Distributed Skip List in Fine-Grain Message Passing Interface

Implementation and Analysis of a Dictionary Data Structure that supports Range Queries

by

Sarwar Alam

B.Sc., BRAC University, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

March 2014

© Sarwar Alam 2014
Abstract

With scalability in mind, we have implemented a pure message-passing distributed data structure ideal for range queries. The structure is a distributed skip list that takes full advantage of Fine-Grain MPI to execute on a variety of scales ranging from a single multicore to multiple machines across clusters. The skip list data structure provides parallel implementation of range queries. Our implementation architecture is based on several services layered on top of each other that simplifies the effort required in building distributed code. Unlike concurrent data structures the distributed skip list operations are deterministic and atomic. The layered service architecture of our implementation exposes several control parameters that make it easy to distribute load, have operation flow control, vary granularity at different layers and tune performance to the underlying machine and network. Range-queries are implemented in a way that parallelizes the operation and takes advantage of the recursive properties of the skip list structure. We investigate a shortcut mechanism that alleviates the bottleneck at the head and introduces semantic trade offs between performance and consistency. We report on the performance of the skip list on a medium size cluster of two hundred cores with twenty thousand processes.
There are two data structure implementations discussed in this thesis. The first is the distributed ordered linked list and the second is the distributed skip list.

The sections that discuss the distributed linked list data structure are Sections 2.4, 2.5, 3.1, 3.2, 3.5, 3.6 and 4.1. These sections were combined then modified to form a paper that has been published in the conference proceedings of Communicating Process Architectures 2013 [1]. This paper was written in collaboration with Dr. Alan Wagner and Dr. Humaira Kamal. I contributed in the design and programming of the implementation and in writing the experiments section.

The sections that describe the distributed skip list implementation are Sections 2.4, 2.5, 3.1, 3.3, 3.4, 3.5, 3.6 and 4.2. These sections were modified and combined to form another paper that is currently under submission to a conference proceedings. This paper was also written in collaboration with Dr. Alan Wagner and Dr. Humaira Kamal. My contribution was in designing, implementing, testing and analyzing the distributed skip list and reporting the results in the experiments section.
Table of Contents

Abstract .................................................... ii
Preface ...................................................... iii
Table of Contents .......................................... iv
List of Figures .............................................. vii
Acknowledgements ......................................... viii
Dedication ................................................... xi

1 Introduction ............................................... 1

2 Background and Related Work ............................. 6
  2.1 Terminology ............................................. 6
  2.2 Sequential Implementation ............................. 8
  2.3 MPI ..................................................... 11
  2.4 Fine-Grain MPI .......................................... 12
  2.5 Related Work ........................................... 14

3 Implementation ............................................. 17
  3.1 Representation .......................................... 17
  3.2 Linked List .............................................. 19
    3.2.1 Design ............................................. 20
    3.2.2 Operations ......................................... 24
    3.2.3 Granularity ......................................... 28
  3.3 Skip List ............................................... 30
# Table of Contents

3.3.1 Design ................................................. 30  
3.3.2 Operations .......................................... 33  
3.3.3 Granularity .......................................... 43  
3.4 Proof of Correctness ................................. 45  
  3.4.1 Freedom from Deadlock ......................... 45  
  3.4.2 Order of Operations ......................... 46  
3.5 Service Interface ................................. 49  
  3.5.1 Applications Processes ....................... 49  
  3.5.2 Shortcuts .......................................... 50  
  3.5.3 Free Process Sub-service ................. 52  
3.6 Parameters ........................................... 55  
  3.6.1 Flow Control (W) ........................... 55  
  3.6.2 Asynchronous Communication (K) ........ 56  

4 Evaluation .............................................. 58  
  4.1 Performance of Distributed Linked List ....... 58  
    4.1.1 Consistency Semantics .................. 59  
    4.1.2 G versus P .................................. 60  
    4.1.3 W and K .................................... 62  
  4.2 Performance of Distributed Skip List ........ 63  
    4.2.1 Consistency Semantics .................. 64  
    4.2.2 Effect of Probability During Tower Creation .... 66  
    4.2.3 Range Queries .......................... 66  
    4.2.4 Effect of Dimension on Range Query .... 68  
  4.3 Summary ........................................... 72  

5 Conclusion ............................................. 73  
  5.1 Future Work and Extension ..................... 74  

Bibliography ............................................. 77  

Appendices  
A Pseudocode ........................................... 82  

v
Table of Contents

B Parameters available in experiments ..................... 85
## List of Figures

2.1 An example ordered linked list. Each node is a location in memory. Every node has a key-value, a payload and a pointer to the next node’s memory address. ............................... 9

2.2 An example of a skip list data structure. Every node has a key-value, a payload and a tower of pointers to successors on different levels. The search path for a query for the key 38 is shown with thick shaded arrows. ................................. 9

2.3 A cluster consisting of 2 machines each with 2 cores and a [4,2,2] mapping with one manager, two list nodes, and an application process per OS process (see Section 3.5 for more details) and 16 MPI processes in total. For our list services, more list node processes will be added per core by increasing the nfg count (currently 4) in the mpiexec command. ................................. 13

3.1 Contents of a single MPI process that corresponds to a single linked list element. Each process contains a key, a payload, a pointer to the next process and the minimum key value of the next process. ................................. 20

3.2 An example of a distributed chain of processes that make up a linked list service. Processes are pointed to using MPI rank semantics. The head process does not contain a key or payload, and a special sentinel pointer in the last element marks the tail of the service. ................................. 21

3.3 Implementation of the ordered-linked list operations (a) FIND, (b) INSERT, and (c) DELETE. Arrows denote the different communications among processes A, B and C (the three processes that participate in the operations). ................................. 25
3.4 (a) Example of a skip list process with connections from previous and next processes, and (b) Message-driven main loop of the process. 31
3.5 An example of a skip list where each list item is a process. 32
3.6 Steps taken to perform the FIND operation in the ordered skip list. The cross-patterned arrows illustrate how a search query travels along the skip list. Shaded arrows represent messaging and thin arrows represent connection links. At any point in the search route, a decision is made whether to skip forward or to go down one level until the search key is found. 34
3.7 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. 36
3.8 Steps taken to perform the INSERT operation in the ordered skip list. Shaded arrows are communications; thin arrows are link connections. M denotes a manager process, while C denotes the free process C. The figure shows only segments of the towers of all processes, hence all processes need not be immediate successors as may be apparent from the figure. 37
3.9 Steps in completing a DELETE operation in the ordered skip list. Shaded arrows are communications; thin arrows are link connections. A node’s tower is trimmed by parts from its top level to its lowest level. A node continues normal operation when trimmed only partially. When the entire tower is trimmed, the deleted node reports back to application and adds itself to local free process list. 40
4.1 Throughput achieved with different consistency semantics with a fixed list size ($2^{20}$), evenly distributed over the number of cores=$O \times M$. Only the no consistency curve shows slight degree of scaling. [P=10]. 59
4.2 Effect of changing $P$, while keeping $P \times G$ constant. The list size and the number of cores are fixed. Semantics used are (a) sequential consistency and (b) no consistency. (ListSize = $G \times P \times O \times M=2^{20}$ and cores=$O \times M=176$). 61
List of Figures

4.3 Throttling effect of $K$ on the throughput while increasing the outstanding requests ($W$) by the application processes. List size used is $2^{20}$. $P = 100$. $O \times M = 192$ cores. Sequential consistency semantics are used in this experiment. .................................................. 63

4.4 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 grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. ................................................................. 65

4.5 Effect of probability during tower height selection on the skip list service. Probability = 0% (not shown) and 100% both result in performance like that of a linked list. ................................. 67

4.6 Scaling behavior of RANGE-QUERY with three fixed size range queries: Small = 100, Medium = 1000 and Large = 10000 results. Workload uses 100% RANGE-QUERY operations. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. ................................................. 69

4.7 Performance comparison between FIND and RANGE-QUERY with a medium sized range query returning 1000 results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. .......................... 70

4.8 Performance comparison between FIND and RANGE-QUERY with a small sized range query of 100 results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. ................................. 71

4.9 Throughput versus dimension for range queries. (Separate y-axes for results per second and reply messages per second). Workload uses 100% RANGE-QUERY operations with range query size of 4096. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. ........ 72
Acknowledgements

I am very grateful to my supervisor Dr. Alan Wagner, who guided me in building up my thesis. His experience made it easy for me to choose directions whenever I was stuck in my implementation. His helping hand would extend to support me mentally and also help me in writing. He was always available and was optimistic in every small steps that I took to build up my masters thesis. It was my pleasure to have the opportunity to work with him.

I would take this opportunity to thank Dr. Norm Hutchinson for reading my thesis and providing useful suggestions and helpful comments.

I am also thankful to Dr. Humaira Kamal, who has helped me in writing the two conference papers that are direct results from this thesis. Her work on Fine-Grain MPI is the main environment on which this thesis is built upon. She was available for help amidst her busy graduation schedule.

I am very thankful to my fellow student friends Ryan Chung, Matthew Dirks, C. Albert Thompson, Shuo Shen, Imran Ahmed and Jahidur Rahman, who were always free for laughs and games that kept me sane and active.

It would not be possible for me to emphasize the magnitude of support and help I have received from my wife. She works hard everyday to make my life easier. I would not have gone the distance without her. She always believed in me and motivated me to reach for the stars.

Finally, I would thank my family, siblings, extended family and friends. My whole life’s great adventures are because of all of you.
Dedicated to my dear wife and parents.
Chapter 1

Introduction

Process-oriented languages in general, and communication libraries like MPI in particular, do not have the rich collections of data structures present in today’s object-oriented programming languages. The lack of data structure libraries for MPI makes it necessary for programmers to create their own and adds to the overall programming complexity of using message-passing languages. The difficulty in having to explicitly manage the distribution of the data is often pointed to as a major impediment to the use of message passing [8]. One approach that is at the heart of process-oriented systems [28,32] and many distributed systems [7,29] is to implement the data structure using its own collection of processes and provide the distributed data structure as a service. We call this service-oriented programming. A service-oriented approach enforces low coupling, because components interact only through message-passing and cannot directly access one another’s data, and leads to higher cohesion inside components, because application and data structure management code no longer needs to be combined together. We investigate the use of a process-oriented design methodology together with a service-oriented programming approach (design pattern) to provide highly scalable, easy to use libraries for complex data structures. We are interested in highly scalable implementations of dictionary data structures for storing key-value pairs to support range queries. A data structure that is often used for this purpose in sequential programs is a skip list [27]. Skip lists support FIND, INSERT and DELETE of key-value pairs and can be augmented with a range query operation that can take advantage of the ordering of the keys to output the result in a single operation.

There are several implementations of concurrent data structure implementations of skip lists build on top of a shared memory programming model
using locks or lock-free techniques [5, 12, 15]. 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.

We take a similar fine-grain approach to parallelization of a skip list but we do so using message-passing. Our design introduces parallelism from the very start where every key-value in the data structure is its own process and operations are implemented by 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. There are several advantages to this approach. 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 being 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 characteristics of the underlying machine. Fourth, it makes it easy to compose not only between OS-level processes but to compose in a fine-grain manner with function-level concurrency inside the program.

We implement our skip lists using FG-MPI [21, 22], a fine-grain version of MPI that extends the MPICH2 [3] middleware. It extends MPICH2 by adding an extra level of concurrency (interleaved concurrency) whereby every OS process comprises of several MPI processes, rather than just one. It has its own integrated runtime scheduler [23], which optimizes communication between processes, and support for dynamically allocating processes. FG-MPI uses a coroutine-based non-preemptive threading model together with its user-level scheduler to make it possible to support thousands of processes inside an OS process to expose finer-grain concurrency with no change to the programming model. FG-MPI constructs “multiple program multiple data” (MPMD) type of programs that scale to hundreds and thousands of processes across a cluster of multicore machines. The support for
having many light-weight MPI processes and the flexibility to map processes to functions\footnote{C functions, named procedures.} to execute concurrently inside an OS process, as well as in parallel across the cores and machines in a cluster makes it possible to consider a service-oriented approach to providing a distributed data structure.

