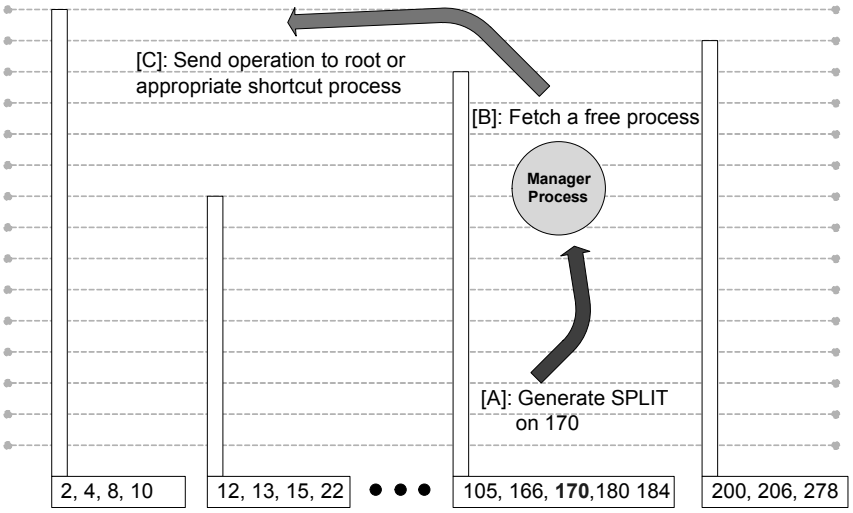
Figure 5. Steps in the INSERT operation for multiple key per process implementation. The operation travels like a FIND request, and inserts the entry locally.

are created. In this case, we go ahead and INSERT the item locally but in our reply to the application we require the application to perform a SPLIT operation (see Figure 6(a)), noting as well that the INSERT succeeded. The list node also marks itself as having a "pending split" and can begin processing the next operation. A marked node operates as usual, except that it only inserts items locally.

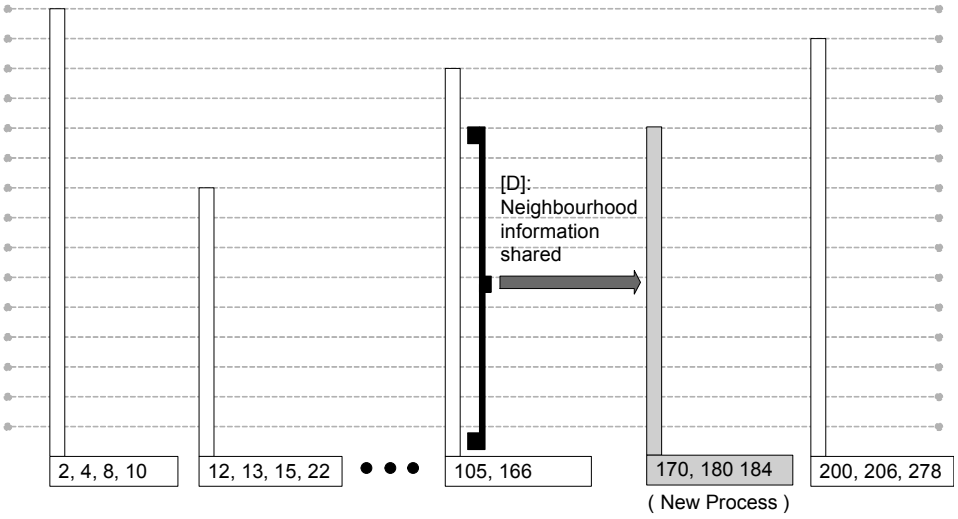
The SPLIT operation uses the `min-key` in the marked node to traverse the skip list inserting itself as described in the INSERT operation for the single key case. Once the operation has traversed the skip list to the location directly ahead of the marked node, the node obtains a free node and the free node and marked node exchange information and data in a similar manner as in the first case. After balancing the load the marked node returns to its unmarked state. However, it is possible that by time the SPLIT operation reaches the marked node the node has been deleted. In this case the new node turns into a zombie process that continues to forward operations but automatically unlinks itself as requests come in from the levels. Eventually once it has removed all of its tower connections it becomes a free process. We chose a lazy removal of zombie processes rather using an operation to actively remove them from the skip list. The SPLIT is an optimisation to dissipate the load and it is possible to introduce different strategies with regards to the load and when to generate a SPLIT operation.

In the case of DELETE, the operation can remove the key from the local process as long as the key to be deleted does not match `min-key` (see Figure 7(a)). When the delete matches `min-key`, then the first time DELETE encounters a pointer to the tower with that `min-key` it needs to wait until the entire operation completes at the target node (see Figure 7(b)). Once the DELETE reaches the target node, there are two cases. If the node is now empty, then the node is unlinked from the list, the `min-key` of the next node on each of the levels is sent back to nodes waiting for the DELETE to finish, and node is freed. If the node is not empty, then we simply send back the new `min-key` value to the waiting nodes. In comparison to the single item per node case, a delete to non `min-key` is faster since unlinking is not necessary, however, `min-key` deletes introduce an additional delay in the pipeline as it has to wait for the operation to complete. This delay is necessary because otherwise it is possible that inserts ahead of the delete may change the key that will become the next `min-key`. In which case the `min-key` value no longer remains consistent from the time delete first encounters the node until eventually reaching the node.

