*LH*Algorithm:*
Scalable Distributed Data Structure (SDDS) and its implementation on Switched Multicomputers

Written by: Salman Zubair Toor
E-Mail: salman.toor@it.uu.se
Teacher: Tore Risch
Term paper for the course:
Principles of Modern Database Systems

Introduction

In this term paper we are going to give a review of the LH* algorithm, one of the most popular Scalable Distributed Data Structures (SDDS). To make this review easy to understand we divide the whole discussion into three main parts: first, a brief overview of how the basic LH* algorithm works and its implementation using insertion, deletion and parallel querying from paper ref-[1]. Second part is about the discussion of a variant of LH* algorithm i-e LH^*_{LH} on switched multicomputers using paper ref-[2] and then in the last we will discuss some of the interesting results. The purpose is to develop good understanding of SDDS and learn how " LH* files can be efficiently scalable to files that are orders of magnitude larger in size than single site files " according to paper ref-[1].

In the last couple of decades computer hardware has become very advanced in terms of processing power and memory requirements. To maximize the utilization of speed and memory there is always a need for efficient algorithms and data structures. Recent architectures have attempted to get benefits from distributed or parallel processing. The reason behind using such architectures is no matter how powerful one processor is we can achieve better performance with two or more than we can with one, but only by using special algorithms and data structures. One big issue in using many processors may decrease the performance. This is because the appropriate number of processors for a particular algorithm is unknown or can be found by experiments. According to the reference papers, one can build an efficient algorithm for distributed or parallel computing environment by keeping the following points in mind.

1. File expands to new servers gracefully.
2. There is no master site that object address computations must go through, e.g. to access a centralized directory.
3. The file access and maintenance primitives, e.g. search, insert and split never require atomic updates to multiple clients.

A structure that can meet these constraints is called a Scalable Distributed Data Structure (SDDS). In the first section we are going to discuss one of the most popular SDDS called LH*. LH* is the generalization of linear hashing (LH) first introduced by Litwin in 1980.

According to the paper ref-[1] "LH* can accommodate any number of client and servers and allow the file to extend to any number of sites with following properties:"

- File can grow to practically any size with the load factor about constant.
- An insertion usually requires only one message, three in worst case.
- Retrieval usually required two messages, four in worst case.
- A parallel operation on a file of M buckets costs at most 2M+1 messages, and between 1 and $O(\log_2 M)$ rounds of messages.

Such features cannot be achieved by a distributed data structure using centralized directories. LH* is especially very useful for very large files. With the help of paper ref-[2] we will discuss one of the variants of the LH* algorithm called LH^*_{LH} on switched multicomputers and try to understand the performance of this SDDS in a parallel computing environment.

Implementation

In order to discuss LH*, it is important to first understand the basic idea of how linear hashing works.

Linear Hashing

Linear hashing is a method for the disk and RAM files that grow or shrink dynamically. In this method files are organized in buckets and these buckets are addressable through a hash function. One example of the hash function:

$$h_i (C) \longrightarrow C \bmod N * 2^i$$

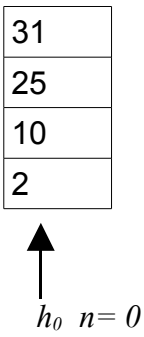
where h_i is the hash function
i = 0,1,2, ...
N = initial number of buckets

When a bucket is full, a split operation uses this hash function and divides the elements of the bucket into two buckets. The split is of two types: controlled and uncontrolled. Whenever a collision occurs in the bucket the split operation called, this type of split is known as an uncontrolled split, according to paper ref-[1]. The load factor for this type of split is 65% to 69% depending on the size of the bucket. There is another method to decrease the load when we have large buckets. Monitor the load factor and if the load is higher than the threshold, the split operation is explicitly called. This type of split is known as a controlled split.

Insertion of items in the bucket

The process of inserting items in the bucket can be explained with the following example.

Example



n = 0 and i = 0

To insert the item in the bucket we need to handle two pointers:

i = Level of buckets

n = special pointer, use to find which function should applied to the key

Consider one bucket having the capacity of four items. All four spaces are filled using the algorithm.

a $\longleftarrow$ $h_i (C)$ *where $h_i (C) = C \bmod N * 2^i$*
If a < n then *N = 1 (initial number of buckets)*
a $\longleftarrow$ $h_{i+1} (C)$ *in our case i=0* *$h_0 (C) \bmod 1$*

A new item (35) appears, causing an uncontrolled split to occur.