Since most MPI implementations bind an MPI process to an OS process, one could implement a dictionary data structures as a collection of separate OS processes. Previous work by Iancu \textit{et al.} \cite{Iancu2013} has shown that over-subscription of MPI processes to cores often leads to poor performance and because of this the usual practice is to execute one MPI process per core. This directly constrains the number of MPI processes to the number of cores which results in the design of processes to match the machine rather than the available concurrency present in the algorithms. This leads to more coupled, less cohesive programs. A second drawback is that when executing with one MPI process per core, it is difficult to keep the application and service processes constantly busy. One common approach to avoid this problem is to use MPI in combination with Pthreads, or some other runtime, to combine the MPI application process with a thread implementing the concurrent data structure. This provides a fine-grain implementation, but at the expense of portability and predictability because the multiple models and runtimes interact in unpredictable ways. Other than MPI, there are concurrent data structures and APIs designed for multicore, like OpenMP, that can be used, but they do not scale to execute on clusters or efficiently scale to the hundreds and thousands of processor cores which is the intent of our design. Our approach to the design of an ordered list service relies on FG-MPI and avoids the previously mentioned problems to provide a scalable data structure service to applications.

In this thesis we give the design, implementation and evaluation of the distributed skip list structure by first introducing how we built an ordered linked list data structure in FG-MPI and then how we extended it to a distributed skip list. This thesis makes the following contributions:

\begin{itemize}
  \item A demonstration of how the FG-MPI extensions to MPI make it pos-
Chapter 1. Introduction

It is possible to follow, within MPI, a process-oriented design methodology where the resulting process structure reflects the problem rather than the machine. This makes it possible to add complex data structures to MPI by using a service-oriented approach.

- We provide a novel design of a pure message-passing implementation of an ordered linked list and skip list targeted to the type of multicore clusters that are common to high-performance and cloud computing environments. Unlike concurrent skip list structures, the system is deterministic and in essence acts like a pipeline with service requests flowing in and results flowing out.

- We show how range queries can be parallelized in this structure where they are split over the extent of the range.

- We introduce a variety of performance-tuning parameters and investigate the effect of these parameters with regards to tuning the service to the underlying characteristics of the machine and network to achieve good performance across varied machine sizes and communication fabrics.

- We introduce shortcuts, a novel technique related to service discovery and relate this to trade-offs between data consistency and performance. We also show that the skip list operations are atomic, and have a non-overtaking property that makes it possible to use shortcuts as an optimization technique to trade off consistency semantics for performance.

- Unlike many MPI programs, our mapping of processes to OS processes on a multicore machine makes it possible to expose a large amount of concurrency that can be predictably scheduled. We also introduce a free process sub-service to allocate and deallocate processes to dynamically adjust to varying loads and demands.

- We achieved scalable performance on a medium size cluster of over 200 cores. Our skip list service can flexibility scale-up with added
Chapter 1. Introduction

granularity and fine-grain concurrency on multicore and scale-out to take advantage of the parallelism in a cluster.

In Chapter 2, we introduce some background information on the data structures we implemented and on MPI and FG-MPI libraries. We list related work that is similar to our implementation in Section 2.5. The steps taken in building the two distributed data structures, Ordered Linked List and Skip List, are presented in different sections in Chapter 3. Chapter 3 also includes sections on auxiliary services that complete our implementation and sections on model variations and parameters. We present performance evaluations separately for our distributed ordered linked list and distributed skip list in Chapter 4. Finally, conclusions and possible future extensions are mentioned in Chapter 5.
Chapter 2

Background and Related Work

This chapter introduces some of the domain specific terminologies used in this thesis and describes the sequential counterparts of the data structures that we developed. We describe the properties of the structures and the operations used on them. A brief introduction to MPI is presented before discussing the new features available in FG-MPI. At the end, this chapter discusses related work and contrast our approach to other work.

2.1 Terminology

The following technical terms are used in the thesis to describe the structure. A key-value pair refers to a record stored in our data structure service. The key is used to index and search the data associated for the record. This data is referred to as value or payload. We refer to tasks that are invoked for our data structure service as operations. The terms requests, messages and actions are also used to refer to operations. The main types of operations that we consider are INSERT, DELETE, FIND, UPDATE and RANGE-QUERY. The INSERT operation is used to enter a new key-value pair into the distributed structure. INSERT operations are halted when there are duplicate keys in the structure or when there are no locations left to put a new record. DELETE is the inverse operation of INSERT, which removes a record that already exists or does not change the structure if the key is not found. The FIND operation is used to check if a key is already present in the structure and if so it fetches the associated value. FIND is also referred to as QUERY or SEARCH. The RANGE-QUERY operation
2.1. Terminology

therefore searches keys that fall within a given range, and when matches are found, the associated values are returned. The UPDATE operation is similar to the FIND operation in that a key is searched for. They differ only in that the UPDATE operation changes the associated payload of the search key whereas FIND returns the current payload provided the key is present in the distributed structure. We therefore do not separately describe the steps needed for the UPDATE operation and only discuss steps for FIND, INSERT, DELETE and RANGE-QUERY operations.

We developed our system in a process-oriented approach. The locations where key-value records are stored are inside processes. Each process executes with its own set of data. Inter process communication is done via message passing. To distinguish between the processes that FG-MPI produces and the processes that the operating system starts up, we differentiate the terms processes and OS processes. We use the term OS process to refer to operating system level processes which may contain multiple processes (or MPI processes) inside them. Processes are also sometimes referred to as nodes. The MPI processes within a single OS process are said to be co-located with each other. The OS processes may themselves be in the same machine across different cores or spread out to different servers or machines. Because data are stored in processes, this implies that a pointer to data uses some type of a process identifier. This identifier is called a rank in MPI and is represented as integer index starting from 0 to the number of MPI processes minus 1. The pointer to a process is also referred by the term link. The term shortcut is used to denote a quick path to a destination process.

Operations reach their destination via message passing between processes. This transfer of operation requests are referred to as hops, skips, jumps or communication. Communications can be in the forward or backward direction, which depends on whether a message is sent to the next process in a link or is sent back as a reply to a message from a predecessor node. Operations are generally started from a head or root process and will not go past the tail or end if a message does reach that far.

There are four types of process functions or named procedures in our service. They are list processes, free processes, manager processes and appli-
2.2 Sequential Implementation

**List processes**. List processes are the nodes that store record values in our service, while free processes are the same nodes but without any records. Manager processes are responsible for allocation of free nodes or processes and they are available for other administrative tasks when needed. Application processes provide the service interface of our distributed data structures to external applications. The *granularity* of an MPI process is a measure of the number of key-value records that it contains. Granularity may also be weakly referred to as a *batch* of records. The *tower height* or *level* of a skip list process refers to the number of link pointers it has to its successors. Performance measures in processes include *latency* or the round trip time for requests and *throughput* which we measure as number of operation requests completed per second. We also use a variation of throughput in terms of the number of reply messages received by a process per second, and the number of effective result values received per second.

The next section describes the basics of the data structures we have implemented in a sequential programming environment to familiarize concepts to better understand later discussions.

### 2.2 Sequential Implementation

An ordered linked list consists of groups of elements that are chained one after another in a sorted fashion. Elements are sorted on a key value and each contains an associated payload. An element points to its successor by storing the address of the memory location of the next node. Operations on a linked list start from the head element and travel forward through the list until the operation key or location is found or the tail is reached. An example linked list is shown in Figure 2.1. An in-memory sequential implementation of a linked list performs the operations INSERT, DELETE or FIND one at a time. The entire data structure has to wait until a single query finishes before it can take another. Algorithms on a linked list have a time complexity of $O(n)$ where $n$ is the length of the list (i.e., the number of elements in the list). We can take advantage of parallelism to provide a stream of multiple operations on a linked list and target different sections of
2.2. Sequential Implementation

Figure 2.1: An example ordered linked list. Each node is a location in memory. Every node has a key-value, a payload and a pointer to the next node’s memory address.

the data structure. One way this can be done is by use of multiple threads and locks on the different sections of the data structure [15]. Unlike the shared memory locking method, we take a pure message passing approach.

A skip list [27] is a data structure similar to that of a linked list but with more pointers to different successor nodes creating shortcuts to search paths through the structure. Skip list operations improve on the performance of a linked list by having an expected time complexity of $\mathcal{O}(\log n)$ for each operation. An example skip list structure is shown in Figure 2.2. Each node in a skip list has a number of links to successor nodes instead of a single pointer to the immediate next node. As a result, a node has shortcut

Figure 2.2: An example of a skip list data structure. Every node has a key-value, a payload and a tower of pointers to successors on different levels. The search path for a query for the key 38 is shown with thick shaded arrows.
2.2. Sequential Implementation

links to distant successors that can speed up search for the destination of an operation. Each pointer on a skip list tower corresponds to a level in the skip list. The lowest level pointers essentially links to the immediate neighbors as in the linked list. Hence, a skip list contains a corresponding linked list at the very lowest level. In fact, the chain of shortcut links on every level of the skip list forms a new linked list. To gain the logarithmic expected performance, a node that appears on a particular level of the skip list should have a diminishing chance of appearing in a level higher. For this purpose, a node can be made to re-appear in a level higher based on a random coin toss with a fixed bias (some probability $P$). This results in creating skip list towers using a Bernoulli process, where the probability of having a tower of height $h$ is $P^h$. The associated linked list on a given level, $l$, of a skip list is a proper subset of the associated linked list on level $l - 1$. An operation on a skip list can therefore take advantage of the sparseness of the associated linked lists on higher levels to quickly jump close to the destination, and go down a level when the operation key is not in the next node of a given level. Figure 2.2 also shows the path taken for an operation that works on the key value 38. The head node is arranged to always have tower level equal to the highest tower in the skip list data structure. An operation starts from the highest level of the head node, and attempts to find the search key on the current level as a linked list. When the operation reaches a node with a value greater than the key, the attempt to find the destination is restarted after going down the current level at the predecessor node. A parallel implementation of a skip list will be able to take advantage of the logarithmic nature of operations to create a more scalable system than that of a parallel linked list, so as to better serve continuous stream of requests from multiple applications.

A range query is an operation that specifies two keys as the start and end of a range, and expects to search all node entries within this range. Range queries can be executed on both the ordered linked list and the skip list because elements are stored in sorted order. A range query can be implemented by first searching the start key of the range, and then iterating through the next nodes until the end of the range is reached. Skip lists allow
2.3 MPI

MPI or the Message Passing Interface is a standard for distributed and parallel computing with pure message passing model as opposed to a shared memory model. The MPI standard is now in its third version. Argonne National Laboratory maintains MPICH2 [3], a free implementation of the MPI-2 standard, which is the library we used for our code. Parallel programs with MPI are process oriented with each process having its own data to work on. Interaction and synchronization between processes on the same or different machines are performed through message passing. MPI programs start with an MPI\_Initialize() function and end with an MPI\_Finalize() function. In between these function calls, an implementation of the MPI standard performs background services that manages reliability, scheduling and synchronization of process communication across cores, machines and networks. Codes written with MPI are compiled using the mpicc command and are executed using the line:

```
mpiexec -n source_file_name
```

MPI provides both synchronous and asynchronous communications between processes. In the former, a sender of a message will wait until a receiver has started receiving, while in the latter type of communication, the sender only waits until it has completed copying the message to its network buffer. MPI also provides both blocking and non-blocking functions at both ends of a communication (i.e., when sending and receiving messages). The basic functions for sending a message to a process are MPI\_Send() and MPI\_Ssend(), where the latter represents the synchronous type of message sending. MPI\_Isend() is generally used for non-blocking sends. The basic
functions for receiving messages are `MPI_Recv()` and `MPI_Irecv()`, where the latter is the non-blocking receive. MPI does not provide freedom from deadlock; instead it is the programmer’s responsibility. The use of non-blocking receives play a key role in helping to avoid a deadlock, but it is neither sufficient nor necessary.

Messages are identified with three parameters of communication, the process identifier which is called the MPI rank, the message tag and a communicator, which defines the communication context or group of processes associated in the communication. MPI rank is similar to an array index; the first process has rank 0 and the last process has rank $size - 1$. A process can request to receive from `MPI_ANY_SOURCE`, which accepts from any sender rank. Similarly, a receiver can request to listen for `MPI_ANY_TAG`. The MPI standard specifies many more communication functions which have use-cases for various application needs.