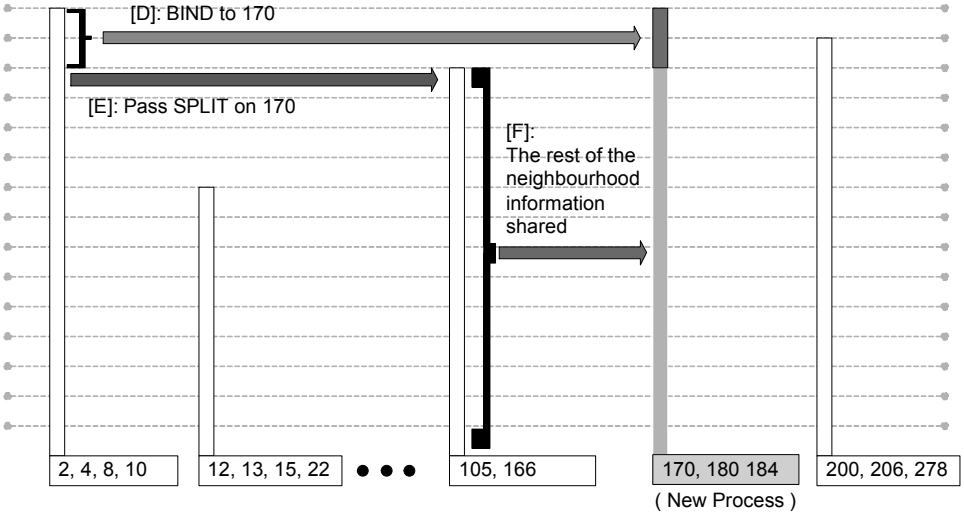
Unlike in the simple linked list, extending the implementation from the single item per process to multiple items per process required several changes. Some changes, like delaying



(a) Generate a SPLIT operation.

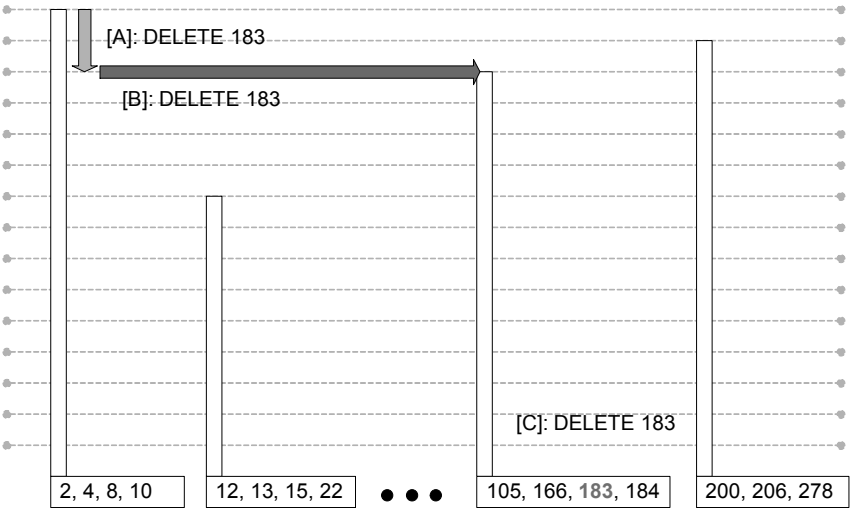


(b) SPLIT when shorter tower.

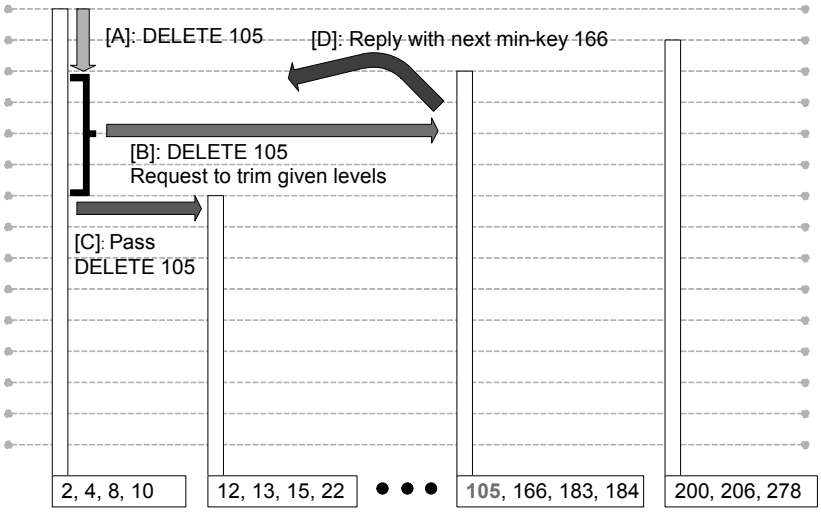


(c) SPLIT when taller tower.

Figure 6. Steps in the SPLIT operation for the multiple key per process implementation. The steps taken is comparable to the INSERT operation of the single key per process implementation.



(a) Delete inside the list of a process.



(b) Delete the min-key of a process.

Figure 7. Steps in the DELETE operation for the multiple key per process implementation. All the reply messages are held back until the whole DELETE operation is received complete.

the DELETE, were necessary to maintain consistency needed for correctness. Other changes, like the addition of the SPLIT operation, were needed to enable parallelism by ensuring new processes were added as needed. Understanding the potential interactions of operations with the processes was complicated, but made simpler by rigorously adhering to two main principles (a) operations only initiated communication in the forward direction, and (b) minimise shared state.