	35
10	31
2	25

h_1 $n = 0$ h_1

Now i = 1 n = 0

$h_1 (C) = C \bmod 2$

h_1 shows the hash function used on bucket. *n = 0,* shows the left most bucket having least value of hash function. *i = 1,* shows the lowest hash function value in the buckets.

Now three more items 27, 8 and 6 are next to be entered in the buckets.

8	27
6	35
10	31
2	25

h_1 $n = 0$ h_1

$h_1 (C) = C \bmod 2$

where N = 1

Now the new item 66 is the next to enter in the bucket. Item 66 will go to the first bucket, according to the hash function. The bucket is full, so an uncontrolled split operation will occur.

	27	66
	35	6
	31	10
8	25	2

h_2 h_1 $n = 1$ h_2

i = 1 n = 1 and the algorithm is

$a = h_i (C)$

where $h_i (C) = C \bmod N * 2^i$

If a < n then

$a = h_{i+1} (C)$

N = 1 (initial number of buckets)

This time items 2, 6 and 10 go to bucket 2 because of the condition. Now again *i = 1* because of the lowest value of the hash function. *n = 0*, points towards the bucket having lowest hash function value.

As we see in the insert function, when a new item comes and the bucket overflows, a split operation occurs and places the new item by introducing a new bucket. Similarly when an object is deleted, and a bucket becomes empty, the merge operation will be triggered. It will then adjust the number of buckets from n to *n -1,* and space will be freed. Similar to the controlled split, merge can also be handled by monitoring load factor and calling the merge function explicitly.

LH* Scheme

LH* is one variant of linear hashing. The file expansion works in the same way as it works in LH algorithm. A client can insert, delete and search for the item by the reference of the key. Each client has an image of *i* and *n* that is used to find the bucket to insert or delete the items. If there is a number of clients than it is not possible to update each of the clients after the insertion or deletion operation (this decreases performance and is against the basic requirements of the SDDS). Consider the following scenario: at site level we have *n* and *i* containing some value. Clients have *ii* and *nn*, copies of the original original *n* and *i*. Each time when a split occurs, a new bucket will be created on the new site. Let us consider *i=1* and *n=1* at site level and at client-1 *ii =1* and *nn =1*. Client-2 wants to enter an item in the bucket that triggers the split operation. The values of *n* and *i* are changed and all the other clients have the old images of *i* and *n*. This will cause an address error for the client. To update the status at client level after each insertion and deletion causes extra overhead, which gets worse as the number of clients increases. The solution to this problem is provided in the LH* scheme. There are three steps for solving this issue, according to the paper ref-[1]

1. The first step is that every address calculation starts with client address calculation.
2. When a server receives the request, it will recalculate the address because it might be possible that the client has old images of i and n. The recalculation confirms whether the request came to the right server or not. If not it will be forwarded to the server that can fulfill the request.
3. The last step is that the client that did the address error will get the Image Adjustment Message (IAM).

By getting this IAM, the client gets very few address errors and can get the required item with a maximum of three forwardings. The best case required one message and the worst case required three to get the element back at client site.

Bucket splitting in LH*

As discussed earlier in linear hashing, splitting works in the same way in LH*. In LH* both controlled and uncontrolled strategies are available. Whenever a split occurs, the values of *i* and *n* will be changed. There is a site that maintains the global values of *i* and *n* called Split Coordinator (SC). In uncontrolled splitting when a bucket is full the site sends a message to the SC. SC responds with a message ' you Split ' and the calculation of new values of *i* and *n* will be started on SC independently. The whole process follows these four steps.

1. Create a bucket $n+2^i$ with level $j+1$
2. Split bucket *n* applying h_{j+1}
3. Update $j \longleftarrow j+1$
4. Commit the split to the coordinator

In each insertion and deletion, it is necessary to send the update image of *i* and *n* to the coordinator. this will make the SC hot spot. This hot spot decreases the performance of the scheme especially when there are many sites and clients. To handle this situation controlled splits are available. There are different strategies are available in controlled splitting to reduce the overall message loads by calculating the threshold for the sites.

File Contraction

File contraction or deletion of items works similar to the split operation. In deletion we also have two cases.