### 2.4 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 Fine-Grain MPI (FG-MPI).

FG-MPI [21] extends the MPICH2 middleware. It 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.

As described in detail in Dr. Humaira Kamal’s thesis [19], an FG-MPI execution, $[P, O, M]$, can be described in terms of $P$, the number of MPI
2.4. Fine-Grain 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 \(N = P \times O \times M\). Figure 2.3 shows a \([4, 2, 2]\) mapping started with the command:

```
mpiexec -nfg 4 -n 4
```

Figure 2.3: A cluster consisting of 2 machines each with 2 cores and a \([4, 2, 2]\) mapping with one manager, two list nodes, and an application process per OS process (see Section 3.5 for more details) and 16 MPI processes in total. For our list services, more list node processes will be added per core by increasing the nfg count (currently 4) in the mpiexec command.

![Diagram of a cluster with 2 machines, 2 cores each, and a [4, 2, 2] mapping with one manager, two list nodes, and an application process per OS process. It shows 16 MPI processes.](image)

We take a service-oriented approach to the design of the system where the skip list has a service interface that interacts with other processes through

---

\(^2\)We refer to these \(P\) MPI processes as *co-located* processes sharing a single address space.
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 deallocation 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 (explained with more details in Section 3.5). 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 [23] 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 initialize 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, there will still be an advantage to having a coarser-grain implementation with MPI processes managing more than one key-value pair.

2.5 Related Work

Linked lists and skip lists are well-known data structure that have been frequently implemented as concurrent data structures [12,16,26]. 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-
2.5. Related Work

hand access and forward-only traversal appear in [16] (Chapter 9). 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.

A notable advantage of our skip list data structure is that because keys remain sorted and we have recursive paths to a range of values, it is relatively easy to support range queries in a scalable manner, which is inefficient on simple hash-based structures. More generally in Peer-to-Peer computing Distributed Hash Tables (DHT) provide a pure message-based implementation of a look-up service designed for Internet applications [18]. The primary focus of these systems is on scalability and availability, particularly in the case where processes in the DHT frequently leave and join (i.e., churn), which is substantially different from our use case.

Our coarsening of the skip list by having each process store a range of keys is similar to Leaplists [4], 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 the amount of messaging and provided a way to better adjust the amount of concurrency to the characteristics of the machine. The focus of Leaplists was also on the use of skip lists for range queries but queries are not split and performed in parallel.

In cloud computing environments hash-based storage structures like Cassandra [25] (and key-values stores) are commonly used to allow for distributed access to the data. Although these systems are based on message passing they are optimized for coarse-grain access to large data. Our distributed linked list and skip list provide similar types of access to data as Cassandra but with more of a focus on scalability and low-latency where we are considering finer-grain 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. There is also similar work on distributed access to R-tree like data structures [11]. This work does not discuss multicore, consistency, and focuses on reducing

\footnote{The lock for the next item in the list is acquired before releasing the lock for the current list item.}
the number of messages and thus a coarse-grain one process per OS process implementation without the type of flexibility of our skip list design.

The focus of MPI is on performance and as a result it supports a wide variety of communication routines to match modern network protocols, interfaces and fabrics. In contrast to synchronous message-passing languages like occam-π [31], MPI programs make frequent use of non-blocking asynchronous communication to overlap computation and communication. The message latency between machines in a cluster is relatively high in comparison to on-chip memory-based messaging and this “send and forget” model of messaging makes it possible to off-load message delivery to the network interfaces and devices. The intent is to increase the network load to better utilize the network hardware and have more messages in flight to provide more work for each machine, thereby improving performance by mitigating the overhead of communication. As shown by Valiant [30], the amount of work needed per machine to overlap with communication depends on the latency and bandwidth characteristics of the network. This suggests that, for scalability, the messaging needs to be able to adjust to the characteristics of the network in much the same way that programs need to take into account the memory hierarchy. The tuning parameters introduced in our design come from a two-level view of the network and the need to adjust the messaging to the fast local and slower non-local communication mechanisms in a cluster of multicore machines. We view this as essential to the design of programs to scale to hundreds and thousands of processors and differentiates our work from languages and systems that execute on a single multicore machine.
Chapter 3

Implementation

We discuss the implementation of our skip list by first describing how we built a distributed ordered linked list that works with streams of operations. We then extend the structure to a distributed skip list and show how operations were modelled. We first introduce the node representation on which our data structures are based, after which we describe aspects of the design that depend on the structure. We then provide details about the different algorithm designs and considerations for our data structures.

3.1 Representation

We have built our distributed data structures with a process-oriented model. Processes hold data elements and communicate algorithmic operations among them. This model creates processes that behave as active elements that perform dynamic tasks on the data stored in them. Each process is designed to contain:

- the value or payload associated with the element in the process,
- the key as index to this element value,
- pointers to successor processes, and
- the minimum key-values (min-keys) of each successor processes.

Element values are distributed across multiple processes across different cores and machines. Key-values help in the search process for element values. Elements are stored in sequential order based on key-values. The identities of successor processes are stored to be able to redirect incoming messages or
operations when necessary. Similarly, the key-values of successor processes
are stored so that processes can look ahead before passing messages, so as
to know when a message is required and when it can be avoided. This serves
both as an optimization by avoiding communication when possible, and also
a key correctness factor by helping avoid deadlock scenarios.

We have modelled our communications to initiate messaging in the for-
ward direction only. This contributes to avoiding deadlocks by not creating
cycles in the route of a query. A message may be sent backwards only as
a reply to an already initiated forward message. A process knows when to
expect a backward reply message by using its look ahead key. Since com-
munication is initiated in the forward direction only, we can avoid recording
identities of predecessor processes because MPI semantics provides mecha-
nism to identify the sender of a message. Service operations on data items
are performed with message passing semantics; operations are passed from
one process to another usually starting from a dedicated head process, or
through a shortcut process. Every operation message contains:

- an operation identifier,
- identity of the source process who generated the operation,
- operation sequence number based on the source,
- the data item for the operation (when applicable),
- the key-value for the data,
- some other service and messaging flags, and
- some information attachments depending on the operation type.

The type of the request is marked in the operation identifier for performing
the correct sequence of actions. The identity of the source is kept along
with a message so that eventually when the operation completes or fails, the
proper reply back to the application can be provided. In addition, we in-
clude a sequence number with messages so that we can reorganize replies in
the order they were sent. Note that in a list structure, successive operation
requests for different data elements from an application process may execute at different relative times depending on the location of actions. Therefore, sequence numbers can help re-order reply messages at the application process. The data item is carried along with an operation message when the request type is an insert operation. If the intended use of our structure is to carry a large payload, for efficiency, the payload data can be detached from the insert messages and sent after the final destination is found. Some attachments and flags are kept for miscellaneous purposes when propagating messages. An example of use of flags is to mark whether a shortcut link (see Section 3.5.2) or a linked-list pointer was used when sending a message, so that we can check if a shortcut fails or not. More details are provided in Sections 3.2.2 and 3.3.2 when we discuss operations on our distributed data structures.

3.2 Linked List

Our first distributed structure is an ordered linked list. In general, in a sequential implementation of a linked list when a single process implements the service, the entire structure is used by one operation at a time. This makes access time very slow if multiple applications need to access a common list. Operations on a linked list travel along its pointer chains in a sequential manner, and data elements are contained in the same machine. There is a potential in this structure for parallel access by allowing multiple requests to hop from node to node. We use this principle in building our distributed linked list. We divide the data items among multiple processes, and these processes communicate operations on these data items. A process sends and receives operations through its neighbor processes in a way that mimics the behavior of a sequential linked list. The advantage of designing the linked list to operate in parallel is that by allowing multiple operations to work together on different sections of the structure, we are allowing simultaneous access and hence achieve higher throughput on the structure.

In the next section, we give the essential design choices that we considered for building our distributed linked list data structure. Section 3.2.2
3.2. Linked List

discusses the algorithms on the final model chosen.

3.2.1 Design

We designed our linked list to be a chain of processes linked one after another using their MPI rank identifiers. Figure 3.1 shows the contents of a linked list process. As mentioned in Section 3.1, each process contains its key and payload values, and contains pointers to the next process using an MPI rank together with the minimum key value of this next process. The rank of the predecessor of a process is not kept because it can be found from the MPI message envelope, especially because all messages to this process come from its predecessor process.

![Diagram of linked list process](image)

Figure 3.1: Contents of a single MPI process that corresponds to a single linked list element. Each process contains a key, a payload, a pointer to the next process and the minimum key value of the next process.

The linked list service starts off with all elements being free processes not yet linked to any other process. As we perform insert operations, processes will be allocated and linked into a chained sequence. The head process for the linked list is fixed to an arbitrarily chosen process.\(^4\) The head or root process is fixed throughout the service execution so as to avoid race conditions during service discovery. The root does not hold any key or data element, and acts as a guard process that is used for service discovery of our linked list service. We do not have a similar guard process for the end

\(^4\)Discovery of head process can be made automatic during initialization.
of the list, because using a sentinel value for the pointer to the next process suffices to know that the end of the list is reached.

Once a process is linked in the service, it acts as a relay for operation messages that travel from the head process to the tail process by means of MPI messages. Figure 3.2 depicts an example chain of linked list processes. Compare this to Figure 2.1 and notice how elements are now identified with MPI ranks, which means that data are distributed across different processes and may not be in the same machine. A process will act upon an operation message when the key for the operation matches that of the internal key (or set of keys in a multivalued variant) stored in the process. Further propagation of the operation terminates at the location where the key matches the internal keys, or when pairs of processes are found between which the operation key should match. Operations also terminate at the end of the list. In case of successful completion of an operation, which may require some intermediate computations and communications to other processes, the process where the final action is performed replies to the request originator (i.e., the source application). During operation failures or during early failure detections, the last process to hold the operation request is responsible for informing the originator.

We have modularized our service to different role players. This allows miscellaneous tasks such as load balancing or service discovery to be con-

\[\text{Figure 3.2: An example of a distributed chain of processes that make up a linked list service. Processes are pointed to using MPI rank semantics. The head process does not contain a key or payload, and a special sentinel pointer in the last element marks the tail of the service.}\]
3.2. Linked List

sidered in isolation. In FG-MPI, an OS process can be made to contain a mixed collection of functions that make it possible to have functions for different roles within a single OS process. The different role players in all of our services are (as introduced in Figure 2.3):

- Local Managers,
- Application Nodes or App-Nodes,
- Free Nodes, and
- List Nodes.

The role of a local manager is to manage service discovery and service redirection so as to maintain balanced request distribution (load balancing). Local managers maintain the free process list in an OS process and serves requests made by list nodes for new free processes. A local manager may contact another manager when its local free list finishes or earlier based on some load balancing strategy. Any service initialization or termination code is performed by local managers. Application nodes are processes that communicate with external applications and redirect external requests to the service. In other words, application nodes are the interfaces for external applications to connect to our distributed list service. In our experiments, to perform load testing we make the application nodes generate uniformly random requests. Free processes are the state of list processes when they have not yet been bound to an element value. A free node transforms to a list node after an insert operation, and goes back to a free node after a delete operation.

Every OS process must have at least one local manager process. More managers in the same OS process may be an option for advanced load distribution management, but this is left for future extensions. Application processes can be distributed across our service in an arbitrary way, but to make best use of parallelism, they should be spread to all OS processes. As a start, we put one application process in every OS process. After the manager and the application process, the rest of the processes in an OS process
3.2. Linked List

start out as free nodes and register themselves in a local free list which we call the bulletin board. Application processes are initiated with knowledge of the identity of the head process. The head process starts as a linked list process and is never deleted.

The processes that we refer to as application processes are actually gateways for relaying instructions from real applications external to the service. Having the application process ‘interface’ as a separate layer helps convert incoming external messages to requests understood by our service. Furthermore, the application process interface can also take part in the load balancing strategy. The interface can initiate and use different service parameters that modify overall service behavior, which can be invisible to external applications.