3. Experiments

In this section we evaluate the performance of our skip list with respect to different consistency semantics and various workloads. We compose each OS process with one application process, one manager process and one or more skip list processes. The application processes generate requests that are serviced by the skip list. For the experiments we vary the list size from over a million key-value pairs to 20 million key-value pairs. INSERT operations are used to randomly insert items until the list is full. Once the list is full we generate requests as specified for the workload and measure the system throughput with respect to operations per second. The measurements reported in the experiments is the median value over 11 ex-

periments. We designed the mix of operations in our workloads to be similar to those used in YCSB (Yahoo Cloud Serving Benchmark) [15]. Most of the workloads we used showed similar trend in scaling behaviour. We report performance of read-heavy workloads (i.e. FIND and RANGE-QUERY) as well as workloads with a mix of FIND, INSERT and DELETE operations.

For the experiments, the test setup consisted of a cluster of 25 machines connected by a 10 GigE Ethernet interconnection network. Each of the machines in the cluster is a quad-core, dual socket (8 cores per machine) Intel Xeon® X5550, 64-bit machine, running at 2.67 GHz. All machines have 12 GB of memory and run Linux kernel 2.6.18-308.16.1.el5. All executions in the experiments were of the form $[P = 100, O = 8, M]$ with M ranging from 2 to 25. Through experimentation we found that $P = 100$ MPI processes per OS process provided a good balance between the benefits of the added concurrency and the overheads of messaging and scheduling. We set $O = 8$, the number of cores per machine, so as to minimise the effect of the operating system on the execution. We start at $M = 2$ to ensure all of our tests include TCP network traffic. The list size is given by $G \times P \times O \times M$ where G is the granularity. A fixed value of $G = 1000$, $P = 100$ and $O = 8$ was used in all the experiments and M was varied from 2 to 25.

There are two variables W and k that can be used to tailor the system to the characteristics of the machine. Variable W, application window size, is the maximum number of outstanding requests that can be submitted by an application process. Increasing W increases the load on the skip list service. Application processes post receive-buffers for every outstanding request, and re-posts requests as soon as it receives a complete reply. The skip list service can be viewed as a large pipe with all the individual application processes contributing to the overall throughput. As latency increases, either because the list has increased or more processes have been added, we increase W to allow there to be more outstanding operations. However, we do not allow it to capture more than its share of the throughput, since it can lead to hot-spots and a decrease in the overall system throughput. Each application process continuously measures the reply latency for operations and adjusts W accordingly. k, the degree of asynchrony, is a throttling parameter that each list node can use to specify how many requests are to be forwarded down the list, before having to wait for a previously forwarded message to complete. The smaller the value of k, the higher the throttling effect on the flow of requests through the list service. There is a limit to increasing k, which depends on the flow capacity of the pipeline. Increasing k beyond this value has no effect since the service is already working at full throttle. We set k to forward requests down the list at full capacity.

3.1. Consistency Semantics

Figure 8 shows the scaling behaviour for the three different consistency semantics supported in our skip list implementation: total-ordering, sequential-consistency, and no-consistency. Along the x-axis, the list size increases from 1.6×10^6 key-values to 20×10^6 key-values. The number of OS processes is kept equal to the number of cores. Since we have one application process per OS process, the load on the skip list service increases as we move along the x-axis. The sequential-consistency and the no-consistency semantics show good scaling behaviour as the number of cores increases. As expected, the performance of the no-consistency semantics is the highest, followed by sequential-consistency and then by the total-ordering semantics. The total ordering curve saturates after a list size of 8 million key-value pairs due to the bottleneck at the head node. For each added OS process (i.e. core) we add an application process and we allow each of the application processes to have at most $W = P = 100$ outstanding requests. This allows there to be at most one outstanding request for every skip list process and, since the out-flow (reply rate) directly determines the in-flow (request rate) there is little benefit in having more than one application process per OS process.

As Figure 8 shows, once requests can use shortcuts, replies can return faster and there is less contention at the head and as a result throughput increases several fold. The performance difference between the no-consistency and the sequential-consistency throughput is small because the tower-based structure of the skip list has built-in shortcuts which aid in the sequential consistency case and the advantage due to no-consistency is less pronounced. The performance difference between no-consistency and sequential-consistency would increase if we reduce the maximum tower height in our skip list. The difference between sequential consistency and no-consistency is more evident in a linked-list, which we discussed in prior work [1].

3.2. Range Queries

Figure 9 shows the scaling behaviour of range queries for three different sized range queries. The largest size query returns 1% of the list, which is a large portion of the skip list and is an extreme type of query. Recall that the range-query is automatically split as it traverses the list and each process with any portion of the result replies to the application. Therefore as the number of cores increases we go from having fewer and larger replies (i.e. returned messages) to having more and smaller replies. Figure 9 reports throughput with respect to (a) the number of operations, (b) the number of returned messages, and normalises the performance across query types by giving (c) the number of returned key-values per second.

As expected, as the size of the range of a query increases, the operations per second that we can send to the skip list decreases. This is because application processes have to wait longer for all replies to return before it can send a new range query. However, as we can see in Figure 9(c), for larger queries we can return more key-values from the service, which is because a single range query operation sweeps through the service generating key-value results as opposed to having multiple separate FIND queries for each result. Figure 9(b) shows the number of returned messages all the application processes receive per second. All three range query sizes do scale with respect to the number of returned messages they can receive.

