A Scalable Data Structure for
A Parallel Data Server

Jonas S Karlsson

February 11, 1997

Contents

List of Figures

Preface

Contents

In this thesis we identify the importance of appropriate data structures for parallel data servers. We focus on Scalable Distributed Data Structures for this purpose. In particular LH* [LNS93], and the new data structure LH*LH [KLR96]. An overview is given of related work and systems that have traditionally implicated the need for such data structures. We begin by discussing high-performance databases, and this leads us to database machines and parallel data servers. We sketch an architecture for an LH*LH-based file storage that we plan to use for a parallell data server. We also show performance measures for the LH*LH and present its algorithm in detail. The testbed, the Parsytec switched multicomputer, is described along with experience acquired during the implementation process.

Parts of the thesis are based on the article on LH*LH [KLR96] published in the lecture notes from the 5th International Conference on Extending Database Technology, in Avignon, France 1996.

Intention

The intention of this thesis is to demonstrate a number of design decisions and also to explain some problems experienced during the development of the LH*LH implementation. This will, we hope, provide a better understanding of LH*LH and experience in implementations of SDDSs as well as some insight in SDDSs themselves.

Thesis Overview

Chapter 1 introduces database systems and parallel data servers as means of achieving high performance.

In Chapter 2 the imporance of data structures is identified. Both the need of scalability and distribution are discussed.

Chapter 3 then introduces the concept of Scalable Distributed Data Structures (SDDSs). The LH* algorithm is explained in detail.

In Chapter 5 a framework for using an SDDS (LH*) is then sketched in the context of the AMOS database system. We then describe the LH*LH algorithm in Chapter 6.

The Parsytec hardware architecture is presented in Chapter 7. Then, in Chapter 8 we provide real measurements for our implementation of LH*LH.

Chapter 9 goes into details of the implementation of LH*LH. Possible improvements are discussed.

Finally, a discussion and proposals for future work are presented in Chapter 10.

Financial Suport

This project was supported by NUTEK (The Swedish National Board for Industrial and Technical Development), and CENIIT (The Center for Industrial Information Technology).

Acknowledgment

First of all, I would like to thank my advisor, Professor Tore Risch for introducing me to databases in detail, especially about implementations. Tore has given much support during my writing of this licentiate thesis. For that I am grateful. Professor Witold Litwin, in Paris, introduced me at an early stage to SDDSs on which I also base my work. I thank him for this generous support during my work on LH*LH.

Finally, I would like to thank my former and current colleagues at EDSLAB for the working environment, especially to Padrone who writes my name correctly. I am also grateful for the help provided by Niclas Anderson and Lars Viklund at PELAB on the Parsytec machine. I am also indebthed to Henrik Nilsson for many interesting discussion hours over tea, and lively email-discussions on topics in computer science. Special thanks goes to our secretary Anne Eskilsson, Ivan Rankin and Billys^{TM}[1]. The last one I hope to be able to avoid in the future...

Thanks,

Jonas S Karlsson
Linköping, January 1997

[1] Billys micro-wave Pan Pizza.

Chapter 1

Introduction

In this chapter we introduce the reader to various types of database systems (DBMSs); we then provide examples of the need of high-performance databases. The main architecture types of databases are explained. The important issue of *scalability* is then identified. We discuss the implicated need for *scalable data structures*, scalability both from an accessing and a processing point of view as well as for updates. In most practical cases, as will be seen, parallellity or distribution is mostly used as a means for implementing high-performance. The problem still remaining is that of scalability, i.e., the ability to grow/resize the application and the database to any unforseen size.

1.1 The Need for High Performance Databases

Databases, do we need them? There is currently a popular trend that shows that people want to combine and/or access data using databases. Many common user applications, e.g. Microsoft products, now permit databases such as MS SQL to access external data sources, for example email-files, diverse database formats, spread-sheet data, and so on. However, there is another trend, using a different approach, at present mostly in the database research community, which is to specialize the DBMS to a specific application. Examples of this can be found in the area of *Engineering Databases* or *Scientific Databases*[FJP90], where a large amount of data is handled not by the application but a database engine. For efficiency the database should be extensible with operations from the application domain and appropriate new types of indices. Computed Aided Engineering (CAE) systems are applications of interest for merging with databases, since they require advance modeling capabilities as well as advance queries; this is explored in the FEAMOS research prototype [Ors96]. It allows matrices to be used in the query language, and equations can be solved by stating a declarative SQL-like query. Indications show that application programs become more efficient and more flexible, also they are easier to build. A popular way to implement this merge is to embed the DBMS (code) into the application as a library. This then lets the application directly traverse and use the data using a so-called *fast path* interface. The database system can then also be extended to use, index, and query application data. *DataBlades* [SM96] is the Illustra concept of a packaging a collection of data types together with access methods, and related functions and operators into a module. Most other DBMS companies now develop similar concepts, under different names, but the idea is the same, modularily extensible databases systems that can handle new types of data.

The trend, to use database for more technical purpose, also draws interest from the telecommunication industry [Dou90]. Interest in this area, is currently much concerned with high-performance reliable DBMSs [Tor95], with down rates of less than a few minutes a year. There are estimates of the needed rates of insertions, updates and also queries, and the number of such events is in the range of 10 000 per second. This is currently not possible with any of the commercially available database systems. Possible applications are directory management, charging of calls, email-databases, and multi-media repositories. These application databases are potentially huge in comparison with a normal databases.

Current database technology does not support these amounts of data, either with the required high availability or with respect to the scalability of processing. That is, today's (database) systems are static in nature, the ability to give the same per transaction performance when the amount of data doubles and/or the number of transactions doubles is missing. The concept of a system being able too, when given added resources, increase its performance lineary (roughly) is called *scalability*. In short a scalable system should when the resources are doubled double the processing power (performance). Distributed and parallel (database) systems are the natural means to build systems that cope with high storage demands. We will now view the principles of such database systems.

1.2 Conventional Databases

Single-user databases are widely available and can be used on off-the-shelf hardware, such as single workstations or PCs.

Central Databases are then the most common type of databases for multi-user environments. They run on single (mainframe) computers. Banking databases, travel agency booking or corporate billing databases are examples of central databases. Several users can access the database using either the old style terminals or using client/server software. SQL is the most widely used and standardized query language, OQL (Object Query Language) and QBE (Query By Example) are other languages which are used.

1.3 Distributed Databases

A Distributed Database System (DDBS) can be defined as "a collection of multiple, logically interrelated databases distributed over a computer network" [ÖV91]. Further on they also define a Distributed Database Management System (DDBMS) as "the software system that permits the management of the DDBS and makes the distribution transparent to the users." The important terms here are "logically interconnected", "distributed over a computer network" and "transparent". Then they give examples of what is *not* a DDBS:

- a networked node where the whole database resides
- a collection of files
- a multiprocessor system

- a shared-nothing multiprocessor system
- a symmetrical multiprocessor system with identical processors and memory components.
- a system where the OS is shared.

A DDBS is often heterogeneous with respect to hardware and operating systems. The data is physically stored at different sites in component databases, and the DDBMS is then the integration of these data into one virtual "database". However, the same capabilities and software are usually part of the individual component database. The transparency is the most important feature of a DDBMS. The main difference compared to multidatabase systems, which we will discuss in the next section, is that a distributed database distributes the data transparently over a number of nodes where each node uses the same DB software to manage its local data and where the nodes are coordinated through the DDBMS. Queries are can then be executed jointly and coordinated to provide efficient execution.

1.4 Multidatabases

By contrast a *Multidatabase System* (MDBS) is built up from a number of autonomous DBMSs. Most of problems in the area of DDBMSs have their counterparts in multidatabase systems, too. However, the design is bottom-up: individual databases' already exist, and they have to be integrated to form one schema. This involves translations of the different databases capabilities during query processing and data exchange. A multidatabase system has to cope with different variants of query languages, and perform all moves of data itself. It acts as a layer of software in-between the databases and the user, and the databases do not communicate with each other. Since multi-database updates are a problem, they are usually not allowed online. Instead, they are executed locally, or in batch mode.