It may seem that a job of a local manager to find free nodes can be done by any list processes because the free node list is a shared resource in an OS process. However, if a free node has to be fetched from another OS process for example, this would require an arbitrary communication to a random list node. This has the potential to create a deadlock as the process being contacted may already have initiated a communication that waits for a reply with the process that requested the free node. Having a separate service responsible for handling free node search removes the possibility of having deadlock because local managers become dedicated for this purpose and the system can be designed without cyclic communication that leads to deadlock. Local managers can be extended to do extra work in between requests and employ different load balancing strategies.

In parallel programming in general, we know that the order of message delivery to a process from different predecessors is not guaranteed. Because of this, there is a potential for race conditions when propagating messages. If a request has to complete in parts and one part of its actions needs to be done from one process and the other part in another process, the relative order of operation for both parts needs to be maintained. There may be occasions when the latter part of a request, which is passed to a second process, may complete before the first part completes in a given process. This happens when both of the processes need to communicate to other
processes which are a varying number of hops away. Therefore, we cannot guarantee which part of the original request will happen first. To avoid these kinds of race scenarios, we have the option to wait for acknowledge messages on every hop. But this scheme of communication will not be able to fully harness the potential for parallelism as then we will not be able to overlap computation with communication. Instead, it is better to use asynchronous communication where messages are sent eagerly and buffered by the MPI middleware at the destination process. To make such eager messaging work, we use \texttt{MPI\_Ssend} for the last message in a multi-part operation between processes, which acts as a synchronization message. Similarly, when a process changes its neighbor, outstanding requests are completed with an \texttt{MPI\_Ssend} terminate connection message. With these measures, we can prevent race conditions, take full advantage of concurrency and easily and effectively overlap computation and communication.

### 3.2.2 Operations

The operations on the distributed linked list that we have implemented are \texttt{FIND}, \texttt{INSERT} and \texttt{DELETE} operations. \texttt{FIND} is used to search for a key in the distributed list and return the associated payload value stored. The \texttt{INSERT} operation is used to add elements to the list. \texttt{DELETE} removes an element if the key is found in the structure. An operation is passed from one node to another, until the position to perform the action is found.

In the following discussions, we assume that there are three arbitrary processes A, B and C that take part in the operations being discussed. Process A is considered to be the process who has already received an operation request and is ready to act upon the operation or relay the message to process B, the successor of process A. Process C is the third participant with varying roles based on the operation type. The same figure (Figure 3.3) is referred by all of the list operation discussions below.
3.2. Linked List

Figure 3.3: Implementation of the ordered-linked list operations (a) FIND, (b) INSERT, and (c) DELETE. Arrows denote the different communications among processes A, B and C (the three processes that participate in the operations).
3.2.2.1 FIND Operation

Figure 3.3(a) summarizes the steps taken to perform a FIND operation in the linked list service. Process A checks its own set of values first to find a match with the request search key. Assume that process A has done so and is ready to send the FIND request to process B. A then passes the request to process B (step 1) and forgets about the operation and moves on to receive the next operation. Process B then checks for a match of the search key in its local set of keys (like in process A). When B finds a match, it replies back a FIND success along with the associated payload to the application process that originally generated the FIND request. When B cannot find a match, it first checks the min-key value of the next node, if such a node exists. When the search key value lies after the current node but before the next min-key value, which means that the successor process can not have a match of the search key (the value cannot be found in the linked list service), process B replies back a FIND failure to the originator process. When the search key is greater than the next process’s min-key value, a match may be found in later parts of the list service, and the request is passed forward to process C, the next process after B.

3.2.2.2 INSERT Operation

Figure 3.3(b) depicts the steps needed for an INSERT operation. Assume that an INSERT request has reached process A, and that the insert key value should lie between process A and process B. Although the predecessor of process B is changing, process B does not need to know that the INSERT is being executed because we do not store information for predecessor of processes (see Section 3.1). Process A first requests for a free node from a manager process. Any manager process can be selected arbitrarily as the start of the free node search, however, in our current implementation, we provide knowledge of two manager processes when initiating a free node request; the local manager of process A, and the local manager of the application node that initiated the operation (i.e., source application). In the second option, the identity of the manager process is provided as auxiliary
information to the data structure queries (see 'attachments' in Section 3.1). The second option has advantage over the first because this effectively balances the selection of new free nodes as we evenly distribute application processes to all OS processes (see Figure 2.3).

Once a new free node (process C) is found, process A is given the identity of the free process as shown in step 1. In step 2, process A sends process C information regarding process B (the original successor of process A) with the min-key value and MPI rank of process B. Process C then becomes linked to process B, however, process A does not yet redirect messages to process C. To avoid race conditions, process A terminates its previous connection to process B by flushing all outstanding requests it has sent to process B before the INSERT request. This is done using an empty synchronous MPI_Ssend message as shown in step 3. Process A then connects to process C, and in the mean time, process C informs the request source application process of the success of the insert operation (step 4).

An insert operation fails when the free node search can not find a free process, in which case process A is responsible for informing failure to the application process that originated the request. If a duplicate value is found via the look-ahead min-key value, then again process A reports failure of the insert operation with a flag marked that indicates duplication as the reason for failure.

### 3.2.2.3 DELETE Operation

Figure 3.3(c) shows the steps for a DELETE request in our linked list service. Assume that process A has the delete operation and is ready to pass this to process B (step 1). Process A halts to wait for a reply from B when the min-key for process B matches that of the delete key, hence process A knows that it should receive a new successor information from process B. Step 2 shows that process B then flushes its connection with its successor process C via sending a terminate connection message. Process B then sends back the deletion completion acknowledge message to process A, with information about process C (step 2). Process A accepts process C as its
new neighbor in step 3. At the same time, process B informs DELETE success to the originator of the request. Finally, process B reverts to the role of a free node and attaches itself to the local free process list (Step 4). The final enqueuing of process B to the free process list is a local operation in its own OS process, and hence it can be done directly by the process itself without having to go through a manager process. This is possible because FG-MPI provides atomic access to shared OS process resources inside every MPI processes because FG-MPI uses non-preemptive scheduling of processes. When a delete key is not found between process A and the min-key in process B, process A then reports DELETE failure back to the application process.

3.2.3 Granularity

We have discussed the operations on our linked list service with a view of having a single data element per process only. An obvious extension to our ordered list is to allow each list process to store more than one item. We have introduced a parameter $G$, which is the maximum number of elements a process can hold, which we specify as the granularity of the service. As previously described each process stores its local set of keys and the minimum key of its successor. It is simple to modify the list process to perform a FIND, INSERT or DELETE operation on its local set of items first before considering successor processes.

The FIND operation requires the least change to support a set of elements in a process. As mentioned, the operation searches the local set of keys first before passing the request to the successor process. The INSERT operation is modified to insert elements in the local set of keys instead of fetching a new free node. This is done until the threshold $G$ is reached, after which another insert operation will cause the process to perform a SPLIT operation. The SPLIT operation is a new action created for supporting the list service with higher granularity. It essentially takes the role of asking for a new free node from the original INSERT operation (Figure 3.3(b) step 1). During a SPLIT, if a free node is absent, an INSERT failure is reported.
If a free node is found, the set of values in the split process is divided in half and the second half is transferred to the new process. This means in Figure 3.3(b), process A will share elements with the new process in step 3. A SPLIT success will not be reported as an INSERT success as it is a local improvement rather than an application induced operation. It is possible to use a parameter to randomize when to invoke the SPLIT operation so that a process can decide to add items to its own set or split its items between itself and a new process. This introduces another opportunity to add a load balancing strategy to attempt to optimize for the number of items to store locally. The possible inverse operation for a SPLIT where we re-combine processes can further optimize the load balancing in an adaptive way, however, we did not explore using such an operation as it may require initiating communication to a predecessor node which may give rise to deadlocks. We used the simple strategy to split the items whenever the number of items exceeds a fixed threshold.

For the DELETE operation, a value is directly deleted from its local set of items. When passing the request however, if the delete key matches the min-key stored in a process, this implies either the next process is to be deleted, or when there are multiple items in the next process, its min-key will change. Both these cases can be handled by the reply from B to A in step 2 of the DELETE operation shown in Figure 3.3(c). Therefore if the next process has no more elements, all the steps for DELETE remains the same as before, but if the next process has more elements, then step 4 is skipped and process B remains chained to the list service and informs DELETE success to the application process.

The granularity can be extended so to allow it to vary dynamically to adjust load among processes. It is possible to set $G$ initially to be small when the list is small and then increase it as the list size increases. Both $G$ and $P$ (see Section 2.4) are helpful in reducing the idle time. Increasing $G$ reduces the amount of messaging and therefore the amount of work done in performing requests. However, to maximize the opportunity for parallelism we need to distribute the list to the $O \times M$ processing cores.
3.3 Skip List

We now discuss how we extend the distributed linked list service to that of a skip list. In the sequential implementation (see Section 2.2), a skip list contains multiple layers of linked lists using towers of pointers to different successor elements instead of pointing to the immediate successor. Each linked list layer is a proper subset of the list below it. An element in one layer is inserted in the layer above it with diminishing chances using a Bernoulli process. This makes it possible to take advantage of the sparseness of higher layers to jump close to the destination when executing queries thereby improving the average number of hops needed to complete the operations. This means that a parallel and distributed implementation of a skip list can not only benefit from being able to stream operations (as our distributed linked list), but also it can do it in a more scalable way by being able to distribute load to multiple neighbor processes at different levels. All other benefits of distributing the linked list will also remain such as creating active elements that can overlap computation and communication.

Most of the modifications that are required to extend our linked list service to the skip list are in the algorithms. These algorithm extensions will be discussed in Section 3.3.2. We first discuss the structural and design extensions in the next section.

3.3.1 Design

Our first step was to extend the next pointer of the linked list structure to contain a group of next pointers in a leveled-tower arrangement as shown in Figure 3.4(a). Any process in the skip list service can now have more than one successor and predecessor processes. As shown in Figure 3.4 each skip list process is comprised of (a) a tower of \(<key, rank>\) pairs where each key is the minimum key (\(min-key\)) of the skip list process with that rank, (b) key (or list of keys in the multi-key case), and (c) the associated data. The height of the tower is probabilistically determined using a Bernoulli process when a process joins the skip list and goes from level 0 to some level \(k\) less than a predefined maximum tower height. Figure 3.4(b) shows the
3.3. Skip List

Figure 3.4: (a) Example of a skip list process with connections from previous and next processes, and (b) Message-driven main loop of the process.

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
3.3. Skip List

the skip list.

Figure 3.5 shows a small sample skip list with four processes and maximum tower height of four. As in a sequential implementation of a skip

On each level, as in the linked list, there is the minimum key value and rank of next MPI process.

![Diagram of a skip list with four levels and four processes](image)

Figure 3.5: An example of a skip list where each list item is a process.

list [27] 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 3.5 each node in the skip list is an MPI process which is identified by its MPI process rank. 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. The tower of pointers allow us to first search at high levels that gives ability to quickly skip close to the final destination.

The skip list service is built from the same principles as our linked list implementation. There is a fixed head process in the service that does not hold a value and never gets deleted. The head process has to have a tower height equal to the maximum tower height among the skip list processes. The head process can be congested with many requests, but the load is

---

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.
quickly dissipated. This is because the head node of the skip list has the highest tower and it contains many shortcut links to other processes. Requests are further dissipated after every hop to a process. A query can jump close to its destination by starting from the higher levels. As a result from its basic structural properties, the skip list makes communication overhead balanced across the service.

Because processes are linked in multiple levels, operations have to be modified to maintain the correctness of the service structure. For instance, an INSERT operation is no longer just an addition of a new process in between a pair of processes. The new node must now be inserted between multiple pairs of processes depending on its random height. Similarly, for DELETE, process tower entries have to be removed from all levels. As a result, we should view operations on the skip list service as a process of searching, building or removing link towers. Our operations are modified to do multiple actions along their search paths. Operations are done in parts by the processes whose towers need to be modified. In other words, an INSERT operation will insert parts of the tower of the new node as the request travels along the skip list service. The same is true for DELETE, which trims part of a tower as the DELETE request traverses the service.