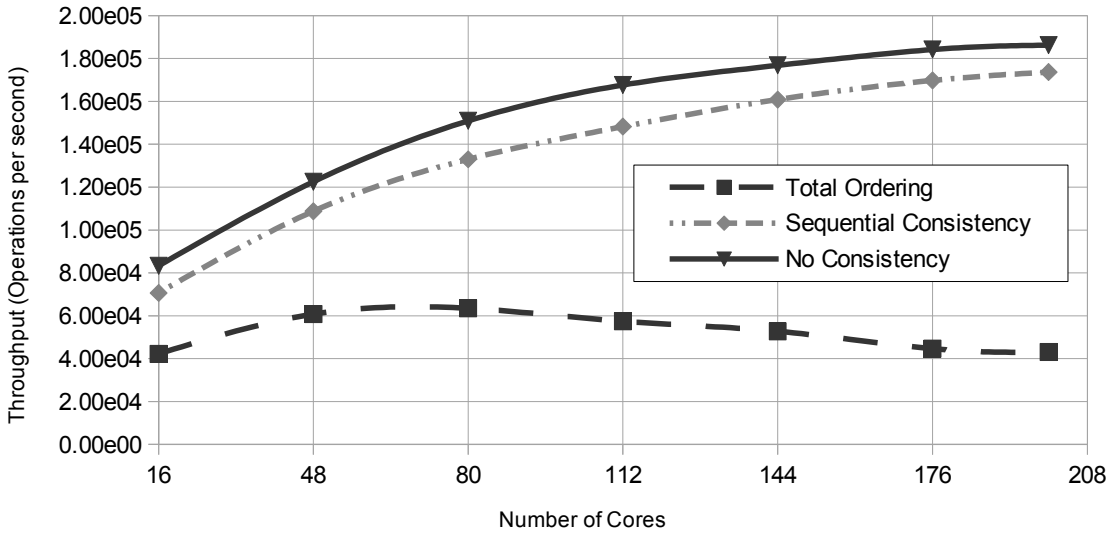
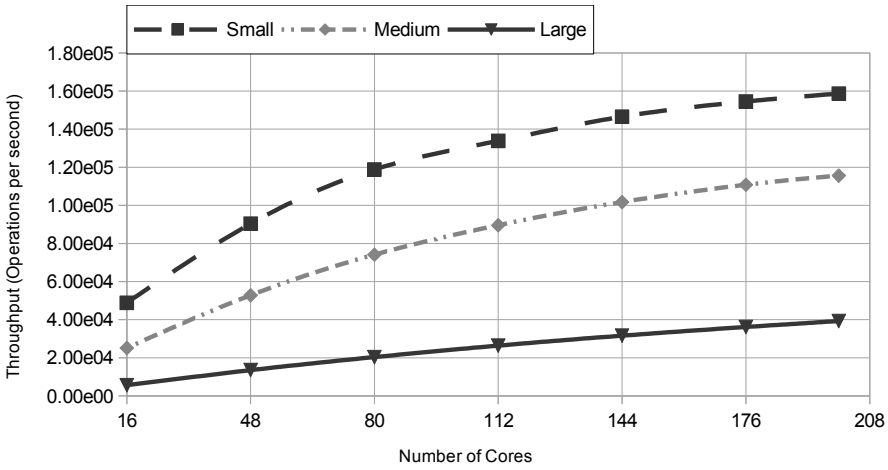
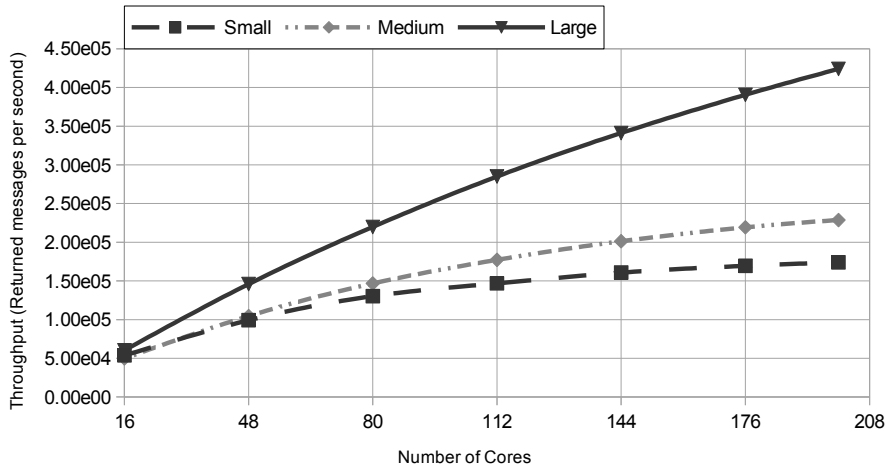