1.5 Data Servers

Another trend is to use to the availability of powerful workstations and parallel computers for managing internal data in a DBMS. Such a computer dedicated for this purpose is called a *Data Server*. An example of this approach is shown in Figure 1.1, where several *Terminals* are connected to an *Application Server* that handles user input and data display, parses the query and calls upon the *Data Server* to execute it. The database itself is stored on a secondary storage media (disk). Data servers also seem to be becoming popular as storage sites of distributed databases[ÖV91]. By dedicating the computer for a data server, it is then easier to tune the memory management algorithms. Usually the database systems have more knowledge than the operating system as to *how* and *when* it uses *what* data. In the 1970s the idea of dividing the database management system into two parts, a *host computer* part and and a *backend computer*, appeared [CRDHW74]. Nowadays the terms *application server* and *data server* are used respectively. Figure 1.1 shows the main idea.

1.6 Parallel Data Servers

Parallel computers are nowadays becoming more and more widespread. For such hardware DDBS technology is used in implementing *parallel data servers*. A parallel data server is essentially implemented on a parallel computer and makes extensive use of the advantages of the parallelism in data management that then can be gained. Often, support for distributed databases is part of the implementation. The data managed is automatically fragmented or declustered, making the system self-balancing. The work on parallel data servers is related to the work on Database Machines, which will be discussed in the next section. However, since special parallel hardware computers are expensive and current technology is advancing fast, the trend is to use a number of networked mainstream machines for implementing the parallel data servers. For the fast interconnect network in building clusters of machines (network multicomputers) the Scalable Coherent Interface, SCI [IEE92], is becoming more and more popular.

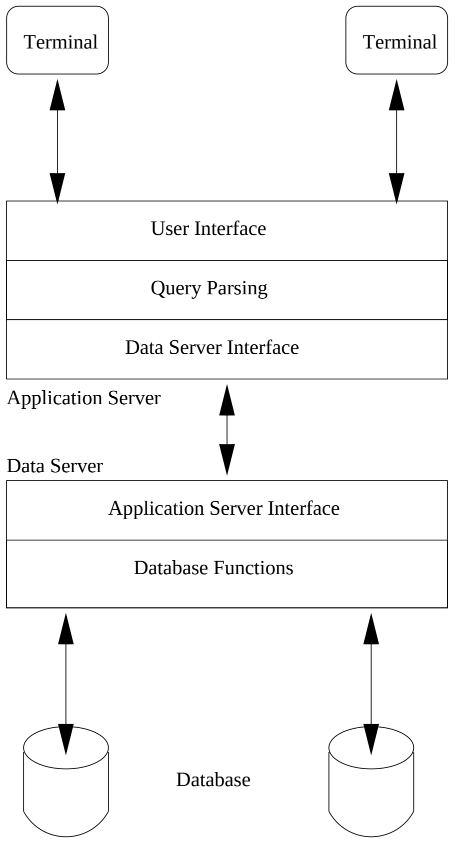


Figure 1.1: Data and application servers.

1.7 Database Machines

Related to the work on parallel data servers is the earlier work done in the framework of *Database Machines*. Below we explain the term and present a short overview of some selected systems. In the next chapter we then go further into details of how large amounts of data are managed in very large systems.

The first mention of a Database Machine was in [CRDHW74]. The term *Database Machine*, or *Database Computer*, or *Data Server* which nowadays is a natural choice in a distributed environment [ÖV91] is often used for a DBMS-dedicated machine. In such a machine there is not an operating system in the ordinary sense. Hence, the DBMS has specially tailored operating system services; in the simplest example this means just device drivers and a monitor. This is in contrast to a more typical DBMS environment on a general-purpose computer with some operating system. The reason for having a dedicated machine with more specialized software and hardware is to overcome the I/O limitations [BD83] of the *von Neumann* computer architecture and other restrictions. Another reason is to be able to use technology that is not yet available off-the-shelf. One way to overcome I/O limitations is to keep the whole database in stable main memory [LR85] or I/O bandwidth can be increased by using parallel I/O [Du84]. Multiprocessor computers have been studied for performance and data availability.

There are mainly two types of parallel computer architectures. The *Shared-Everything* type of computers provide high performance but are not scalable to any larger sizes. All the nodes share memory, disks and all other resources are typically communicated via shared buses. The Sequent Computers and Sun SPARC/Center machine are other examples of shared-everything computers. It is widely known that this architecture limits the size of an efficient system to around 32 processors. However, it is relatively easy to program. The alternative, the *Shared-Nothing* computer type, requires extensive programming to share any information, and to perform any kind of work jointly using the available resources. Often new algorithms have to be engineered, and much research is concerned with finding algorithms to use the power of the shared-nothing computers. The benefits are that, if one succeeds in programming the shared-nothing computer in a scalable way, the application can scale to many more than just 32 processors.

1.8 Overview of Some Data Servers

In *Parallel Database Systems: The Future of High Performance Database Systems* [DG92] there is an overview of state of the art commercial parallel systems. *Teradata* is a shared-nothing parallel SQL system that shows near-linear speed-up and scale-up to a hunred processors. The system acts as a server back-end and the front-end application programs run on conventional computers. The *Tandem NonStop SQL* system uses processor clusters running both server and application software on the same operating system and processors. The *Gamma* system, too, shows near linear speed-up and scale-up for queries; it runs on Intel's iPSC/2 Hypercube with a disk connected to each node. A recent implemenation of *Oracle* runs on a 64-node nCUBE shared nothing, with good price- performance measures: also it was the first to provide 1000 transactions per second.

Examples of shared-nothing databases are Bubba [BAC+90], Teradata DBC/1012 [Cor88], Gamma [DGG+86] and the Tandem Nonstop SQL [Tan87]. Examples of shared-memory database systems are XPRS [SKPO88], and the Sequent machine.

Bubba [BAC+90] started out in 1984. The aim was to design a scalable, high-performance and highly available database system that would cost less per performance unit than the mainframes in the 1990s. At the beginning the Bubba project was mostly concerned with parallelizing the intermediate language, FAD. FAD was used for LDL [CGK+90] compilation. The FAD language has complex objects, OIDs, set- and tuple-oriented data manipulators and control primitives. Both transient and permanent data are manipulated the same way. The FAD program was translated into the Parallel FAD language extension, in combination with the Bubba Operation System (BOS) they built. In the project data placement, process and dataflow control, interconnection topology, schema design, locking, safe RAM and recovery were studied. They later regretted including all these features and functions that limited the complete study of the complex systems. In their first prototype they learned quite a few "lessons", as they say. Parallelity, for example, gives rise to extra costs in terms of processes, messages and delays. Dataflow control was another important issue. Also redesign of the language and implementation were carried out. Another problem was their usage of three different storage formats for objects (disk, memory and message). They used the C++ environment that did not make things easier. So in their second system they used the C language. The second, and perhaps more realistic prototype, was rewritten from scratch. There were several reasons for this, including C++, new programmers, and serious robustness problems. Since this was not to be a commercial system, not all important features of the Bubba system were implemented. In the new system only one type of object representation was used, and it was the same for disk, memory and messages. For the Bubba Operating System, the AT&T UNIX was used, with some extensions. Their conclusions at the end of their final Bubba prototype are that "Shared nothing is a good idea (but has limitations)"; "dataflow seems better than remote procedure calls (RPCs) for a shared-nothing architecture"; "More compilations and less run-time interpretation", "Uniform object management", and that for fault tolerance it is better to replace a failing node than trying to make the nodes fault tolerant. Apart from this, they mention that there was some trouble finding a commercally-available hardware platform for their work. Even though the hardware and software were bought, there were both software (operating system) and hardware bugs, but eventually the system functioned properly.

Another system was the *PRISMA/DB* system [AvdBF+92], which was a parallel, memory relational DBMS. It was built from scratch using easily available hardware at the time. It was built on the POOMA shared-nothing machine. Each of the 100-nodes (68020) had 16 Mbytes of memory and they were interconnected in a configurable way. The Parallel Object Oriented Language (POOL-X) was developed, that featured processes, dynamic objects, and synchronous as well as asynchronous communication. Some of its specialities were that it could create tuple types on the fly and conditions on them could be compiled into routines. This helped to speed up scanning, selections and joins. The project was not as successful as one might expect from a main-memory system. It was not really a magnitude better than disk-based DBMSs. The problems included the facts that the hardware did not run in full speed, that the hardware was outdated when the project was evaluated, and that the compiler of the experimental programming language was not fully optimized. But among the positive results they found that by using their language they managed to build a fully functional DBMS, and the project could then be finished on time.