1. When a bucket is empty the SC will send a message to the second-to-last bucket to merge with the last bucket and recalculate the values of *i* and *n*.
2. Some threshold level can be set for the buckets. If the load is lower than the threshold level then the merge operation is called.

Due to the merge operation it might be possible that the client will request to a site that no longer exists, because of the difference of *i* and *n* images at client level. Again there are different possible ways available, two of them are:

1. In this scenario SC will sent the IAM to the client.
2. The values of *i* and *n* at client level will be set to zero, since bucket zero will be the last bucket in the file to be deleted.

Allocation of sites

Address of a bucket is a logical address that is translated into the actual address *s* of a server or site onto which the file can expand. The translation should be performed similarly for both clients and servers. According to paper ref-[1] the expansion of the file on different sites can be handled in the following two ways.

1. The set of sites on which a file can expand is known to the server in advance using static tables. This approach can be used in a local net of an organization or on all the processors of a multiprocessor machine.
2. The set of sites on which the file can shrink or grow dynamically according to the size and the sites. This is more suitable for the wide area networks.

For the first option, a table can be maintained on the server in which addresses of all the available sites (servers) are written. A few kilobits of table can handle all the servers of an organization or all the nodes in a large cluster. Using this approach one can expand gigabytes of size in RAM or terabytes in hard disk.

In Case 2, the dynamic management of site address provides better memory management on the servers and also allows large range on the servers to be used for file expansion. A copy of this dynamic table is also available on the split coordinator. Whenever a split or a merger occurs, SC will be updated. Still, all these operations and queries do not make SC a hot spot because the client gets far fewer addressing errors.

Parallel Queries

Using the LH* algorithm a file can expand distributively on the server, and the overall structure of the file allows the user to run the parallel queries on it. For parallel querying the following two options are available.

1. Queries can only get a response from the server that has the requested results. This approach reduces the load and gives results in less time.
2. The completion of the query requires response from all the sites on which the file has buckets. This approach is comparatively more time-consuming but guarantees that all the sites process the request without any error.

One of the main features of the LH* scheme is that the client does not know the exact extent of the file. In parallel query processing, when one site receives a query, it forwards that message to its offspring that have buckets related to that file. This process will continue until all the buckets that have the file will receive this query request. For the message forwarding there are two types of options available,

1. Broadcast/Multicast messaging scheme.
2. Point-to-point messaging scheme.

Each of these approaches has its own benefits and drawbacks, see paper ref-[1] for complete details.

LH* on switched multicomputers

Reduced communication among different nodes makes the LH* scheme very suitable for multiprocessor machines. In paper ref-[2] the performance of LH* was analyzed on switched multicomputers. The LH* scheme presented in the paper ref-[2] is slightly different from the basic LH* called LH^*_{LH} .

The multicomputer machine used for this analysis was Parsytec GC/Power architecture with distributed memory. The machine has 128 POWER-PC-602 RISC processors. Each node has two processors and 32MB shared memory and four T805 Transputers (a family of micro processors designed for parallel processing). Each Transputer has four bidirectional communication links. The nodes are connected through the big fat cable.

The communication between client and server is safe (due to the architecture of the multicomputer machine, the user does not need to wait for the acknowledgment). Client can start more than one process to insert, delete or search the elements in the site. The multiple channels increase the throughput. In the architecture discussed in paper ref-[2], the clients started from higher nodes and the server started from lower nodes.

The functionality of LH* was already discussed in previous sections. Now we will discuss the server-side implementation, dynamic partitioning strategy and the concurrent request processing on switched computers used in paper ref- [2].

Server Architecture

According to the paper, the server consists of two layers: LH* manager and LH manager. LH* manager handles communication and concurrent splits, whereas LH manager manages objects in the buckets. For LH*, the buckets in the paper ref-[2] used some extra parameters that help make the scheme more robust. In the implementation of LH^*_{LH}, the LH* bucket contains

1. LH Level *i*, splitting pointer *n* (same as we discussed before) and *x*, number of stored object.
2. A dynamic array of pointers (LH* bucket is implemented using a linked list).
3. LH buckets with records.