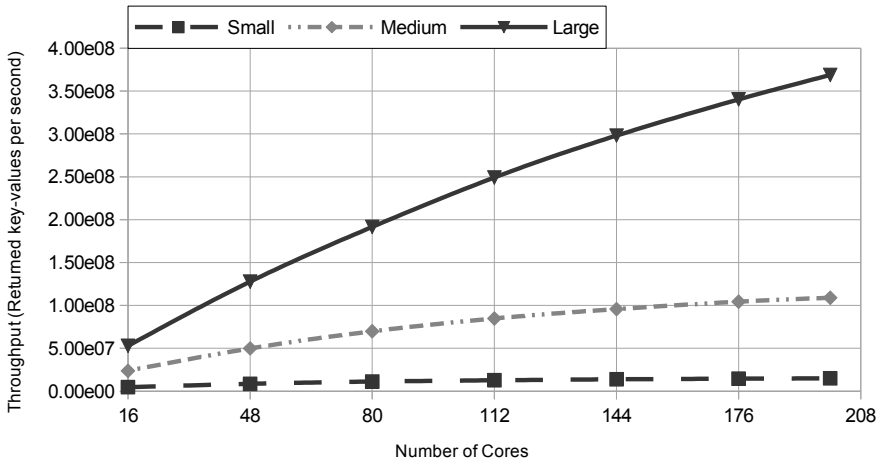
Figure 8. The number of operations per second versus the number of cores for the three different consistency semantics, total-ordering, sequential-consistency, and no-consistency. Workload uses 100% FIND operations. Configuration: List size = $G \times P \times O \times M$ where $G = 1000$, $P = 100$ and the total number of cores=$O \times M$ range from 16 to 200.



(a) Number of operations per second

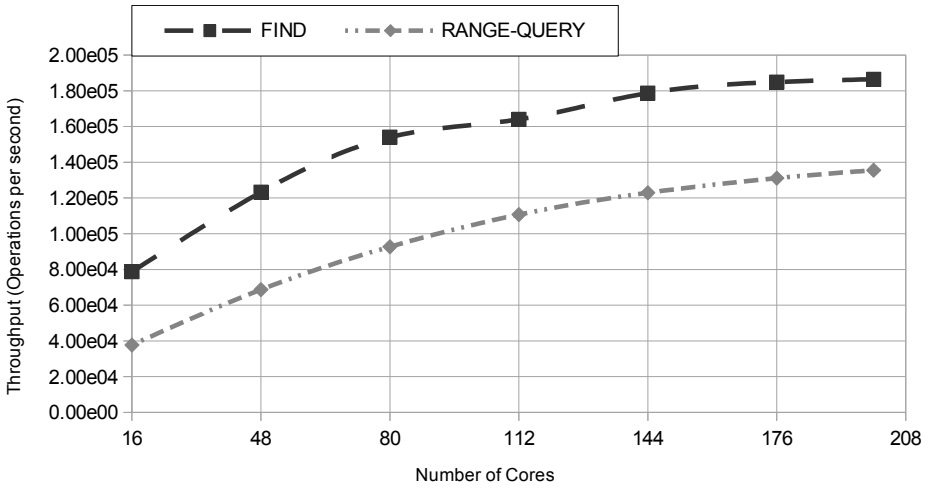


(b) Number of returned messages per second

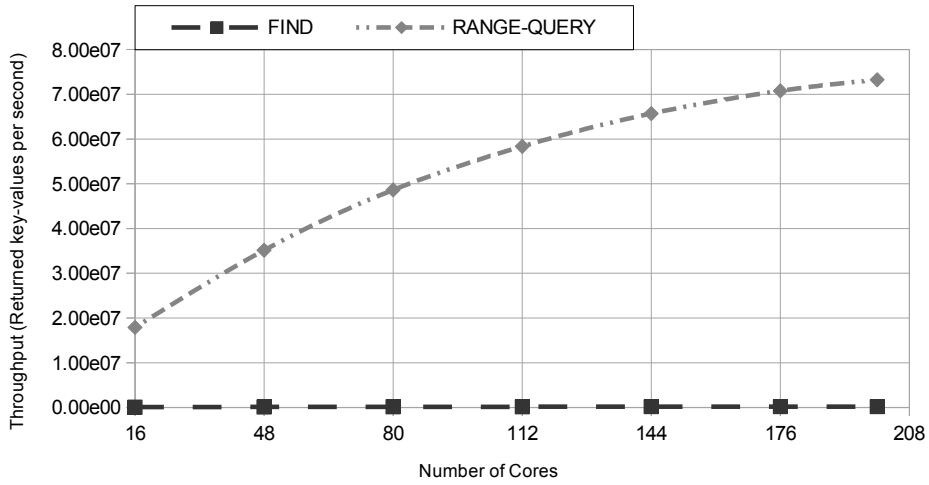


(c) Number of returned key-values per second

Figure 9. Scaling behaviour of RANGE-QUERY with three fixed size range queries: Small = 100, Medium = 1000 and Large = 10000 maximum key-value results. Workload uses 100% RANGE-QUERY operations. Semantics used is no-consistency.