The *XPRS* (eXtended Postgress on Raid and Sprite) DBMS [SKPO88] was aimed at high availability and high performance for complex ad-hoc queries in applications with large objects. It was optimized for either a single CPU system or a shared-memory multiprocessor system. The aim was to show that a general purpose operating system can also provide high transaction rates and that custom low-level operating systems are not a necessessity. They were much concerned with removing *hot spots* in data accesses. This was done by reducing the time the locks are being held using a new locking schema, and by running DBMS commands in a transaction in parallel. A *fast path* schema was proposed to achieve high performance, as opposed to the common method of stripping out high-level functionality (such as query optimizations and views) from the DBMS. For better performance when I/O-ing large objects they built a two-dimensional file system. This achieved reduction of the mean time to failure (MTTF) then cured by using RAID [PGK88] or striping techniques that provides fault tolerance. These techniques keep a bit parity block for N disk blocks (on different disks). This block can be used to reconstruct any of the $N+1$ blocks from the N other blocks. Thereby the overhead is reduced to $1/N$.

DBS3 [BCV91] takes a newer approach, using the assumption that the success of RISCs lies in simplicity and high performance compilers, that large main memories will be available, and that one should rely on advanced OSs. The first implies that a good optimizer and simple basic DB units are more important than having a complex design and a complex language to program it. Furthermore the whole database can be entirely stored (cached) in main memory. This simplifies optimization and cache management. Lastly, portability is now important and most newer OS include better means for memory management (cache tuning, virtual memory, mapped I/O) and transactions (threads). Permanent data is stored apart from temporary data. This two-level storage divides data so that permanent data is stored on disk and temporary data is managed in (virtual) main memory. It can make use of fragmented relations, both temporal and permanent. Zero or more indices can be used for each fragment. The transaction processing is aimed at online transactions and decision-support queries. Using their parallel execution model, they finally achieved good intra-query parallelism, using pipelining and declustering.

As part of the *Sequoia 2000* Project at the University of California, the *Mariposa* project [SAP+96] proposes a micro-economic model for *Wide Area Distributed Database Systems*. Their system uses terms from the market economy: sellers, bidders, brokers, budget, purchase, advertisements, yellow pages, coupons, bulk contracts and the term “greedy”. The idea is to set up an economy and means for trade and then let the “invisible hand” guide the actual trading of resources.

1.9 Current Trends

In the beginning of the 80s special hardware was very popular; nowadays one tries to use ordinary high-performance workstations. New extensions in the area of memory management are beginning to emerge in the operating systems (UNIX, Windows NT) that will allow more and better control over the system’s resources. And instead of parallel computers the networked parallel computer is becoming more widespread. Special interconnect, such as SCI [IEE92], is nowadays a setting standard in high-performance network computers. However, the pioneer work on database machines has nowadays evolved into parallel data servers.

1.10 Conclusions

The need for “big” database servers will always be here; distributed cooperative solutions arise, but nethertheless local solutions will dominate.

My impressions of the experience gained from the projects mentioned above can be concluded as follows:

- In a shared-nothing architecture dataflow is a better paradigm than RPC for query processing.

- The system should be self-managing and self-balancing.
- Fault tolerance is provided by replacement of a faulty node rather than making a node fault tolerant.
- Do not build your own hardware, it will become outdated fast.
- New hardware and operating systems are error-prone; move to a stable platform.
- Do not write your own experimental implementation language.
- Do not write a compiler, you will not be able to optimize it.
- Shared-memory is easier to program than shared nothing; it does, however, not scale to many more nodes than around 32. Newer parallel computers are likely to have shared-memory (everything) processor nodes connected into a shared-nothing multi-computer.
- Use the same format for all storage of the same data: on disk, in memory, in buffers, in messages.
- Use fast-path access to data instead of stripping high-level functionalities from the DBMS.
- Parallel I/O-systems give high-performance.
- Do not implement all features and functions in all possible variants.

Chapter 2

Properties of Data Structures for Parallel Data Servers

In this Chapter we will give an introduction to required properties of data structures that are used in distributed data applicatons, such as Parallel/Distributed Data Servers. The main objectives are Scalability, Distribution and Availability. As will be noted in the conclusion, a combination of all these three features is needed.

2.1 The Problem

Modern systems manage high volumes of data, and if they implement data access paths (indices) at all, they are often hard-coded with the application's data. Data is indexed through some key identifying the data; this can efficiently be implemented by using hashing algorithms or some tree structure that keeps the data sorted. It is now well-known that most systems use mainly variants of Linear Hashing [Lit80] or B(+)-Trees [BM72] for their access paths. Other examples include R-trees (spatial), AVL-trees (main memory sorted index) and SpiralStorage.

Simple hashing algorithms often require that the number of data items to be stored is known in advance so that the correct amount of memory (i.e. slots) can be allocated for the array. When the amount of data stored in the structure grows, it eventually reaches the limit of allocated memory, which is often solved by allocating a bigger array and rehashing all the elements. Consider an interactive program that stores large amount of elements; each of them must then be rehashed. This operation will take a long time, keeping the user waiting. One alternative is to use buckets where the items are stored in lists. This leads, in the worst case — when the number of elements is much larger than the number of buckets employed — to a solution very close to linear search, thus having search time increasing with the amount of stored data. The main problems with these simplistic approaches are that during reorganizing the data is not user-accessible for a while, or that search time increases with the increasing amount of stored data. In other words, the ability for the (growing) data structure to dynamically adopt itself *smoothly* is lacking, which means that the access performance deteriorates.

2.2 Scalability

In DBMSs the need for scalable data structures is more obvious than for specialized programs. Whatever arbitrary upper limit is set on the amount of data a data structure can handle, it will probably be exceeded at some future time. A *scalable data structure* can be characterized by the following:

- Insert and retrieval time is independent of the number of stored elements (i.e., it is more or less constant).
- It can handle any amount of data, there is no theoretical upper limit that degrades the performance.
- Furthermore, it is desirable that it grows and shrinks gracefully, not having to reorganize itself totally (as some hashing structures do total rehashing of all stored elements), but rather incrementally reorganizes itself during normal processing.

Linear Hashing [Lit80] is an example of a scalable data structure, which is an algorithm for managing random access data that can dynamically grow or shrink in size. It is based on ordinary hashing schemes and has therefore the advantage of direct access, but not the limitations of a fixed array of buckets. The array is allowed to grow when the data structure reaches a certain saturation limit, or shrink when it decreases below some limit. This is achieved through splitting and merging of individual buckets. Other variants include Spiral Storage and Extensible Hashing. The access cost for hashing structures is approximately constant.

B-trees [BM72] are another example. This algorithm maintains a set of ordered data. The data is stored in leaf nodes that are allowed to store a minimum and a maximum number of elements. When the number of elements exceeds the maximum or is below the minimum, the leaf is either split or merged with other leaves. The index is maintained in the tree nodes using a similar principle, providing in the end not linear access cost but logaritmic. Other variants of scalable ordered data structures are AVL-trees, 2-3-trees, and many other treestructures.

2.3 Distribution

Sometimes the amount of data is larger than what can efficiently be managed or used by a single workstation. Even if this amount of data could be connected physically to one workstation, the processing cababilities of the workstation would not be enough for searching and processing the data. Then, instead, one can employ a *distributed data structure* that distributes the data over a number of nodes, i.e. workstations. Such a data structure can then be used to keep very large amounts of data online. One way to do this is is to apply a *hash function* on the keys that partitions the data into fragments. The simplest idea is that each fragment stores the data of one bucket, one or several fragments are then stored on each node. Some other schemas require a central directory that is visited before each retrieval or insertion of a data item to get the address of the node storing it. This solves the problem of finding where the data resides if some of it has been moved (because of reorganization). However, the directory can easily become a hot spot when many clients are accessing it. Solutions using hierarchies of distributed directories can then be used, they can then cache results of earlier requests to improve performance. This is a schema similar to the well-known internet DNS service [Ste94]. Another, simplier distribution strategy is to store one field of a record for all records in a file entirely in one node and other fields on other nodes. However, if the amount of data grows fast, this is no scalable solution. We notice that these distributed data structures will also have to be scalable over any number of storage sites. This is a relatively new concept — SDDSs — Scalable Distributed Data Structures. This will be our topic for the next chapter.