3.3.2 Operations

In the following sections, we discuss the operations for the skip list, and assume that we have four processes A, B, C and D with their own tower heights. The discussions assume that an operation has travelled to a certain level, and that at this level, process A is the current node with the operation request, and process B is the next node pointed to by the tower entry in process A at the this level. Process C and D are other processes whose roles vary depending on the operation being discussed.

3.3.2.1 FIND Operation

Figure 3.6 shows the steps in performing search for a given key in our skip list service. The cross-patterned arrows can be visualized as the route followed
by a query for keys greater than 43, which is the second lowest entry in the
tower of process B. The overall process is similar to the sequential algorithm
except that we are now passing messages between different processes. The
search is started from the highest tower of a given receiver process. On every
step, a comparison is done between the search key and the look ahead min-
value of the next process at the current level. If the next process provides a
useful jump, which is true if the min-key is less than or equal to the search
key, the request message is relayed to this next process and the current
process proceeds to receive new queries. However, if the min-key value is
greater than the search key, the current level is decremented and the same
process is repeated at the level below. These steps are done until we reach
the lowest level at some process after which we report success or failure
to the generator application. The INSERT and the DELETE operations

Figure 3.6: Steps taken to perform the FIND operation in the ordered skip list.
The cross-patterned arrows illustrate how a search query travels along the skip list.
Shaded arrows represent messaging and thin arrows represent connection links. At
any point in the search route, a decision is made whether to skip forward or to go
down one level until the search key is found.
3.3. Skip List

also search for the first process from where to start execution in a pattern similar to FIND as shown in Figure 3.6. The FIND operation is summarized in Listing 3.1 where the process either replies to the client (Steps 3-6), forwards the request, or drops down to the next lower list.

Listing 3.1: FIND Operation

```plaintext
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 }
```

Because in the tower min_keys are local, it generates either one reply or one request. The ∞ marks (or 'NA') in the towers correspond to ends of the linked lists at the levels where they appear. We use a sentinel to mark this, and a request can only move down a level when this mark is present.

3.3.2.2 INSERT Operation

The process to be inserted or deleted from the skip list service needs to either link or unlink itself from the skip list as a request traverses the list. The basic link and unlink operations are shown in Figure 3.7. For an INSERT operation, depending on the height of the tower being inserted, we need to update the min-key values and the next process pointers as we traverse the service. For example, in Figure 3.7, process C is to be inserted at a location between A and B which requires A to send [y, B] to C. For a DELETE, we need to unlink the node to be deleted from the list, C, which will require A to send a message to C and C to reply back with its information about B.

Figure 3.8 shows the individual steps in performing INSERT in the skip list service. As mentioned before, the operation corresponds to the insertion
3.3. Skip List

Figure 3.7: 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.

of a tower between multiple processes. Therefore, the process is executed in parts, building the final tower level by level. As an optimization, the same predecessor and successor pair that may appear on a continuous number of levels of the skip list is handled with a single message (as opposed to sending separate messages on each level). The tower height for the new node is determined in advance when the INSERT request is received. The tower height adds a constraint when deciding whether to move forward or go down a level during propagation of an INSERT operation. The INSERT operation is executed in the following manner:

- By following the search path mentioned in Section 3.2.2.1 and when the current level is equal or below the decided new process tower height, we find the first process in which to perform the INSERT operation. This is marked as the shaded region in the tower in step 1 of Figure 3.8. As this is the starting point of action, process A triggers a free node request just as in our linked list service.

- When a free process, C, is found, its rank is sent back to process A. A will then augment the identity of C along with the INSERT request so that when the rest of the tower of process C is constructed, a new free process search is not performed and information is passed to C.
3.3. Skip List

Figure 3.8: Steps taken to perform the INSERT operation in the ordered skip list. Shaded arrows are communications; thin arrows are link connections. M denotes a manager process, while C denotes the free process. The figure shows only segments of the towers of all processes, hence all processes need not be immediate successors as may be apparent from the figure.

- Process A then sends a terminate connection request to process B in step 2 as in our linked list service. A single terminate connection message is provided for all levels where C lies between A and B. This terminate connection allows us to flush pending requests from A to B.
3.3. Skip List

for the levels pertaining to the request.

- In step 3, process A then sends an INSERT-completion request to process C. Process C knows to wait for such a message after it is selected in the free node search service. The INSERT-completion message sent to C from A is given a different operation identifier which we call BIND.

- When process C receives a BIND message, it connects to B at the given levels. Now the tower of C is built partially only up to the current level of the INSERT operation. Process C responds only to BIND messages until it has built its tower completely, after which it behaves as a regular skip list process. This ensures that no subsequent requests can overtake the linking to process C.

- Finally in step 4, since the INSERT request happens in parts, the request is passed along the level on which the INSERT was just performed (passed to process D). When the lowest level is reached, process C will have completed building its tower and it reports success to the application process. This is the point when the propagation of the INSERT request stops.

An INSERT request fails if a free process cannot be found, or if a duplicate exists. Absence of free process can be reported as early as step 1 in Figure 3.8. However, detecting duplicates early before having built parts of the tower for the new process is not guaranteed. If the new tower is shorter than the duplicate we report failure as soon as we encounter the duplicate. But if the new tower is taller, we replace the existing duplicate by completing the INSERT operation before generating a DELETE request for the duplicate whose tower is now in the shadow of the newly inserted process.

The INSERT operation is summarized in Listing 3.2. The operation has 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. 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 list_full back to the application.

Listing 3.2: INSERT Operation

```
INSERT(request, newprocess, level)
{
  if ( level <= request_height && request_key < key[level] )
    link[key,level,newprocess];
  if( level==0 )
    reply success;
  else
    INSERT(request,newprocess,level-1)
  else if( request_key == key[level] || request_key == local_key )
    reply duplicate
  else if( request_key > key[level] )
    forward request;
  else
    INSERT(request,newprocess,level-1)
}
```

### 3.3.2.3 DELETE Operation

The DELETE operation is the inverse of INSERT but the basic propagation mechanism is similar. The first position to operate upon is found via the stair case like descent outlined previously. Figure 3.9 shows the steps in the DELETE operation. The operation proceeds in the following manner:

- In step 1, process A sends a DELETE action request to process B for the levels where process B is the next process to process A.

- Step 2 requires process B to send a terminate connection request to process C, the next process of B. There may be multiple successor processes in the levels for a DELETE operation, in such cases, this step and the next are repeated for each unique successor. Process A expects the multiple replies for each unique successor and stops receiving when all the associated levels are accounted for.
In step 3, B removes its link to C, and sends back information about C to process A.

In step 4, process A connects to process C in the associated levels.

Just as in the case of INSERT, the DELETE operation progresses by
3.3. Skip List

unlinking or trimming the tower of the node to be deleted by parts. Therefore, the remaining parts of the DELETE operation is passed to the next process below the level already unlinked, which is to process D.

- Finally, process B reports the DELETE success to the application, as well as enqueues itself back to the local free list when its entire tower is unlinked.

When only a part of a node’s tower is unlinked, the node to be deleted still behaves as a normal skip list process, processing other requests as if it will not get deleted. Process B does not receive further messages at the unlinked levels because no more processes have pointers to B at these levels. The entire tower of process B eventually gets completely deleted when the DELETE operation reaches the bottom most level at some future process. This is when the propagation of the DELETE operation is halted.

The DELETE operation is summarized in Listing 3.3. This is the only operation that needs to perform an unlink operation which requires a process to send back information to a predecessor in the list.

Listing 3.3: DELETE Operation

```c
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  }
```
3.3.2.4 RANGE-QUERY Operation

In general, both linked list and skip list services can perform a range query because the elements are ordered. The skip list service has the benefit of performing range queries in parallel by dividing a query into smaller sub queries to the multiple successor nodes at different levels. The propagation of each query or split sub query resembles that of a FIND; but whenever there is a decision on whether to go forward or down (see Figure 3.6) the range query has the following four options:

- when there may not be an instance of the range from current node to next process, send the entire query forward and change the start of the range to next,

- when there may not be an instance of the range from next process to end of range, send the entire query down the level and change the end of the range,

- when the next process can split the range and both splits can have an instance, divide the range query and update the start and end of the split ranges, and

- terminate the query if none of the above is true.

Splitting range queries means that an application can expect multiple replies for a single range query. Application processes know when to stop waiting for replies for a single range query when the complete extent of the range is returned. This is possible as each range split divides the overall span of a query, and the application can match incoming replies with the extent covered so far. This method works for multiple dimensional ranges also, but with higher dimensions, the range is non-contiguous and as a result, more reply messages are generated.

The RANGE-QUERY routine is summarized in Listing 3.4. Steps 11–13 splits the query when it is possible. Every skip list process with data within the range specified in the query will reply to the application.
3.3. Skip List

Listing 3.4: RANGE-QUERY Operation

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

In the multi-key per node case, the number of replies 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.

### 3.3.3 Granularity

Just as in the linked list service, it is possible to extend the skip list service to have processes manage multiple list items. Again, the FIND operation is straightforward to extend, in that the local set of keys need to be searched before passing to a neighbor process. Since a RANGE-QUERY is essentially an extended FIND operation that can be split while traversing the list, it can also be easily extended to match the range query on the local set of keys. A consequence of increasing granularity is that we can expect variable sized replies from processes back to applications for range queries. We use a double messaging approach to handle variable sized reply, where the first reply contains the size of the results and the second message delivers the
3.3. Skip List

actual results.

The extension for the INSERT operation is similar to that made for the linked list service (in Section 3.2.3); it is performed on the local set of keys only until a predefined threshold for granularity is reached. The node split procedure for the skip list can have two cases, after a free node is fetched from a manager process then:

- if the randomly determined tower height for the new node is equal to or less than the tower height of the node to be split, then the tower is created and all neighbor information is shared before passing half of the keys and payloads, or

- if the new tower height is higher, then an INSERT operation similar to the single item skip list INSERT (see Section 3.3.2.2) is performed to place the new node tower first before sharing keys and payloads.

An INSERT operation therefore now only adds elements to the local set of keys. The splitting of a process is the operation that builds the skip list towers. This splitting process (named SPLIT) is separate from the INSERT in that it is automatically invoked in processes as an optimization to dissipate load. It should be noted that second case for SPLIT (as listed above) has to build a skip list tower again in parts. The SPLIT operation has to traverse through multiple processes before it can finish. During this time, the node that generated the SPLIT cannot wait for the operation to complete as any outstanding messages from its predecessor would be halted. As a result, the node being split is marked to have invoked a SPLIT operation and moves on to operate on other requests. This is possible because, as mentioned, the SPLIT operation is an optimization and not the last step of an INSERT. While marked, a node will not request further SPLITs, and the programmer has the option of not inserting any more values until the split finishes or allow the node to grow larger than the threshold for granularity. It also can be the case that by the time the SPLIT operation completes, the entire original process that generated the SPLIT request gets deleted. In such a situation, because we allow communication initiation in the forward
direction only, we convert the new split node to a zombie process that relays messages and unlinks itself automatically as requests come to it at different levels. Eventually the zombie process will become a free node again.

For the DELETE operation, when the min-key matches the delete key, then either the next process becomes empty and is unlinked revealing a new neighbor process or the min-key needs to be updated keeping the same existing neighbor process. In the second case, we have the potential for a race condition. This is occurs because the DELETE operation is done in parts, and a process updates only the min-key value for its neighbor before relaying the DELETE operation to another node. As a result, subsequent messages from this process to its updated neighbor may overtake the remaining parts of the relayed DELETE operation. To avoid this problem, the DELETE reply from the successor process whose min-key is about to change is held back until the entire DELETE request is received.

3.4 Proof of Correctness

Since the skip list is a super set of the linked list service, the proof of correctness presented for the skip list will suffice to prove the correctness of the linked list service. In the following sections, we prove the correctness of our implementation first by reiterating how our system is free from deadlock and then present the order guarantee of operations on the skip list.

3.4.1 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
3.4. Proof of Correctness

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.

3.4.2 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 ordering, sequential consistency, and no consistency semantics described in Section 3.5.

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 \( R = r_1, r_2, \ldots, r_k \) be the list of request messages received by the processes.