(a) Operations per second



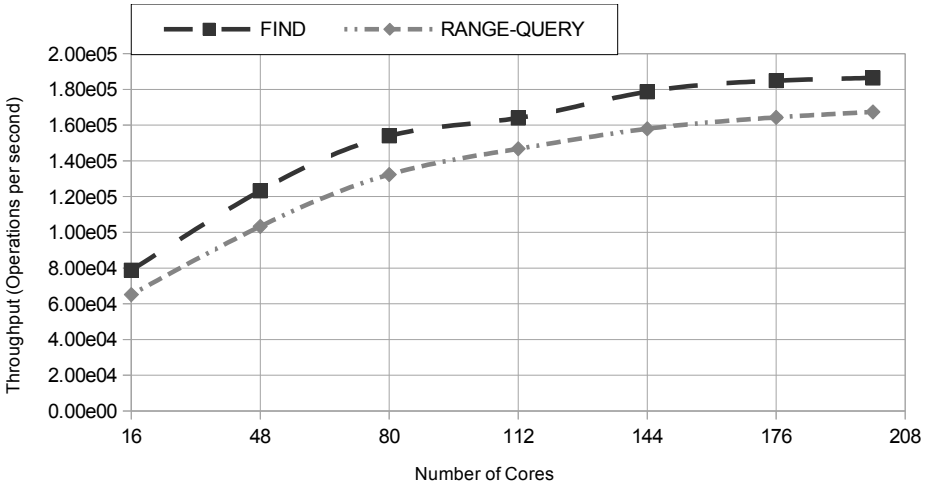
(b) Returned key-values per second

Figure 10. Performance comparison between FIND and RANGE-QUERY with a medium sized range query returning 1000 key-value results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no-consistency.

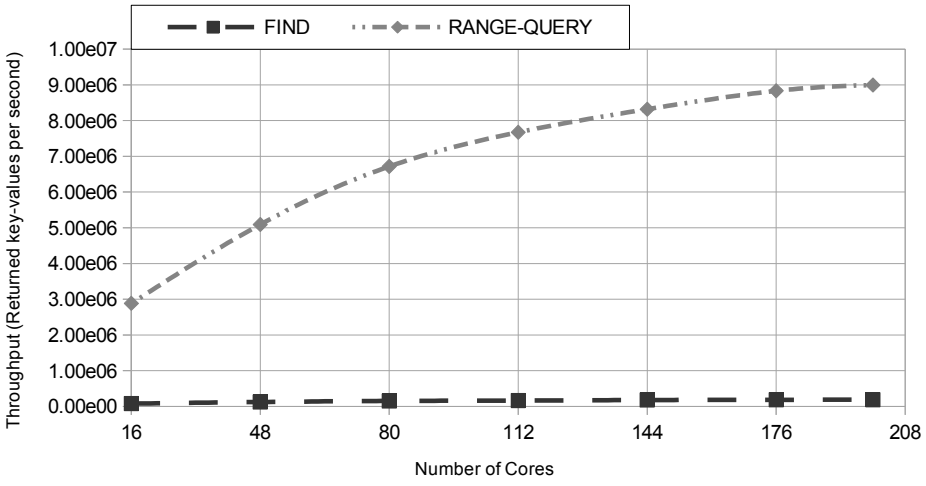
Figure 10 and 11 compare using a RANGE-QUERY versus a corresponding set of FIND operations for medium and small range queries respectively on a mixed workload. For small range queries, in terms of operations per second, the throughputs are similar. However, we obtain many more key-values per second with the range queries. We obtain a similar result for large range queries but with a reduced throughput but even more key-values per second. The results are returned via sending messages containing the key-values and corresponding payloads that fall inside the range queries. It demonstrates the ability of our skip list implementation to take advantage of the orderliness of the data to perform better on range queries.

3.3. Effect of Dimension on Range Query

Figure 12 shows performance of our skip list for multi-dimensional range queries. As before we fix the size of result and evenly distributed the ranges across the dimensions so that for D dimensions each dimension had the D^{th} root of the results. Because the skip list is ordered dimension by dimension the only key-value results that are likely to be returned by one



(a) Operations per second



(b) Returned key-values per second

Figure 11. Performance comparison between FIND and RANGE-QUERY with a small sized range query of 100 key-value results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no-consistency.

process in a single message are those whose range is in the last dimension. As a result, adding dimensions further aggravates the tendency for the number of messages to grow to the number of returned key-values as the number of cores increases. This is clearly evident in Figure 12 where there is a performance drop as the number of dimensions increases. This was expected since skip lists do not generally perform as well for high-dimensional data. There are other techniques such as skip-web [16] that provide an interesting extension.

3.4. Real-life Example

An example as to why skip list data structures are useful for highly dynamic applications we implemented a dynamic graph application[3] with a dataset of all commercial aircraft flights in North America for one month. The data consists of arrival and departure times, locations and additional information about the aircraft. As shown in Figure 13 we created a web-socket front end to display this information. Users enter a query in the boxes to the left, or choose

[3] TAO [17] by Facebook was introduced to solve a similar problem on social graphs.

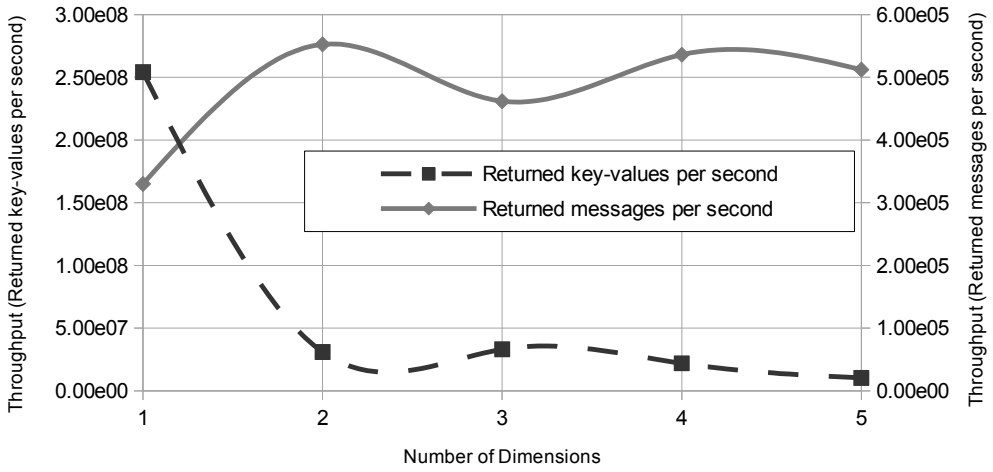


Figure 12. Throughput versus dimension for range queries. (Separate y-axes for returned key-values per second and returned messages per second). Workload uses 100% RANGE-QUERY operations with range query size of maximum 4096 key-values. Semantics used is no-consistency.