2.4 Availability

Sometimes distribution is used in combination with some redundancy to achieve *high availability*. High availability is necessary, for example in banking and telecom [Tor95] applications, but also in all other areas with mostly online transactions, or where the information is of such importance that the extra down time of reading backups cannot be allowed. Using a high availability schema, disk crashes as well as some other sources of read or write errors can then be recovered from. The classical variant here is RAIDs, *Redundant Array Independent Disks* [PGK88] where a number of disks are connected to one or several computers. One of the disks is used for storing a parity page; this page is calculated by *xor*-ing a disk-page from each disk and storing the result on the parity disk. Each time a write is being performed, this page is updated. If one of the disks fails or a page on one of the disks fails, it can be recovered (reconstructed) from the other disks. Using more disks for parity can ensure recovery from more "errors" and thus gives higher availablity.

2.5 Conclusions

New applications of databases put requirements on the memory management. Among these requirements we identified that high performance requires scalability, distribution and high availability. Data structure access has to hide the implementation of these above aspects

and make the access transparant. In this Chapter we discussed these three (orthogonal) aspects, but not the important joint usage of them. Scalable solutions will by necessity be distributed and will require high availability and new data structures are important for achieving this. We now turn to this area.

Chapter 3

SDDSs

SDDSs (Scalable Distributed Data Structures) such as a distributed variant of Linear Hashing, LH* [LNS96], and others [Dev93][WBW94][LNS94], opens up new areas of storage capacity and data access. There are three requirements for an SDDS:

- First, it should have no central directory to avoid hot-spots.
- Second, each client should have some approximate image of how data is distributed. This image should be improved each time a client makes an addressing error.
- Third, if the client has an outdated image, it is the responsibility of the SDDS to forward the data to the correct data server and to adapt the client's image.

SDDSs are good for distributed computing since they minimize the communication which minimizes the response time and enables the processor time to be used more efficiently.

The data sites termed *servers* can be used from any number of autonomous sites termed *clients*. To avoid a hot-spot, there is no central directory for the addressing across the current structure of the file. Each client has its own *image* of this structure. An image can become outdated when the file expands. The client may then send a request to an incorrect server. The servers forward such requests, possibly in several steps, towards the correct address. The correct server appends to the reply a special message to the client, called an *Image Adjustment Message* (IAM). The client adjusts its image, avoiding repeation of the error. A well-designed SDDS should make addressing errors occasional and forwards few, and should provide for the scalability of the access performance when the file grows.

3.1 Related work

In traditional distributed files systems, in implementations like NFS or AFS, a file resides entirely at one specific site. This presents obvious limitations not only on the size of the file but also on the access performance scalability. To overcome these limitations distributions over multiple sites have been used. One example of such a scheme is *round-robin* [Cor88] where records of a file are evenly distributed by rotating through the nodes when records are inserted. The *hash-declustering* [KTMO84] assigns records to nodes on the basis of a hashing function. The *range-partitioning* [DGG+86] divides key values into ranges and different ranges are assigned to different nodes. All these schemes are *static*, which means that the declustering criterion does not change over time. Hence, updating a directory or declustering function is not required. The price to pay is that the file cannot expand over more sites than initially allocated.

To overcome this limitation of static schemes, dynamic partitioning start to appear. The first such scheme is DLH [SPW90]. This scheme was designed for a shared-memory system. In DLH, the file is in RAM and the file parameters are cached in the local memory of each processor. The caches are refreshed selectively when addressing errors occur and through atomic updates to all the local memories at some points. DLH appears impressively efficient for high insertion rates.

SDDSs were proposed for distributing files in a network multi-computer environment, hence without a shared-memory. The first scheme was LH* [LNS93]. Distributed Dynamic Hashing (DDH) [Dev93] is another SDDS, based on Dynamic Hashing [Lar78]. The idea with respect to LH* is that DDH allows greater splitting autonomy by immediately splitting overflowing buckets. One drawback

is that while LH* limits the number of forwardings to two[1] when the client makes an addressing error, DDH may use $O(\log_2 N)$ forwardings, where N is the number of buckets in the DDH file.

Another SDDS has been defined in [WBW94]. It extends LH* and DDH to more efficiently control the load of a file. The main idea is to manage several buckets of a file per server while LH* and DDH have basically only one bucket per server. One also controls the server load as opposed to the bucket load for LH*.

Finally, in [KW94] and in [LNS94] SDDSs for (primary key) ordered files are proposed. In [KW94] the access computations on the clients and servers use a distributed binary search tree. The SDDSs in [LNS94], collectively termed RP*, use broadcast or distributed n-ary trees. It is shown that both kinds of SDDSs allow for much larger and faster files than the traditional ones.

[1] In theory, communication delays could trigger more forwarding [WBW94].

Chapter 4

LH*

We will now describe the LH* SDDS, and later on we describe LH*LH.

LH* is a data structure that generalizes Linear Hashing to parallel or distributed RAM and disk files [LNS96]. One benefit of LH* over ordinary LH is that it enables autonomous parallel insertion and access. The number of buckets and the buckets themselves can grow gracefully. Insertion requires one message in general and three in the worst case. Retrieval requires at least two messages, possibly three or four. In experiments it has been shown that insertion performance is very close to one message (+3%) and that retrieval performance is very close to two messages (+1%). The main advantage is that it does not require a central directory for managing the global parameters.

4.1 LH* Addressing Scheme

An LH*-*client* is a process that accesses an LH* file on the behalf of the application. An LH*-*server* at a node stores data of LH* files. An application can use several clients to explore a file. This way of processing increases the throughput, as will be shown in Section 8.2. Both clients and servers can be created dynamically.

At a server, one *bucket* per LH* file contains the stored data. The bucket management is described in Section 6.2. The file starts at one server and expands to others when it overloads the buckets already being used.

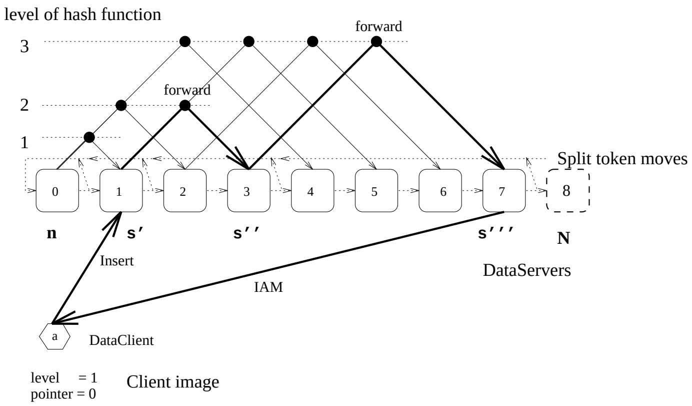


Figure 4.1: LH* File Expansion Scheme.

The global addressing rule in LH* file is that every key C is inserted to the server s_C, whose address $s = 0, 1, ...N-1$ is given by the following LH addressing algorithm [Lit94]:

$$s_C := h_i(C)$$

$$\text{if } s_C < \text{ then } s_C := h_{i+1}(C),$$

where i (LH* file level) and n (split pointer address) are file parameters evolving with splits. The h_i functions are basically:

$$h_i(C) = C \bmod (2^i \times K), K = 1, 2, ..$$

and $K = 1$ in what follows. No client of an LH* file knows the current i and n of the file. Every client has its own *image* of these values, let it be i' and n'; typically $i' \leq i$ [LNS93]. The client sends the query, for example the insert of key C, to the address $s'_C(i', n')$.

The server s'_C verifies upon query reception whether its own address s'_C is $s'_C = s_C$ using a short algorithm stated in [LNS93]. If so, the server processes the query. Otherwise, it calculates a forwarding address s''_C using the forwarding algorithm in [LNS93] and sends the query to server s''_C. Server s''_C acts as s'_C and perhaps resends the query to server s'''_C as shown for Server 1 in Figure 4.1. It is proven in [LNS93] that then s'''_C must be the correct server. In every case of forwarding, the correct server sends to the client an Image Adjustment Message (IAM) containing the level i of the correct server. Knowing the i and the s_C address, the client adjusts its i' and n' (see [LNS93]) and from now on will send C directly to s_C.

4.2 LH* File Expansion

An LH* file expands through bucket splits as shown in Figure 4.1. The next bucket to split is generally noted bucket n, $n = 0$ in the figure. Each bucket keeps the value of i used (called LH*-bucket level) in its header starting from $i = 0$ for bucket 0 when the file is created. Bucket n splits through the replacement of h_i with h_{i+1} for every C it contains. As a result, typically half of its records move to a new bucket N, appended to the file with address $n + 2^i$. In Figure 4.1, $N = 8$.