Each recode contains the calculated hash value called “pseudo key”. The bucket splits when *L = 1*

Where $L = x / (b_l * m)$

and *b = Number of buckets in LH file*
m = file parameter, mean number of objects in buckets

Partitioning of LH files

According to the technique that is followed in the paper ref-[2], both of the layers share the pseudo key *J*. The key size is 32 bits. LH* manager uses the lower bits called *l* of *J* and LH manager uses higher bits called *j*, where *j*+ *l* <= *J*. The value of *l* determines whether the split will occur or not. On each split the value of *l* is increased by one and the value of *j* is decreased by one.

Concurrent Request Processing

The concurrent operation of handling client requests while the server is doing a split operation is done by the splitter. If the request is about the element that is shipped, then the splitter forwards the request to the new site. Otherwise, the request is processed within local LH-table. A request that concerns the current LH bucket being shipped is first searched among the remaining objects in that LH-bucket. If not found, it is forwarded by the splitter. All forwards are serialized with the splitter task.

Shipping

In LH^*_{LH} the transfer of the objects from one bucket to the next can be handled in two ways.

1. bulk shipping (the objects are transferred in bulk)
2. individual shipping

The bulk splitting only needs to send one message because the environment that is used in communication is safe. The individual sending of objects creates very high communication overhead. In bulk shipping, it is also important to note how much overhead the packing and unpacking of the bulk creates, since the objects are not contiguous in memory. The best possible way that is used in this paper is to create limited sized bulk.

Results of LH* and its implementation on switched multicomputers

In this section, we will briefly discuss the results presented in both of the papers ref-[1] ref-[2] under discussion. The basic measure of performance of LH* is the number of messages required to complete an operation. A random search for a value C needs two messages on average, and four messages in the worst case. In random insertion without address error, only one message is needed. This performance can never be achieved in the centralized directory structure. In worst case, i.e. when an address error occurs, three messages are needed. The insertion can also initiate the message for the split coordinator when the bucket overflows, but that message does not cost much overhead because this operation is asynchronous. In paper ref-[1] a simulation model is used to verify the performance of the LH* scheme. For details of how the simulation model was designed see Result section of paper ref-[1]. Using the model, a number of experiments was done in which clients insert 10,000 to 1,000,000 random objects when building a file, having bucket range 20 to 20,000. Now we are going to discuss some of the interesting results.

1. The results showed in the paper ref-[1] verified the performance predicted in the theory of LH* scheme. The performance of the LH* file is better than that of an ordinary file, especially for large files with bucket capacity $b > 200$. The results also show that the IAM creates very little overhead for the large bucket size. The estimated overhead for the large buckets is 0.2%.
2. The load factor is estimated with and without load control for various bucket capacity *b* and thresholds *t*. All the results presented in the paper ref-[1] show that the load factor without load control is efficient, and even gets better results with the load controlled strategy. In the paper ref-[1], five different strategies are analyzed for the load factor.
3. Another very interesting case has been discussed in the result section of paper ref-[1], i.e. the performance of a client when one is less active than the others. The experiment uses two clients with an insertion ratio 1:1000 with bucket size 50. This test is quite important, since the less active client will definitely get more address errors than a fast client. The results of this experiment show that the performance of the less active client only degraded less the 10% as compare to fast client. The table in paper ref-[1] section 4.3.4 shows the performance of less active clients compare to active clients at different insertion ratio.

Now we are going to briefly discuss the results of paper ref-[2].

1. The results show the scalability of LH^*_{LH} scheme on switch multicomputers. In experiments, insertion of elements n = 1,2,3 ..... 235,000 was performed simultaneously by k clients. The results showed that the time taken by 1 client doing 235,000 insertion is 330 seconds where as 2 clients took 170 seconds. The time was reduced to 40 seconds with 8 clients.
2. The results also show the comparison of static and dynamic splitting strategies. The static splitting uniformly leads to longer time needed to build the file whereas the dynamic splitting gives better timings for building the file and even gives better performance when the number of clients is higher.
3. Section 5.2 shows some very interesting statistics regarding bulk and individual splitting. By experiment, it is proved that the bulk splitting is much faster than the individual splitting. Still the right size of the bulk can only be determined with experiments because it depends on the load factor created by packing, unpacking and the overhead on the communication medium.

References

1 – W. Litwin, M-A. Neimat, and D. Schneider. LH*: A scalable Distributed Data Structure (SDDS)

2 – Jonas S Karlsson, Witold Litwin, and Tore Risch. LH*$_{LH}$: A Scalable High Performance Data Structure for switched Multicomputers.