As given by the listings in Section 3.3.2, requests are atomic and all messages between processes can be realized 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
3.4. Proof of Correctness

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 \(r^a_i \in R^a\) and \(r^b_j \in R^b\), let \([r^a_i \tr r^b_j]@P\) denote that both \(r^a_i\) and \(r^b_j\) are executed by process \(P\), and receive \(r^a_i\) executes after receive \(r^b_j\) in process \(P\).

**Definition 3.4.1** Let \(\succ\) be the equivalence relation defined on the set of receive events \(r_i \in R\) for a collection of operations \(R\) on process list \(P\).

Define \(r_i \succ r_j\) (\(r_i\) causally depends on \(r_j\)) if and only if (a) \(r_i, r_j \in R\) for some \(R\) and \(i > j\), or (b), for some process \(P \in P\), \([r_i \tr r_j]@P\).

**Lemma 3.4.2** Given two operations \(R^a\) and \(R^b\) on process list \(P\). If, for some \(P \in P\), \([r^a_i \tr r^b_j]@P\), then for all \(k\), such that \(r^a_{i+k}, r^b_{j+k}\) exist, \(r^a_{i+k} \succ r^b_{j+k}\).

**Proof** Assume to the contrary, take the first occurrence such that \(r^a_{i+k} \succ r^b_{j+k}\) but \(r^a_{j+k+1} \succ r^b_{i+k+1}\). Given we have a process list, since \(r^a_{i+k}\) and \(r^b_{j}\) both occur in process \(P\), it follows that \(r^a_{i+k}\) and \(r^b_{j+k}\) occur in the same process. As well, \(r^a_{i+k+1}\) and \(r^b_{j+k+1}\) both must occur in the next process in the list. The relations \(r^a_{i+k} \succ r^b_{j+k}\) and \(r^b_{j+k+1} \succ r^a_{i+k+1}\) forms a crown, which implies that the messaging cannot be realizable as a synchronous communication [24], which contradicts that the communication is synchronous.

Lemma 3.4.2 captures the property that operations in the list are performed by processes in order and cannot jump ahead of one another. Also, with respect to shortcuts, Lemma 3.4.2 implies 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 lemma.

**Lemma 3.4.3** On a skip list with operations \(R\) and processes \(P\). If \([r^a_i \tr r^b_j]@P\), then either (a) If \([r^a_{i+1} \tr r^b_{j+1}]@P\) or (b) for \(m, n > 1\), \(r^a_{i+m}\) and \(r^b_{j+n}\) do not visit the same process.
3.4. Proof of Correctness

**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 3.4.2. Assume true for a skip list of maximum height \( h - 1 \) and now consider a skip list of height \( h \). Consider two operations \( r^a_i \) and \( r^b_j \) 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 \( r^a_i \) and \( r^b_j \) 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 \( r^a_i \) and \( r^b_j \) 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 \([r^a_{i+1} \triangleright r^b_{j+1}]@P\) by Lemma 3.4.2. 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 \( m, n > 1 \), \( r^a_{i+m} \) and \( r^b_{j+n} \) 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
3.5 Service Interface

An important part of the design is the implementation of the service interface. As introduced before in Section 2.4, we created processes that connect to external applications, which we call “application processes”. They form a separate layer on top of our skip list service. Application processes first need to know how to discover the skip list service. We configure the application processes with a pointer (i.e., MPI rank) to the head of the list. Although the list communication is deadlock-free, the composition of the list with the application processes introduces the following deadlock scenario. It is possible for there to be a communication cycle starting from (a) the application process making the request, to (b) the list nodes traversed, to (c) the final list process that sends the reply message back to (a). The message interface to the service must enforce the condition that the application be ready to receive all replies whenever they have outstanding requests to the service to avoid a deadlock. There are also important performance consequences to receiving replies quickly, since at steady-state the in-flow of requests equals the out-flow of replies; each new request depends on the receipt of a reply. The application processes are also given hints to shortcuts during service discovery to speed up the process of delivering operations to their final destination. Shortcuts are further discussed in the next section and Section 3.5.2. The other key service layer previously mentioned that ensures proper operation of our skip list service is the free process sub-service, which is discussed in Section 3.5.3.

3.5.1 Applications Processes

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 initialization.
3.5. Service Interface

The application processes access the service by sending requests to skip list processes. Skip list processes reply directly back to application processes. 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. 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 [10] 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 3.4.2 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 linearization 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.

3.5.2 Shortcuts

In FG-MPI, all co-located processes share the same address space and it is easy to coordinate access to a shared structure because processes are scheduled non-preemptively. Therefore we can take advantage of the tower information of the co-located processes as a source of potential shortcuts into the skip list. The skip list processes can post keys, ranks and tower heights for the application process to consider as alternative skip list locations to send a request. For example, if an application process is performing
3.5. Service Interface

a \texttt{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. The use of shortcuts does not destroy the existing behavioral properties of our skip list service. The use of a shortcut is valid as long as we jump to a node which contains a value less than that of the request key.

Shortcuts remove the head as the bottleneck but at the expense of linearizability. 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:

\textit{Total Ordering} :
All application processes send requests to the head.

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

\textit{No Consistency} :
Both the application processes and the skip list processes use shortcuts when possible and there is no guarantee about operation 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.
3.5. Service Interface

Shortcuts are only hints, as they may not be valid when operations eventually reach them (for example, when a shortcut is deleted by its predecessor). A failure is detected as soon as we discover that we are pointing to a free process or to a process with a larger key. On failure, we notify the application process, which can send the request to the head to ensure it will succeed. We are also able to turn off the use of shortcuts for a particular request and an application can re-submit a request with shortcuts turned off.

There are other ways in which shortcuts fail. For an INSERT or DELETE the height of the shortcut may not be sufficiently high to ensure there are no connections between nodes on levels above the shortcut. We can detect these situations whenever a shortcut leads to a process that points to a greater tower height than its own tower.

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 Lemma 3.4.3 and introduces a race condition between the operation taking the shortcut and other operations. However, the first operation to reach the tower wins and then its order with the other operations is preserved for the rest of the service.

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. Sequential consistency can be maintained using shortcuts without climbing and finally when climbing is permitted there is 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.

3.5.3 Free Process Sub-service

We introduced manager processes for the purpose of free process allocation and finding free processes from other OS processes. The advantage of having separate service responsible for free node allocation is that it can employ
3.5. Service Interface

different strategies to fetch free processes, to change the overall load balancing in the service. Manager processes can communicate between each other to find the best route for free process search in our list service. We call this cooperative procedure a free process sub-service, which is stacked on top of our list service through FG-MPI dynamic function mapping. Managers also help in service initiation and termination procedures.

We take advantage of FG-MPI’s ability to expose large-scale concurrency to create a pool of free processes at the beginning of the program. This pool of free processes can be used to dynamically allocate and deallocate processes in response to the application load. A free process is reincarnated as a list node process in case of an INSERT operation and released back to the pool on DELETE. This can be viewed as spawning of processes on a fixed namespace. The free processes are all blocked waiting for an allocate request and do not impose an overhead on the scheduler (see Section 2.4).

At the start of a \([P, O, M]\) execution for our list service, three different types of MPI processes are launched per OS process; one of them is an application process, the second is a manager process and the remaining \(P - 2\) processes are list processes. Under this mapping the maximum size of the list is \((P - 2) \times O \times M\) with \((P - 2)\) nodes per OS process. Initially the ordered list processes are enqueued in a local pool of \((P - 2)\) free processes.

Other mappings are possible and the only constraint is that whenever an OS process has an ordered-list process there must also be a manager process. This allows us to define the size of the list as well as control the distribution of list nodes to OS processes to balance the communication and work between co-located processes (interleaved) and non co-located processes (parallel).

Managers use a predefined load-balancing strategy whose goal is to evenly distribute list processes to OS processes while trying to keep communication local between co-located processes. The trade-off between these two conflicting goals depends on the distribution of keys and overall workload and thus should be configurable. In our implementation, manager processes first attempt to satisfy free node requests locally. If there are no local free nodes, then the manager passes the allocation request onto the next manager where all managers have been configured into a ring and each knows
the next manager in the ring. The manager who succeeds in allocating a
free node replies directly to the list node making the request, allowing the
associated INSERT to complete. If no free node is found by any of the
managers after one round robin cycle, then the local manager signals failure
back to the list node who made the request. In turn, the list node notifies
the application process that the INSERT operation failed because the list is
full.

The cycle in communication of manager processes creates a deadlock
potential, but this is avoided by ensuring that each manager posts a receive
buffer for receiving a message before attempting to send a message to the
next manager in the ring. Because the receive buffer is always pre-posted,
MPI middleware can always deliver a message sent by one manager process
to the next manager in the ring.

The free node request can be initiated from any manager, however, list
nodes are currently not made aware of the identities of the managers except
their own local managers. If we select a local manager first, this produces
a strategy where the local free list is favored first, and the distribution in
selecting new free nodes is not spread out throughout the service. It also
means that neighbor nodes are more likely to be in the same OS process. To
improve on load distribution, we include the rank of the manager local to the
application process that sends an INSERT request along with the message,
and this is the first manager selected for the free node search. Since we
would ideally distribute application processes across all OS processes, this
effectively distributes the selection of free nodes across our service. As well,
requests are more likely to be served locally to the application process.

In the case of a DELETE, the free node registers itself back on the free
node list. As described in Section 3.5.2 locally we maintain a look-up table
that processes can post to and consult. This relieves the local managers of
the role to add processes back to the free list. This does not create any
race conditions, since FG-MPI processes are non-preemptive threads, which
implies atomic access to shared resources.

Load-balancing is a good example of a strategy that impacts performance
and needs to be configurable. The cost can depend on the workload, the
relative cost of communication between local and non-local processes, the
distribution of keys, and also the size of the list and $P$ (the number of local
processes per OS process). An advantage of this service-oriented approach
is that load-balancing code is not scattered through the program but in
one place and adding a new strategy only requires extending the manager
processes inside the free node service.

3.6 Parameters

The service interface to our implementation exposes parameters that can
be controlled easily during the start of the service or while in operation.
Because of the modularization in the service layers, the configuration of
parameters in the code is straightforward and allows for fine tuning per-
formance on the systems and networks used. An external application has
the option to set desired parameters in the application process. The be-
behavior of parameters depends on the application targeted and its workload.
In the following sections, we introduce the useful subset of parameters that
is available to applications to improve the fluidity of messaging in the list
service. We have already seen one parameter in the previous discussions
(Sections 3.2.3 and 3.3.3), which is the granularity of the service. All the
currently available parameters at run-time are listed with brief descriptions
and their default values in Appendix B. The effects of some of these param-
eters will be evaluated in Chapter 4.

3.6.1 Flow Control ($W$)

We define a configuration parameter $W$ in the application process to control
the flow of requests to the list. $W$ is the maximum number of outstanding
requests to the list service that an application process can provide at any
time. Let $cw_i$ denote the current number of outstanding requests (i.e., num-
ber of requests minus number of replies) from process $i$. To avoid deadlock
and improve performance, we require that whenever $cw_i > 0$, the applica-
tion process must be able to receive a reply. The sum of all $cw_i$’s at time
3.6. Parameters

$T$ is the total load on the service. In our experiments, we try using a fixed $W$ for all applications which proved to be the best option for the linked list service, and also made $W$ be modified adaptively based on the changes in the flow of operations from an application process, which was better in balancing load in the skip list service. In the adaptive configuration, $W$ varies the rate of requests to reach a steady-state where globally the flow of requests to the service matches the flow of replies from the service. $W$ cannot be set arbitrarily high, particularly when all requests go to the head of the list, since the saturation point tends to occur towards the head of the list and there are overheads in the middleware that increase, which result in decreased throughput.