After the split, n is set to $(n+1) \bmod 2^i$. The successive values of n can thus be seen as a linear move of a *split token* through the addresses $0, 0, 1, 0, 1, 2, 3, 0, ..., 2^i - 1, 0, ...$ The arrows of Figure 4.1 show both the token moves and a new bucket address for every split, as resulting from this scheme.

4.2.1 Splitting Control Strategies

There are many strategies, called *split control* strategies, that one can use to decide when a bucket should split [LNS96] [Lit94] [WBW94]. The overall goal is to avoid the file overloading. As no LH* bucket can know the global load, one way to proceed is to fix some threshold S on a bucket [LNS96]. Bucket n splits when it gets an insert and the actual number of objects it stores is at least S. S can be fixed as a file parameter. A potentially better performance strategy for an SM environment is to calculate S for bucket n dynamically using the following formula:

$$S = M \times V \times \frac{2^i + n}{2^i},$$

where i is the n-th LH*-bucket level, M is a file parameter, and V is the bucket capacity in number of objects. Typically one sets M to some value between 0.7 and 0.9.

The intuition behind the formula is as follows. A split to a new server should occur at each $M \times V$ global insert into the data structure, thus aiming at keeping the mean load of the buckets constant;

$$\text{global number of inserts/number of server} = \text{constant}.$$

For a server without any knowledge about the other servers it can only use its own information, that is, its bucket number n and the level i, to estimate the global load. It knows that any server $< n$, server $0..n-1$, has split into server $2^i..2^i+n-1$ and both these thus have half the load of the servers that are not yet split, servers $n..2^i-1$. The number of servers can be calculated to $2^i + n$, which gives us an estimated global load of

$$M \times V \times (2^i + n).$$

Servers that were split or new servers have half the load, $S/2$, of those that are to split that have the load S. The n new servers come from n servers, totally $2 \times n$ servers with the load $S/2$, and $2^i + n - 2 \times n$ remaining servers to be split later with a load of S. The total of these servers can then be expressed as

$$\tfrac{1}{2} \times S \times 2 \times n + S \times (2^i - n).$$

This can be simplified to $S \times 2^i$. Setting the global estimate equal to the last expression provides after some simplication

$$M \times V \times (2^i + n) = S \times 2^i.$$

Solving for S gives the above expressed formula for S.

The performance analysis in Section 8.2.1 shows indeed that the dynamic strategy should be preferred in our context. This is the strategy adopted for LH*LH.

4.3 Conclusion

LH is well-known for its scalability and the new distributed LH* is also proven scalable. Both of these hashing algorithms use the actual bit representation of the hash values; these are given by the keys. Hashing in general can be seen as a radix sort in an interval where each value has a bucket where it stores the items. LH can in turn be viewed as a radix sort using the lower bits of the hash value for the keys. It furthermore has an extra attribute that tells us the number of bits used, and a splitting pointer. The splitting pointer allows gradual growth and shrinkage of the range of values (number of buckets) used for the radix sort.

LH* is a variant of LH that enables simultaneous access from several clients to data stored on several server nodes. One LH bucket corresponds to the data stored on a server node. In spite of not having a central directory, the LH* algorithm allows for extremely fast update of the client's view so that it will access the right server nodes when inserting and retrieving data. LH* [LNS93] was one of the first Scalable Distributed Data Structure (SDDSs). It generalizes LH [Lit80] to files distributed over any number of sites. One benefit of LH* over LH is that it enables autonomous parallel insertion and access. Whereas the number of buckets in LH changes gracefully, LH* lets the number of distribution sites change as gracefully. Any number of clients can be used; the access network is the only limitation for linear scaleup of the capacity with the number of servers, for hashed access. In general, insertion requires one message, and in the worst case three messages might occur. Retrieval requires one more message. But the main issue is that no central directory is needed for access to the data.

Chapter 5

spAMOS: System Architecture Framework

spAMOS is the name of our proposed system architecture where an extensible DBMS, such as AMOS[FRS93] in this case, runs on an ordinary workstation and uses computing and storage resources on a Parallel Machine[1]. spAMOS stands for *scalable parallel AMOS* [2]. Amos will be discussed briefly in Section 5.2.

The architecture uses the client-server framework. A client is logically a process that accesses some resource at a server, and a server has processing or storage capabilities for use by clients. A client can process data and can, depending on the physical environment, be run on the same processing element.

5.1 Conceptual View

The main two elements in the client-server view are, the *Frontend* workstation and a *Backend* parallel machine as shown in Figure 5.1. In the frontend, shown in Figure 5.2, a *Transaction client* (TC) accesses data using *Transaction Servers* (TS) executed in the backend. The transaction server uses a *Data Client* (DC) to access the physical data stored

[1] Here we use the term in the general sense, even including a (local) group of networked workstations.

[2] Relations to the word *spam* can also be found.

in the *Data Servers* (DS). The latter implements the server part of the SDDS and the former the client part.

A *data client* (DC) is a thread of execution which accesses information that resides on distributed *data servers* (DS). The access is achieved through the *Data Client Interface* (DCI) which is a layer of software that hides the distributed access to the distributed data. It can be extended with more operations that concern the access and manipulation of the data structure.

A *transaction server* (TS) implements algorithms such as joins, query execution and aggregation, by using methods in the DCI. So the transaction server is actually a data client, since it uses distributed data. A *transaction client* (TC) then connects to or initiates to one or several transaction servers. A transaction server can in its turn connect to other transaction servers, by connecting playing the role of a transaction client, using a *transaction client interface* (TCI).

These logical elements can be mapped nicely onto processes on separate processors in a parallel machine, but sometimes for processing purposes they could be on the same node. The idea is that transaction servers should be dynamically created as needed to perform efficient queries and/or updates of data stored in the data servers. The amount of data servers used for storing the data will also vary according to the algorithms used for the data storage (LH*LH for example).

5.2 AMOS

AMOS [FRS93] (Active Mediators Object System) is an Object-Oriented database system. It is written and optimized with memory residency in mind. Using the foreign predicates/functions interface it is easily extended to use data from various application domains. Hints can be given to the cost-based optimizer about the cost and selectivity of the defined predicates. The queries are transformed, compiled and optimized into the internal language ObjectLog, that later is interpreted. The optimizer uses a variety of algorithms to cope with the huge search space that arises in complex queries. For example randomized optimization with hill climbing is used [Näs93]. Interfacing can be done to a variety of different languages, C is the native language, but in-

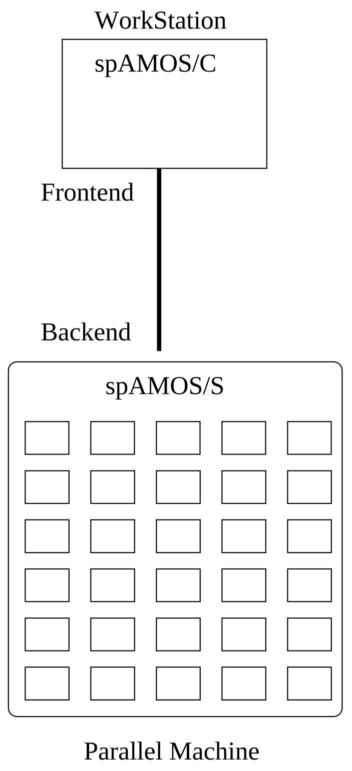


Figure 5.1: The spAMOS system; frontend workstation using a backend parallel computer.

terfaces are available from Lisp, Fortran and via a client/server tcpip interface [Wer94]. Recently AMOS has been extended with (E)CA-rules [Skö94], multi-database distribution (Query Language [Wer96]).

As we see it, AMOS is a system with the appropriate capabilities to be used for a prototype implementation of the spAMOS frontend.

5.3 A Query Example Scenario Discussion

As an example of this, we present a possible scenario in figure 5.2. We have a workstation WS, with an interconnect to a parallel machine PM. The user at the workstation is issuing a query that needs to scan one table, T1, stored in data servers DS1-DS5 for matching a predicate, P1. We assume here that this predicate is cheap to execute and reduces the output to fractions R1 of the original data stored in T1. Let us say that we then execute a computationally intense predicate P2 on the fraction R1 giving the result R2. This is done by TS1-TSn. We assume then that moving the data is cheap and allows more parallelity than executing the predicate at DS1-DS5. The results R2 are to be summed; this summation is performed after sending the data to transaction client TC0 that carries out the addition and then the transaction server TS0 sends it back to WS.