a range by re-sizing the green and red boxes in geographical display. This creates a range query that returns all aircraft departing from the green box or those arriving in the red box. Each edge represents the flight of one aircraft. Once queried the data is continuously updated in time as flights depart and arrive at their destination.

The back-end system uses our skip list implementation with application processes continuously receiving new data to add to the skip list. We use the longitude and latitude of the departure and arrival location (i.e., the edge between two vertices in the graph) lexicographical sorted by first the departure and then arrival location. This makes it possible to perform queries over two spatial ranges. We extended the behaviour of the skip list process by having the process invoke a DELETE operation when the elapsed time is after the arrival time. The system is continuously adding and deleting flights as well as responding to one or more web client queries from the front end.

An advantage of our implementation is that it is fully distributed efficiently making use of both multicore and multiple machines to scale across hundreds and potentially thousands of cores. As well, unlike the typical concurrency libraries, we did not have to add a new operation to automatically delete nodes, since the nodes were processes, we simply had processes invoke an existing operation.

4. Related Work

A skip list is a well-known data structure that has been frequently implemented as a concurrent data structure [3,4,18–22]. Unlike the work on concurrent data structures our implementation does not depend on shared memory and does not have the non-determinism inherent in lock-based designs. There are some similarities in the design, for example, the notion of atomic actions, consistency, and techniques such as hand-over-hand access and forward-only traversal. Another significant difference is that the list elements are active processes where the data structure has control over the operations, not the application processes as in the case of concurrent data structures.

Our coarsening of the skip list by having each process store a set of keys is similar to Leaplists [22], a concurrent data structure where each node stores multiple keys. Unlike Leaplists, which used transactional memory, in our case the multiple keys helped to reduce

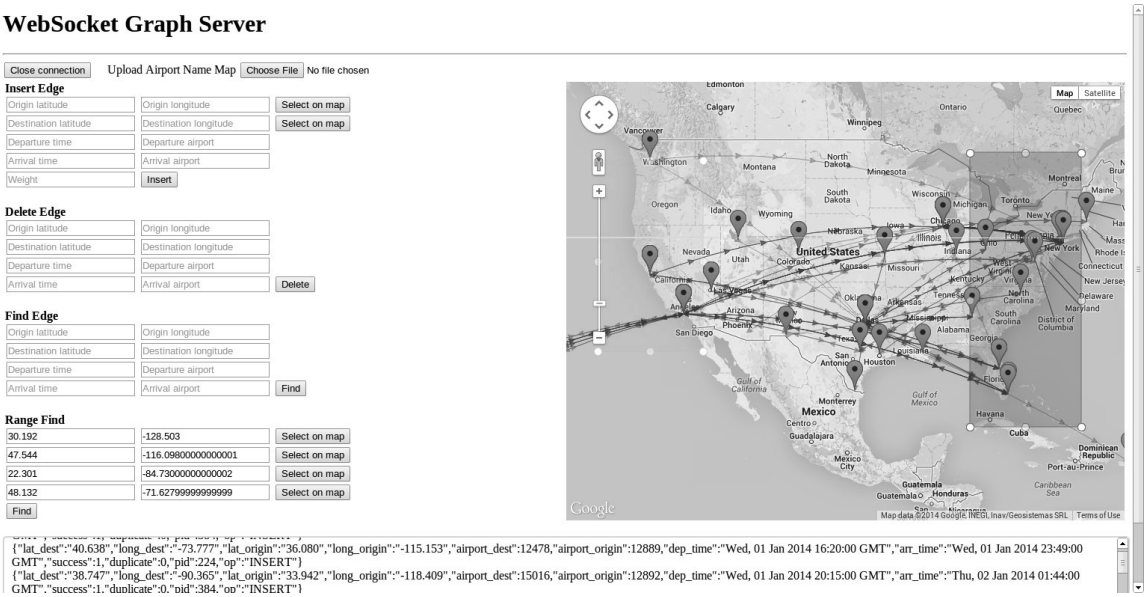


Figure 13. Websocket interface to aircraft example with data from our skip list. [4]

the amount of messaging and provided a way to better adjust the amount of concurrency to the characteristics of the machine.

As previously mentioned hash-based key-value stores are an important data structure in distributed computing environments. There are systems designed to extend key-value stores to efficiently process range-queries in the case of disk-based systems [23]. As well, there is recent work on key-value stores for in-memory structures [24], but these have not been extended to range queries. Our implementation is focused on in-memory data structures and support for range queries. SD-Rtree [25], RAQ [26] and SkipTree [27] are distributed tree data structures that support multi-dimensional range queries. Our implementation has the advantage of creating query splits automatically based on the probabilistic nature of the skip list tower creation. Where there is dense data, there are more towers and hence more opportunity for parallel access.