3.6.2 Asynchronous Communication (K)

MPI supports both synchronous and asynchronous communication. In most MPI implementations, short messages are sent eagerly and buffered at the destination until matched by a receive. MPICH2, and as a result FG-MPI, has a global buffer pool that allocates space for eager messages in addition to other dynamic MPI objects. There is no flow control on eager sends and MPI simply aborts the execution when the buffer pool in the middleware of an OS process is exhausted. There is no simple technique for detecting these types of buffer problems in MPI, but it is possible to avoid them without resulting in completely synchronous communication \cite{6}. To avoid these errors we implemented a bounded asynchronous scheme where for every $K$ standard MPI Sends (potentially eager sends) we send a synchronous MPI Ssend() to ensure that there are no outstanding matching messages between the two processes. The flow of messages in the skip list is deterministic and the value of $K$ does not affect correctness, provided $K$ is less than available buffer size. In the experiments in Chapter 4 we maximize this value, however, for the description of the system it will be easier to consider $K = 0$, the synchronous case. The synchronous case is interesting because it is one technique to test for potential deadlock situations that are masked when buffering is available. Because MPI cannot guarantee the amount of buffering available
for asynchronous communication the MPI standard defines a safe program as one that can execute synchronously. However, it is important to increase $K$ sufficiently (with diminishing advantage, see Section 4.1.3) to overlap the computation and communication, when there is sufficient space in the buffer pool. This is all the more important in FG-MPI because there are more MPI processes sharing the middleware’s buffer pool and thus a greater chance of exhausting it.

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 queuing 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 busy.
Chapter 4

Evaluation

We evaluate the performance of our list services by measuring throughput in terms of the number of operations that can be processed per unit time. We show how this performance varies when changing a parameter in the service while keeping the other parameters fixed. In some of the experiments, we keep a property of the service fixed, for example the total length of the list. In all of the experiments, we report the steady-state throughput of the ordered list. The INSERT operation is initially used to build the list to capacity and then different workloads are used to measure the throughput.

For the experiments, the test setup consisted of a cluster with 25 machines connected by a 10GigE 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.

The experiments for the linked list service are reported in Section 4.1 and for the skip list service are shown separately in Section 4.2. The initial configurations are listed in these sections as well. Since the parameter space of the implementation is very large, it is difficult to exhaustively cover all parameter variations. Instead, we report on the interesting parameters only and with a workload that is representative of the general trend in the behavior of our list services.

4.1 Performance of Distributed Linked List

In this section we experiment with different parameters of the ordered linked-list service and show how varying them affects the performance. Recall that operations on a linked list are inherently non-scalable and our objective here
4.1. Performance of Distributed Linked List

is to demonstrate the working of a service-based approach and the flexibility in tuning of its parameters.

4.1.1 Consistency Semantics

Figure 4.1 shows the throughput achieved with the three consistency semantics of the distributed linked list described in Section 3.5.2, i.e., total ordering, sequential consistency and no consistency. The size of the linked list was set to $2^{20}$ in this experiment and the number of list nodes were evenly distributed over the number of cores $(O \times M)$, by setting $\frac{ListSize}{(O \times M)} = P \times G$, with $P$ set to 10. As expected, the throughput with no consistency is the highest followed by that for sequential consistency and then total ordering. In Figure 4.1 as we move from left to right along the x-axis, the list is becoming more distributed and the number of items being stored per list node is decreasing, while keeping the list size constant. The no consistency semantics benefits from more distributed nature of the list because the requests are spread over more cores. One could argue that, in the probabilistic sense, selecting the optimal shortcut for the no consistency semantics takes
4.1. Performance of Distributed Linked List

a maximum of $\mathcal{O}(O \times M)$ steps. Operations with the other two types of semantics i.e., total ordering and sequential consistency have to traverse the list to reach the correct list node and on the average take $\mathcal{O}(O \times M \times P)$ steps assuming $G$ is small enough to neglect its cost. Total ordering also suffers from congestion in the head of the list, which is why its curve is, as expected, the worse of the three.

4.1.2 G versus P

Figure 4.2 shows how performance changes when interleaved concurrency ($P$) is increased while reducing granularity ($G$), keeping the list size and the number of cores fixed. Since the number of cores is fixed, the overall amount of parallelism is fixed and the figure shows the trade-off between interleaved concurrency versus coarser-grain processes with more list items inside a process. Figures 4.2(a) and (b) show the trade-off between $G$ and $P$ with sequential consistency and no consistency shortcut semantics. In Figure 4.2(a), as expected, it is better to reduce $P$ and increase the granularity, $G$.

Parameters $G$ and $P$ define the first two levels in the communication hierarchy of the machine. In terms of traversing the list, from fastest to slowest, there is traversal (a) inside a process, (b) between processes inside the same OS process, (c) between processes on the same machine, (d) between processes on separate machines. There is a significant difference in communication costs between these levels (a) tens of nanoseconds, (b) less than a microsecond, (c) a few microseconds, and (d) on the order of 50 microseconds. Making $G$ large and $P$ small is analogous to keeping all data in the primary cache rather than the secondary cache. Figure 4.2 is a simple illustration of the trade-offs.

There are interesting questions that arise with regards to efficiently managing communication costs across the communication hierarchy. On a list of size $N$, assuming uniformly distributed keys, we have that $N = G \times (P \times O \times M)$ where the expected number of hops is $\frac{1}{2}(G + P \times O \times M - 1)$. With regards to latency (response time) the optimal distribution is to have
4.1. Performance of Distributed Linked List

![Diagram](image)

(a) Sequential Consistency.

(b) No Consistency.

Figure 4.2: Effect of changing \( P \), while keeping \( P \times G \) constant. The list size and the number of cores are fixed. Semantics used are (a) sequential consistency and (b) no consistency. (ListSize = \( G \times P \times O \times M = 2^{20} \) and cores = \( O \times M = 176 \)).

\( P = 1, \ G = N/(O \times M) \) and map OS processes consecutively from one machine to the next to keep communication as local as possible. But for throughput, increasing \( G \) limits the rate at which requests enter the system, since when viewed as a pipeline the search inside the list of size \( G \) at the
4.1. Performance of Distributed Linked List

FIND location delays all requests behind the current one. Although this depends on the location of the FIND, in steady-state it is likely to occur often and have more of an effect closer to the head of the list where the load is higher. Intuitively, larger $G$ reduces the capacity of pipeline and speeds up requests while increasing $P$ increases the capacity of the pipeline but slows down requests. We believe the value of $G$ in Figure 4.2(a) was never large enough to see this effect and provide a rationale for increasing $P$ in the sequential consistency case.

The throughput shown in Figure 4.2(b), with the use of shortcuts inside the list of processes, is significantly larger than that of Figure 4.2(a). $P$ determines the number of possible shortcuts inside each OS process. Once there are more than a few shortcuts there is a high probability of the existence of a shortcut to a closer OS process. As a result, the expected number of hops begins to depend on the number of OS processes rather than the size of the list. There is a diminishing return for adding more shortcuts since there is less chance of it being of value. As Figure 4.2(b) shows the first 100 shortcuts increase performance after which more shortcuts is adding more overhead than benefit as $P$ increases.

4.1.3 $W$ and $K$

Parameter $W$ specifies the number of outstanding requests that each application can submit to the list service. Increasing the value of $W$ increases the load on the system. $K$ is a throttling parameter that each list node uses to specify how many requests can be forwarded down the list, before having to wait for a previously forwarded message to complete. In the perfectly balanced case, when $W$ equals $P$, each list node receives exactly one application request. When $W$ exceeds $P$, each list node progresses multiple application requests up to a maximum of $K$. The smaller the value of $K$, the higher the throttling effect on the flow of requests through the list service. Figure 4.3 shows the effect of different values of $K$ on the throughput. The effect of small $K$ clearly limits the throughput. Increasing $K$ increases fluidity in the network for request propagation. There is a limit to the rate
4.2 Performance of Distributed Skip List

In this section we evaluate the performance of our skip list with regards to different consistency semantics and on various workloads with a focus on range-queries. Because the skip list service is expected to perform better than the linked list service, we test for stronger scaling by not keeping the list size fixed and instead we keep granularity constant even if that would mean that the list size will grow with the increase in the number of cores used. For the experiments we vary the list size from $2^{20}$ (over 1 million) to $2^{24}$ key-values.

All experiments were of the form $[P = 100, O = 8, M]$ with $M$ ranging from 2 to 25. We arbitrarily set $P = 100$ except when we measure the

![Figure 4.3: Throttling effect of $K$ on the throughput while increasing the outstanding requests ($W$) by the application processes. List size used is $2^{20}$. $P = 100$. $O \times M = 192$ cores. Sequential consistency semantics are used in this experiment.](image-url)
4.2. Performance of Distributed Skip List

effect of varying $P$. We set $O = 8$, the number of cores per machine, so as to minimize the effect of the OS on the execution. We start at $M = 2$ to ensure all of our tests include TCP traffic. We set the granularity $G = 1000$ so that $G \times P \times O \times M = 2^{20}$ when $M = 2$. For each added machine ($M = 2$ to 25), one application process per core ($O = 8$), one manager process per core, and 98 skip list processes are created, thereby increasing the list size by $98 \times G$. Thus list size varies from $2^{20}$ to $2^{24}$ as $M$ varies from 2 to 25. We do not include the single machine performance in the figures because the focus is on scalability and the potential for the system to use hundreds of cores with a view to its scalability to larger machines.

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 allowed by an application process. Application processes post receive-buffers for every outstanding request, and re-post requests as soon as they receive 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 throughput. Each application process continuously measures the reply latency for operations and adjusts $W$ accordingly. Lastly, we maximize $K$, the degree of asynchrony, between skip list processes which increases the capacity of the pipe, and in effect opens up the throttle all the way.

4.2.1 Consistency Semantics

Figure 4.4 shows the scaling behavior for the three different consistency semantics supported in our skip list implementation: total ordering, sequential consistency, and no consistency. Figure 4.4 gives the results for using up to 200 cores and 20,000 MPI processes across 25 machines. The size of the

---

7On a single machine, because of the low communication costs, throughput on one 8 core machine was equivalent to the throughput across 24 cores of three machines.
4.2. Performance of Distributed Skip List

Figure 4.4: 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 grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$.

list varies from $2^{20}$ to $2^{24}$ by keeping a fixed granularity inside each MPI process. Therefore as the number of cores increases, more load is added to the work in the system. As expected, the performance of the no consistency skip list is best, followed by sequential consistency, while the total ordering skip list gives the worst performance.

The flat trend of the total ordering semantics is the result of increased flow pressure to the head. For each added OS process (i.e., core) we also add an application process and we allow 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 to having more than one application process per OS process. As Figure 4.4 shows, once requests can use shortcuts, replies can return faster and there is less contention at the head and as a result throughput dramatically increases. The upward trend for sequential consistency semantics indicates that the benefits now out-weigh the the cost of the added communication, and in the no consistency case we get further benefits of shortcuts. Both the
4.2. Performance of Distributed Skip List

sequential consistency and the no consistency semantics show good scaling behavior with the number of cores within the range of machines used for our experiments.

4.2.2 Effect of Probability During Tower Creation

The skip list service uses a fixed probability to repeatedly toss coins to determine the height of the tower of a new node. The height of the tower is given by the number of consecutive tosses that produce the same result (one side of a coin). This way of tossing is called a Bernoulli process. The chances of producing high towers decrease exponentially.

Using a very high probability results in most of the list processes having similar leveled high towers. The effect is that the service cannot effectively distribute queries. Similarly, a very low probability will result to a structure closer to that of a linked list and again the result is inefficient request distribution. This is illustrated in Figure 4.5. The best choice of the probability should be close to the fifty percent region (in theory) to allow a binary tree like distribution of operations. However, as the figure shows, our service benefits more around a probability of eighty percent. This may be because at around this value, there can be many high towers and there can also be a good mix of different tower heights. This allows requests to be distributed to many successor processes quickly and hence have a number of search paths for queries.

4.2.3 Range Queries

Figure 4.6 shows the scaling behavior 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 to having more and smaller replies. Figure 4.6 reports throughput with respect to (a) the number of operations, (b) the number of messages, and
4.2. Performance of Distributed Skip List

Figure 4.5: Effect of probability during tower height selection on the skip list service. Probability = 0% (not shown) and 100% both result in performance like that of a linked list.

normalizes the performance across query types by giving (c) the number of results 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 they can send a new range query. However, as we can see in Figure 4.6(c), for larger queries we can obtain more results from the service, which is because a single range query operation sweeps through the service generating
4.2. Performance of Distributed Skip List

result replies as opposed to having multiple separate FIND queries for each result. Figure 4.6(b) shows the number of reply messages all the application processes receive per second. All three range query sizes do scale with respect to the number of reply messages they can receive.