The design and implementation of the data client interface allows for possible database accesses to the data structure (LH*LH) on a parallel machine. This means that it allows both multiple files to be accessed and many clients to access the same files.

5.3.1 Notes on the query example

In this example there are several situations that indicate that there should be some planning and estimation of the costs for sending the data and calculating the predicates. We will now present some considerations about this. Since P1 is cheap and reduces the amount of data, it is cheaper to compute it at the DSs than moving the data. This reduces communication, thereby saving time. The data is then sent to TS1-TSn.

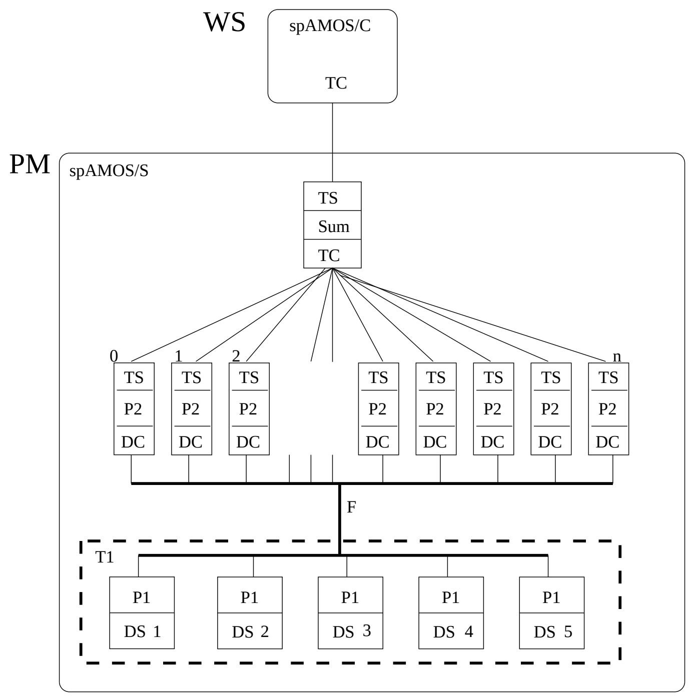


Figure 5.2: An example of a possible query evaluation.

The number of TSs is not determined but can be dynamically adapted (to availability and load). There are several possible choices of how the outgoing tuples are to be distributed: hashing on some key (LH* could then provide the means for scalability and automatic adjustability, by having a "clever" server load definition); a ticket system where the senders get a number of tickets that tells them where to send the data; a central "switch" that keeps track of which servers are free and then forwards the data. As one can see there are a number of possible alternatives here. The data is then sent from the TSs to the main TC0 that sums and returns the result to TC at WS. An alternative here is to let the TS1-TSn perform local summing and sending the partial results to TC0. This again depends on the cost of the communication. The generation and optimization of such an execution plan is beyond the scope of this thesis.

5.4 Conclusions

We have now given a sketch for a possible architecture where the resources of a parallel machine are utilized mostly for data storage and processing. Control, optimization and query initialization and result display still lie at the frontend workstation initiated by the (online) user. This, we feel, will let an ordinary workstation use the resources of a parallel machine without having to limit itself to that type of environment in the matter of operations.

Chapter 6

The LH*LH Algorithm

In this section we will go further into details about the design and implementation choices that have been made. The prototype has been made on the Parsytec machine, but can be used in any threaded (network) multi-computer.

6.1 Introduction

Below we present the LH*LH design and performance. With respect to LH* [LNS93], LH*LH is characterized by several original features. Its overall architecture is geared towards an SM (Switched Multicomputer) while that of LH* was designed for a network multicomputer. Furthermore, the design of LH*LH involves local bucket management while in [LNS93] this aspect of LH* design was left for further study. In LH*LH one uses for this purpose a modified version of main-memory Linear Hashing as defined in [Pet93] on the basis of [Lar88]. An interesting interaction between LH and LH* appears, allowing for much more efficient LH* bucket splitting. The reason is that LH*LH allows the splitting of LH*-buckets without visiting individual keys.

The average access time is of primary importance for any SDDS on a network computer or SM. Minimizing the worst case is, however, probably more important for an SM where processors work more tightly connected than in a network computer. The worst case for LH* occurs when a client accesses a bucket undergoing a split. LH* splits should be infrequent in practice since buckets should be rather large. In the basic LH* schema, a client's request simply waits at the server till the split ends. In the Parsytec context, performance measurements show that this approach may easily lead to several seconds per split, e.g. three to seven seconds in our experience (as compared to $1-2$ msec per request on the average). Such a variance would be detrimental to many SM applications.

LH*LH is therefore provided with an enhanced splitting schema, termed *concurrent splitting*. It is based on ideas sketched in [LNS96] allowing for the client's request to be dealt with while the split is in progress. Several concurrent splitting schemes were designed and experimented with. Our performance studies shows superiority of one of these schemes, termed concurrent splitting with bulk shipping. The maximal response time of an insert while a split occurs decreases by a factor of three hundred to a thousand times. As we report in what follows, it becomes about 7 msec for one active client in our experience and 25 msec for a file in use by eight clients. The latter value is due to interference among clients requesting simultaneous access to the server splitting.

The first implementation of LH* was performed using the *Parallel Virtual Machine* software, PVM [MSP93], on a number of HP workstations. The reason was mainly that the Parsytec machine at that moment was newly installed and quite unstable, thus unavailable most of the time. Later, this has partly influenced the implementation in such a way that library primitives dealing with hardware or environment specifics have been abstracted in an almost transparant way.

LH*LH allows for scalable RAM files spanning over several CPUs of an SM and its RAMs. On our testbed machine, a Parsytec GC/PowerPlus with 64 nodes of 32 MB RAM each, a RAM file can scale up to almost 2 GBytes with an average load factor of 70%. A file may be created and searched by several (client) CPUs concurrently. The access times may be about as fast as the communication network allows it to be. On our testbed, the average time per insert is as low as 1.2 ms per client. Eight clients building a file concurrently reach a throughput of 2500 inserts/second i.e., 400 μs/insert. These access times are more than an order of magnitude better than the best ones using current disk file technology and will probably never be reached by mechanical devices.

6.2 The Server

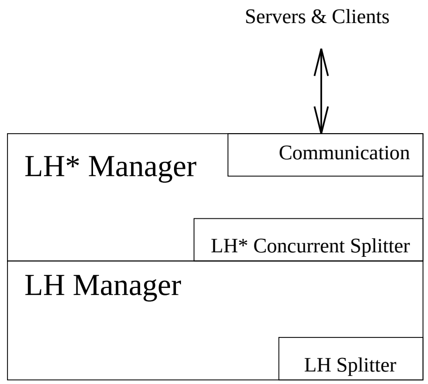


Figure 6.1: The Data Server.

The server consists of two layers, as shown in Figure 6.1a. The LH*-Manager handles communications and concurrent splits. The LH-Manager manages the objects in the bucket. It uses the Linear Hashing algorithm [Lit80].

6.2.1 The LH Manager

LH creates files able to grow and shrink gracefully on a site. In our implementation, the LH-manager stores all data in the main memory. The LH variant used is a modified implementation of Main Memory Linear Hashing [Pet93].

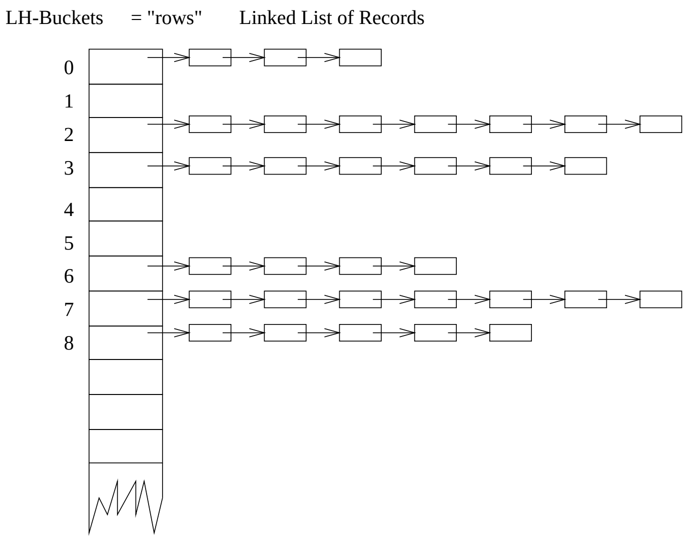


Figure 6.2: The LH-structure.