Pastry [28] and Skip Webs [16] are designed for peer-to-peer systems. Although these systems are based on message passing they are optimized for coarse-grain access to large data. Our distributed skip list is built with a focus on scalability and low-latency for in-memory access to data rather than large blocks of data residing on disk. As well, as expected, distributed storage systems are designed for availability and reliability, which we do not consider at present.

5. Conclusions

In this paper we presented a novel implementation of a fully-distributed, in-memory, skip list data structure and augment the structure to support range-queries. We discussed a service oriented approach to scalable distributed data structures in MPI where the list nodes are active processes that have control over the operations, unlike in concurrent data structures. The key-values in the skip list belong to processes instead of being tied to memory, which makes it easier to load-balance them across cores on one or more machines. We showed that the operations can be implemented atomically and that there is an interesting ordering property whereby operations cannot overtake one another. This property makes it possible to reason about operations deterministically and made it possible to implement total ordering

[4] Web socket server and front-end implemented by Edward Soo.

and sequential consistency semantics. We introduced shortcuts, a novel mechanism used for service discovery and as an optimisation technique to trade-off consistency semantics with performance.

Our design enables a number of tuning parameters that allow us to adjust the amount of asynchrony to better overlap communication with computation and overlap the application requests with skip list operations. It is also possible to adjust the granularity and experiment with different load-balancing strategies by changes to the free node service, shortcuts, or tower heights. There is the added task of tuning these parameters. We have been investigating automated techniques for setting and testing parameters and the possibilities for adaptively optimising the values during execution. Given the diverse performance characteristics of cluster and cloud infra-structures, the ability for the system to adapt to the infra-structure is necessary for performance portability.

The experiments demonstrated the ability to scale to an ordered skip list with millions of data items with up to 20,000 MPI processes executing on 200 cores. We showed that the sequential-consistency and no-consistency semantics show good scaling behaviour with the size of the list and the number of cores. We investigated the scaling behaviour of range-queries on different sized queries and showed that they scale well with respect to the number of results received by the application. Range-queries outperform use of multiple find operations in terms of the number of key-value pairs returned per second. We also report query searches on higher dimensions for completeness. As expected, skip-lists do not perform as well for high-dimensional data. We also demonstrated the advantage of our skip list design through a dynamic graph application that uses real-time streaming data.

In summary, the advantage to our message-based design is its overall flexibility. The system acts deterministically with operations flowing in and results flowing out. There are no timing issues, the system can take advantage of multicore, and will operate correctly whether it runs inside one machine or hundreds of machines. Our design provides the flexibility to adjust the granularity, the concurrency and the amount of parallelism.

Acknowledgements

We gratefully acknowledge NSERC Canada and Mitacs Inc. for their support of this research. We thank Edward Soo for help on the airline application. We thank the reviewers for their helpful comments and wish to especially thank the reviewer who helped us significantly improve Section 2. FG-MPI is available for download.

References

[1] Sarwar Alam, Humaira Kamal, and Alan Wagner. Service Oriented Programming in MPI. In *Communicating Process Architectures 2013*, pages 93–112. Open Channel Publishing Ltd., England, August 2013.

[2] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[3] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[4] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[5] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 1106–1114, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[6] Humaira Kamal and Alan Wagner. Added concurrency to improve MPI performance on multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229 –238, sept. 2012.

[7] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4):293–309, 2014.

[8] Argonne National Laboratory. MPICH2: A high performance and portable implementation of MPI standard. Available from http://www.mcs.anl.gov/research/projects/mpich2/index.php.

[9] Bin Liang, Yiqun Liu, Min Zhang, Shaoping Ma, Liyun Ru, and Kuo Zhang. THUIR-DB: A large-scale, highly-efficient index fast-access key-value store. *Journal of Computational Information Systems*, 9(6):2347–2355, 2013.

[10] Adam Prout. The Story Behind MemSQLs Skiplist Indexes. Available from http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes.

[11] Alex Brodsky, Jan Bækgaard Pedersen, and Alan Wagner. On the complexity of buffer allocation in message passing systems. *Journal of Parallel and Distributed Computing*, 65(6):692 – 713, 2005.

[12] G.F. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. International computer science series. Addison-Wesley, 2011.

[13] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[14] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9:173–191, 1995.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[16] Lars Arge, David Eppstein, and Michael T. Goodrich. Skip-webs: efficient distributed data structures for multi-dimensional data sets. In Marcos Kawazoe Aguilera and James Aspnes, editors, *PODC*, pages 69–76. ACM, 2005.

[17] Nathan Bronson et al. TAO: Facebook's Distributed Data Store for the Social Graph. *USENIX Annual Technical Conference (ATC)*, 2013.

[18] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[20] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[21] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 11–pp. IEEE, 2003.

[22] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: lessons learned in designing tm-supported range queries. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM.

[23] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. of ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 25–36, 2012.

[24] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[25] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree: A scalable distributed Rtree. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 296–305, 2007.

[26] Hamid Nazerzadeh and Mohammad Ghodsi. RAQ: A range-queriable distributed data structure (extended version. In *In Proceeding of Sofsem 2005, 31st Annual Conference on Current Trends in Theory and Practice of Informatics, LNCS 3381*, pages 264–272, 2005.

[27] Saeed Alaei, Mohammad Ghodsi, and Mohammad Toossi. Skiptree: A new scalable distributed data structure on multidimensional data supporting range-queries. *Computer Communications*, 33(1):73–82, 2010.

[28] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.