Both Figure 4.7 and Figure 4.8 compare using a RANGE-QUERY versus a corresponding set of FIND operations for medium and small sized range queries respectively. For small range queries, in terms of operations per second, the throughputs are similar. However, we obtain many more results per second with the range queries. We obtain a similar result for large range queries but with a reduced throughput but even more results 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.

4.2.4 Effect of Dimension on Range Query

Figure 4.9 shows performance of our skip list for multidimensional 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 results that are likely to be returned by one 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 results as the number of cores increases. This is clearly evident in Figure 4.9 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 [2] that provide an interesting extension.
4.2. Performance of Distributed Skip List

Figure 4.6: Scaling behavior of RANGE-QUERY with three fixed size range queries: Small = 100, Medium = 1000 and Large = 10000 results. Workload uses 100% RANGE-QUERY operations. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. 

(a) Operations per second

(b) Reply messages per second

(c) Results per second
4.2. Performance of Distributed Skip List

![Graph](image)

(a) Operations per second

![Graph](image)

(b) Results per second

Figure 4.7: Performance comparison between FIND and RANGE-QUERY with a medium sized range query returning 1000 results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. 

70
4.2. Performance of Distributed Skip List

Figure 4.8: Performance comparison between FIND and RANGE-QUERY with a small sized range query of 100 results. Workload uses 20% INSERT, 20% DELETE and 60% of either only FIND or only RANGE-QUERY. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. 
4.3 Summary

In summary, we see that the different consistency semantics produce different scaling pattern in the performance of our list services. The linked list service performs best with the no consistency semantics while the skip list service scales with the sequential consistency and the no consistency semantics. The skip list service provides a scalable mechanism to perform range queries that is suited for ranges with small dimensions. A range queries in our service outperform a corresponding set of individual search operations, which may be useful in real world applications. The performance also show a predictable behavior of the service that can be controlled by a set of parameters with regards to application needs and different resource constraints. This research exposes a good potential for scalability when designing distributed data structures with message passing interface.

Figure 4.9: Throughput versus dimension for range queries. (Separate y-axes for results per second and reply messages per second). Workload uses 100% RANGE-QUERY operations with range query size of 4096. Semantics used is no consistency. Configuration: List size grows from $2^{20}$ to $2^{24}$, MPI process per OS process, $P = 100$. 
Chapter 5

Conclusion

In this thesis we presented and evaluated the design of a novel distributed message-passing implementation of a skip list to support range queries. We also show a service-oriented approach to the design of a distributed data structures. The implementation of a distributed data structure as a service uses its own collection of processes which makes it possible to separate the data structure code from the application code resulting in reduced coupling and increased cohesion in the components of the system. 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 and sequential consistency semantics.

Scalability across machines and communication fabrics requires the addition of tunable parameters to control resources based on the characteristics of the machine and networks. In our design we introduced a variety of these parameters to control computation/communication overlap ($K$), interleaving versus parallel ($P$), load on the data structure ($W$) and process granularity ($G$). There is the flexibility to adjust the granularity, the concurrency and the amount of parallelism. These parameters help adjust the amount of asynchrony to increase the message capacity in the system and to provide more slackness.

Our implementation borrows ideas from work on concurrent list structures but is based on pure message-passing and does not require support for shared memory. There is some commonality with distributed systems in cloud environments but with a focus on scalability and latency, and not availability and reliability. Our ordered list implementations can execute on
a single multicore but more importantly they can also potentially scale to machines with thousands of cores to support sorting and querying for large datasets that can take advantage of the large amounts of available memory.

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. In the past with FG-MPI we have executed programs with over 100 million MPI processes [20].

The experiments demonstrated the ability to scale to an ordered skip list with over a million data items with up to 20,000 MPI processes executing on 200 cores. We showed small range-queries performed very well and that range queries out-performed simple finds. Our ordered list performance for range queries decreases as the number of dimensions increased.

In conclusion, by using a service-oriented approach we are able to create a list data structure that can be used within MPI programs. The skip list is a relatively complex data structure and our design decouples the code needed to implement the list from the application and encapsulated the code related to performance inside the library while exposing a simple set of parameters for the application programmer or user to flexibly tune the performance of the system. This supports our view that a service-oriented approach to distributed data structures provides an interesting design space for large scale, high performance, distributed systems. It lies at the intersection of parallel computation and distributed system design, and well-suited for the types of cluster of multicore machines used in high performance and cloud computing environments.

5.1 Future Work and Extension

There are a number of possible extensions to the ordered list service implemented, which can help optimize message communication and improve overall throughput and load balancing. In the current implementation, no attempt was made to compress extra attachment information and flags in
5.1. Future Work and Extension

process messages. The functionality of the service was the first priority of the research and not the optimization of code. Messages can be packed efficiently to speed up communication between processes. Similarly, it is possible to perform request buffering in code to add slackness to the system. A process can hold off messages that it needs to pass to its next neighbor until enough messages arrive and are ready to be transferred together to the next process, or a current operation requires communication to the successor process. This will improve the communication and computation overlap at runtime, and we can benefit from this option even if the network or the MPI environment do not offer network buffering. The overall number of communication messages will also be reduced, which should remove a good amount of overhead for initiating message handshakes between processes. This would also mean that buffering will be aware of the different types of messaging used and the message buffering process is not done blindly.

Another similar extension is in batching requests. A batch of successive operations may be joined together and be sent off from an application process. This batch may be split like that of the current range-query and multiple replies from different processes should be expected. This again will reduce the communication and produce a better overlap for performance improvements.

The delete operation can be made to convert processes to zombie processes instead of doing the unlink process right away. This kind of lazy-delete strategy can help when a later request can sweep through zombie processes and delete them with less overhead than deleting each node one-by-one. This will present another probabilistic parameter to the system, which decides when is the right time to perform the a sweep through the list to harvest the zombie processes.

The free process sub-service is currently configured to use a round robin scheme. A possible extension to this is to advertise load to other managers at regular intervals and use the current state knowledge to route free node requests so as to speed up the search and perform a better load balancing strategy. It may benefit to allow more than one manager in an OS process to again distribute load among manager processes.
5.1. Future Work and Extension

In our skip list implementation with high granularity, we can consider making a COMBINE operation that is the inverse of the SPLIT operation. At the same time, we can program flow of data elements between neighboring processes. These extensions will allow for better distribution of values among processes even when an adversarial approach to inserting values is employed. However, this will require us at times to initiate communication in the backward direction. This may not be straightforward, but removing the forward-only constraint may create room for other interesting operations in our service.

We should also look to use our skip list service with some real external applications. The Yahoo Cloud Serving Benchmark (YCSB) [9] can be one external simulation application that can measure performance of our service relative to other existing key-value store implementations. However, YCSB is targeted more towards services that are closer to databases where the key-value store does not fit the memory and requires storage into disks. We have not considered storage into disks in the current scope of our research, but this is certainly a future research direction.

Finally, we can use our current approach to the implementation of distributed data structures to create other potentially distributable data structures. An example structure can be the cover tree data structure or structures used for nearest neighbor algorithms. These implementations can benefit by having active data elements that together form a coordination structure.
Bibliography


Appendix A

Pseudocode

The collection of pseudocodes for all the skip list functions are given below.

**LEGEND:**

- R is Request_Type
- K is Key
- V is Value
- S is Source_Address
- O is Other_Information

**SKIPLIST NODE:**

**MAIN FUNCTION:**

```
1 LOOP
2     Recv (R,K,V,S,O) from ANY
3     Switch (R)
4         FIND : call_find()
5         INSERT : call_insert()
6         DELETE : call_delete()
7         OTHER : call_other()
8     EXIT : terminate()
```

**FIND:**

```
1     If K == nodeKey Then
2         Send nodeValue to S [Report Success]
3     Else
4         currentLevel = NODE_LEVEL_TOP
5         Decrement currentLevel up to (nextKey <= K)
6     If currentLevel >= NODE_LEVEL_BOTTOM Then
7         Send (R,K,V,S,O) to ‘Next’ at currentLevel
8     Else
9         Send NULL to S [Report Absence]
```
## Appendix A. Pseudocode

### INSERT:

1. Get newTowerHeight from ‘O’
2. currentLevel = NODE_LEVEL_TOP
3. Decrement currentLevel up to (nextKey < K) or (level <= newTowerHeight)
4. If nextKey < K Then
   5. Send (R,K,V,S,O) to ‘Next’ at currentLevel
5. Else
   6. Request_for_a_new_free_node()
   7. Find range of levels where K is between current and next
   8. For each distinct successors:
      9. Update successor info in ‘O’
      10. Send (R,K,V,S,O) to newNode with R=BIND
   11. set currentLevel after the range
6. If currentLevel >= NODE_LEVEL_BOTTOM Then
   7. Send (R,K,V,S,O) to ‘Next’ at currentLevel

### BIND:

1. Update successor info for the given range of levels
2. LOOP until tower completely built
   3.Recv (BIND,K,V,S,O) from ANY
   4. Update successor info for the given range of levels
6. Send nodeRank to S [Report Success]

### DELETE:

1. If nodeKey == K Then
   2. Get range of levels for DELETE from ‘O’
3. For each distinct successors:
   4. Update successor info in ‘O’
   5. Send (R,K,V,S,O) to messageSender
Appendix A. Pseudocode

Decrement nodeTowerHeight

If nodeTowerHeight == 0
   Append self to FreeList
   Send nodeRank to S [Report Success]

Else
   currentLevel = NODE_LEVEL_TOP
   Decrement currentLevel up to (nextKey <= K)

   If currentLevel < NODE_LEVEL_BOTTOM Then
      Send NULL to S [Report Absence]
   Else
      If nextKey < K Then
         Send (R,K,V,S,O) to ‘Next’ at currentLevel
      Else
         Find range of levels where K == nextKey

         Update range info in 'O'
         Send (R,K,V,S,O) to ‘Next’ at currentLevel

   LOOP until end of range
  Recv (DELETE-REPLY,K,V,S,O) from next
Appendix B

Parameters available in experiments

The parameters available when running experiments are given below with brief description and default values.

Parameters Available:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>10</td>
<td>Number of OS-processes</td>
</tr>
<tr>
<td>nfg</td>
<td>10</td>
<td>Number of processes inside an OS-process</td>
</tr>
<tr>
<td>b</td>
<td>100</td>
<td>Granularity threshold</td>
</tr>
<tr>
<td>w</td>
<td>10</td>
<td>Window size for applications to make outstanding requests</td>
</tr>
<tr>
<td>k</td>
<td>20</td>
<td>Number of requests sent eagerly by list nodes to their neighbors</td>
</tr>
<tr>
<td>IP</td>
<td>0</td>
<td>INSERT probability</td>
</tr>
<tr>
<td>DP</td>
<td>0</td>
<td>DELETE probability</td>
</tr>
<tr>
<td>RFP</td>
<td>0</td>
<td>RANGE-QUERY probability (Note: that the rest of the probability is for FINDS)</td>
</tr>
<tr>
<td>AS</td>
<td>1</td>
<td>Should we use APP shortcuts</td>
</tr>
<tr>
<td>LS</td>
<td>0</td>
<td>Should we use LIST shortcuts</td>
</tr>
<tr>
<td>MD</td>
<td>1</td>
<td>Dimension for Range Query</td>
</tr>
<tr>
<td>ST</td>
<td>1</td>
<td>Special Test ON or OFF. Special test fills the whole list with INSERT before changing to workload specified by [IP, DP and RFP]</td>
</tr>
<tr>
<td>SP</td>
<td>100</td>
<td>Expansion for the Range Query</td>
</tr>
<tr>
<td>TP</td>
<td>50</td>
<td>Probability used to toss coins when generating tower height via Bernoulli process</td>
</tr>
</tbody>
</table>
Appendix B. Parameters available in experiments

LT    -->  [default: 0] Latency Threshold to increase or decrease Window Limit, W, in app processes.

ITER -->  [default: 1000] Number of iterations before an MPI_Wtime() is called.

To compile project use:
-----------------------
make all

To run project use:
-------------------
make run      (for linked list)
make runb     (for linked list with granularity)
make srun      (for skip list)
make srunb     (for skip list with granularity)