The LH file in an LH*-bucket (Figure 6.2b) essentially contains (i) a header with the *LH-level*, an *LH-splitting pointer*, and the count x of objects stored, and (ii) a dynamic array of pointers to LH-buckets, and (iii) LH-buckets with records. An LH-bucket is implemented as a linked list of the records. Each record contains the calculated hash value (*pseudo-key*), a pointer to the key, and a pointer to a BLOB. Pseudo-keys make the rehashing faster. An LH-bucket split occurs when $L = 1$, with:

$$L = \frac{x}{b \times m},$$

where b is the number of buckets in the LH file, and m is a file parameter being the required mean number of objects in the LH-buckets (linked list). Linear search is most efficient up to an m about 10.

6.2.2 LH* Partitioning of an LH File

The use of LH allows the LH* splitting in a particularly efficient way. The reason is that individual records of the buckets are not visited for rehashing. Figure 6.3 and Figure 6.4 illustrate the ideas.

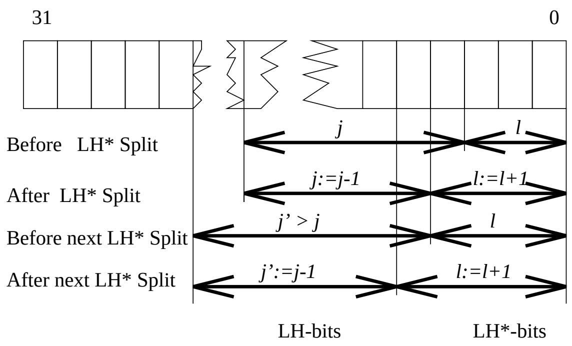


Figure 6.3: Pseudo-key usage by LH and LH*.

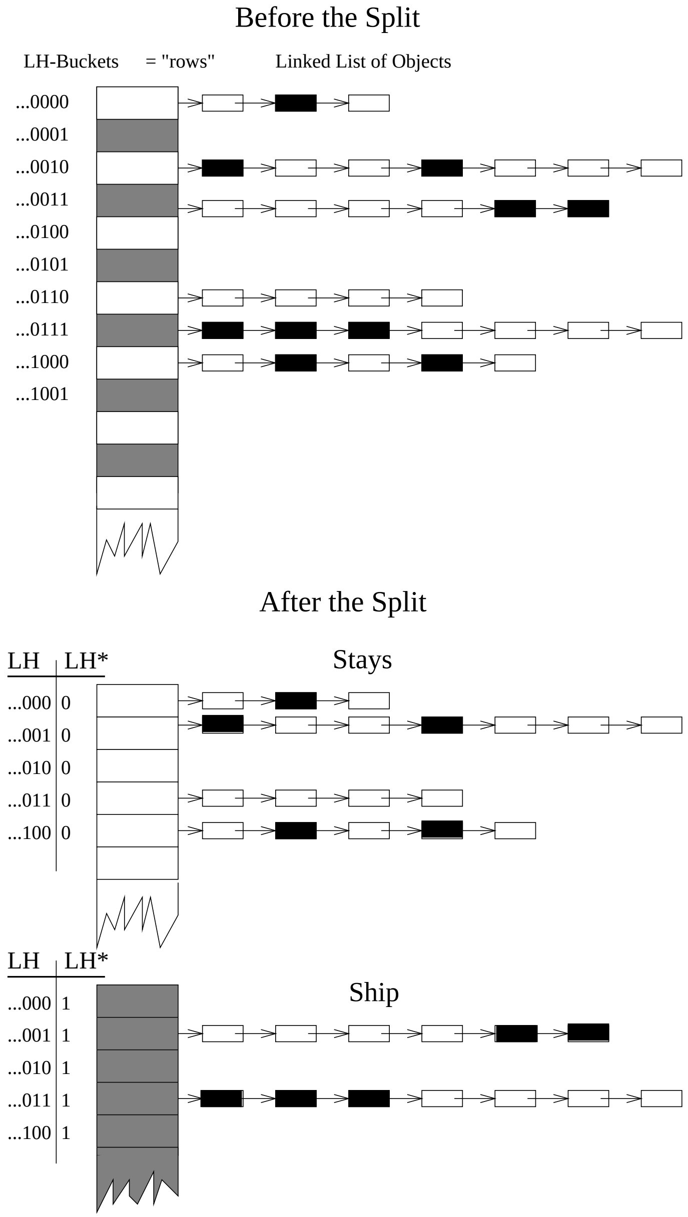


Figure 6.4: Partitioning of an LH-file by LH* splitting.

LH and LH* share the pseudo-key. The pseudo-key has J bits as shown in Figure 6.3; $J = 32$ at every bucket. LH* uses the lower l bits $(b_{l-1}, b_{l-2}, \ldots b_0)$. LH uses j bits $(b_{j+l-2}, b_{j+l-3}, \ldots b_l)$, where $j + l \leq J$. During an LH*-split l increases by one, whereas j decreases by one. The value of the new lth bit determines whether an LH-bucket is to be shipped. Only the odd LH-buckets, i.e. with $b_l = 1$, are shipped to the new LH*-bucket N. The array of the remaining LH-buckets is compacted, the count of objects is adjusted, the LH-bucket level is decreased by one (LH uses one bit less), and the split pointer is halved. Figure 6.4 illustrates this process.

Further inserts to the bucket may lead to any number of new LH splits, increasing j as shown in Figure 6.3 to some j'. The next LH* split of the bucket will then decrease j' to $j' := j' - 1$, and set $l := l + 1$ again.

6.2.3 Concurrent Request Processing and Splitting

A split is a much longer operation than a search or an insert. The split should also be atomic for the clients. Basic LH* [LNS93] simply requires the client to wait till the split finishes. For high-performance applications on an SM multi-computer it is fundamental that the server processes a split concurrently with searches and inserts. This is achieved as follows in LH*LH.

Requests received by the server undergoing a split are processed as if the server had not started splitting, with one exception: a request that concerns parts of the local LH structure already shipped is queued to be processed by the Splitter. The Splitter processes the queue of requests since these requests concern LH-buckets of objects that have been or are being shipped. If the request concerns an LH-bucket that has already been shipped, the request is forwarded. If the request concerns an LH-bucket not yet shipped, it is processed in the local LH table as usual. The requests that concern the LH-bucket that currently is being shipped first search the remaining objects in the LH-bucket. If not found there, they are forwarded by the Splitter. All forwardings are serialized within the Splitter task. More detailed information of the algorithm and the possible choices in implementation are given in Section 9.3.1.

6.2.4 Shipping

Shipping means transferring the objects selected during the LH*-bucket split to the newly appended bucket N. In LH* [LNS96] the shipping was assumed basically to be of the *bulk* type with all the objects packed into a single message. After shipping has been completed, bucket N sends back a *commit message*. In LH*LH there is no need for the commit message. The Parsytec communication is safe and the sender's data cannot be updated before the shipping is entirely received. In particular, no client can directly access bucket N before the split is complete.

In the LH*LH environment there are several reasons for not shipping too many objects in a message, especially all the objects in a single message. Packing and unpacking objects into a message require CPU time and memory transfers, as objects are not stored contiguously in memory. One also needs buffers of sizes at least proportional to the message size, and a longer occupation of the communication subsystem. Sending objects individually simplifies these aspects but generates more messages and more overhead time in the dialog with the communication subsystem. It does not seem that one can decide easily which strategy is finally more effective in practice.

The performance analysis in Section 8.2.2 motivated the corresponding design choice for LH*LH. The approach is that of bulk shipping but with a limited message size. At least one object is shipped per message and at most one LH-bucket. The message size is a parameter allowing for an application-dependent packing factor. The test data using bulks of a dozen records per shipment showed to be much more effective than individual shipping.

6.3 Notes on LH*LH Communications

In the LH*LH implementation on the Parsytec machine a server receiving a request must have issued the *receive* call before the client can do any further processing. This well-known *rendezvous* technique

enforces entry flow control on the servers, preventing the clients from working much faster than the server can accept requests[1] . There is no need to make the insert operations provide a specific acknowledge message, since communication is "safe" and therefore not needed on the Parsytec machine. IAMs, split messages with the split token, and general service messages use the asynchronous type of communication to remove the possibility of deadlocks. We avoid deadlocking by never letting the servers communicate synchronously with a server having a lower logical number. When a data client requests data from a data server, it must receive the answer directly without engaging in any other communication, since the server otherwise would be blocked. This is enforced in the data client interface by making look-up operations atomic.

The choice of synchronous communication for normal communication, for example not IAMs and similar control messages, does, however, not mean that the requests on the datastructure client must be synchronous. That is, when a client issues the operation insert, the client only waits till the message has been delivered to the server, then both the server and the client can continue the processing. Presumably the server executes the insert operation internally, but the client does not wait for any acknowledgement of the completion of the operation.

6.3.1 Communication Patterns

A parallel machine has a communication *topology* that is inherited from the hardware interconnect. For example, if a number of nodes are interconnected in a ring communicating only in one direction, it is obvious that problems that communicate in the same manner benefits from such a topology. In this case, a program that performs pipe-lined execution would suit well, but a program where all nodes communicate with the other nodes in a star pattern would lose. This means that if a program communicates in such a way that it can directly use the

[1] The overloaded server can run out of memory space and then send outdated IAMs; this is a fact when using PVM [MSP93] together with many active clients on a few servers. There is no flow-control when sending. Messages were stored internally by PVM, and the receiving process eventually grew out of memory. This indicates the need for data flow control

underlying topology minimizing the number of hops the messages have to traverse, it will be efficient. Many general-purpose parallel machines try to mimic different topologies by imposing a *virtual topology* onto the real topology. In the case of having a grid-type (mesh) interconnected multi-computer, a Parsytec for instance, the pipe-topology can easily be implemented by a clever allocation of the nodes that mimimize the communication hops. Other types of topologies [LER92] include Star Network, Toroidal Mesh, Tree Network, Ring Network, Fully Connected Network, and N-Dimensional Hypercubes.

The question in our case is whether LH*LH would benefit from any such topology; the answer depends very much on what kind of operations dominates its use. If a known number of clients access data at a number of servers, they could be placed in such a way that the total communication paths were minimized. Also, servers could be placed in a Tree Network to minimize communication when broadcasting requests, such as scanning, to an entire LH*LH-file. One interesting fact to take into account is the way the Parsytec routes messages. It routes them in a static way, always the same route from one node to another, first horizontal and then vertical. An unfortunate allocation could place data clients in such a way that they interfere by using the same static routes for all communication. We leave this area open for research.

An SDDS internal communication has a fixed pattern for communication, which stems from the splitting and forwarding strategy; in the LH* this is a tree-like structure as can be seen in Figure 4.1. There is, however, no special pattern for the clients accessing the data structure, and then no natural special static topology that could be used for the LH* algorithms. However, in the Parsytec environment, static communication links must be established to use the fastest means of communication. Therefore, when the program is started, we establish links from each node to each of the other nodes for the whole machine. Two links are in fact established — one for data messages and one for control messages. The latter is given higher priority in the program. The rationale for this is that, for example, a split message or a token should not be unnecessarily delayed to the receiver because this could badly influence operation. Mailbox communication [2], that is semantically

[2]Sometimes also referred to as *store-and-forward*.

better suited for SDDSs was found to be both slower and unreliable in the Parsytec environment.

Other communication needs arise when applying reduction, such as summing and scanning implementation. It is then natural to use some sort of “limited” broadcasting or multicasting. Multicasting sends the same message to a group of specified machines. It is favourable for initializing scanning operations. On a Parsytec this has to be implemented by point-to-point primitives. Simply implementing scanning by just iterating over a complete list of all known Data Servers is not feasible since, among other things, it would require the list and its size to be known in advance or updated during the messaging. An easy way to acomplish is to use the hidden tree structure that we discussed earlier which is formed during the splitting of the LH* nodes, this being a parent node, and new nodes can be seen as children nodes. In this way the problem is easily distributed and the time complexity decreases from $O(n)$ to $O(log(n))$, where n is the number of data servers. Also, this efficiently takes care of the problems with presplit or merged servers since the parent knows all about its children.

Chapter 7

Hardware Architecture

The Parsytec GC/PowerPlus architecture (Figure 7.1) is massively parallel with distributed memory, also know as MIMD (Multiple Instruction Multiple Data). The machine used for the LH*LH implementation has 128 PowerPC-601 RISC-processors, constituting 64 nodes. One node is shown in Figure 7.1a. Each node has 32 MB of memory shared between two PowerPC processors and four T805 Transputer processors. The latter are used for communication. Each Transputer has four bidirectional communication links. The nodes are connected through a bidirectional fat (multiple) grid network with packet message routing.

LH* was first implemented on an HP workstation in order to test some aspects of distribution. Then LH*LH was implemented on the Parsytec GC machine, keeping in mind and reusing parts from the first LH* implementation. Message-handling is encapsulated to facilitate porting to other physical (parallel) architectures. Some experiments have been made on the handling of data stored locally in Data Servers in order to improve file growth by splitting hash buckets in background processing.

7.1 Communication

The communication is point-to-point, and the software libraries supports both the synchronous and asynchronous communication [Par94].

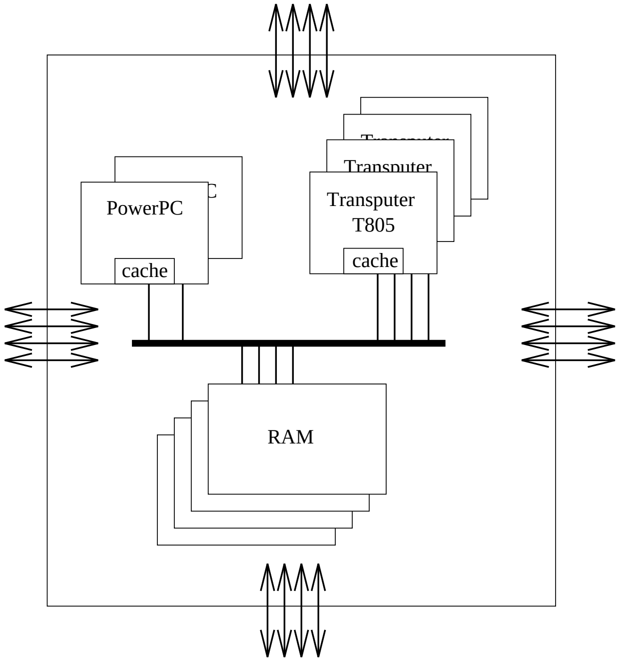


Figure 7.1: One node on the Parsytec machine

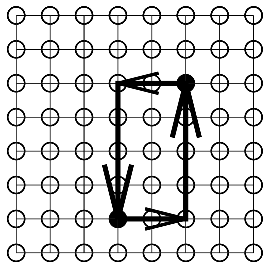


Figure 7.2: Static routing on a 64 nodes machine between two nodes.

Connections can be established using links, optionally on a virtual topology. Mailbox-communication is also available, which is the most general form of communication, but is limited due to the store-and-forward principle used. Broadcasts and global exchange are implemented using the above presented forms of communication presented above.

The number of virtual connections is not limited by the number of physical connections. The virtual connections can be used to implement a hand-crafted topology of the processors. There are also predefined libraries for the most common topologies such as pipe, 2- dimensional, 3-dimensional, 2-, 3-torus, tree and hyper-cube.

The response time of a communication depends on the actual machine topology. The closer the communicating nodes are, the faster is the response. Routing is done statically by the hardware as in Figure 7.2b with the packages first routed in the horizontal direction.

Chapter 8

Performance Measures

8.1 Measure Suite

We performed the tests using 32 nodes since, at the time when the tests were performed, only 32 nodes were available at our Parsytec site. The Clients were allocated downwards from node 31, while servers were allocated from node 0 and upwards, as shown in Figure 8.1. The Clients read the test data (a random list of words) into main memory in advance to avoid the I/O disturbing the measurements. Then the clients started inserting their data, creating the test LH*LH-file. When a client sent a request to the server, it continued with the next item only when the request had been accepted by the server (rendezvous). Each time, just before the LH*LH file was split, measures were collected by the splitting server. Some measurements were also collected at some clients, especially timing values for each of that client's requests.

8.2 Performance Evaluation

The access performance of our implementation was studied experimentally. The measurements below show elapsed times of various operations and the scalability of the operations. Each experiment consists of a series of inserts creating an LH* file. The number of clients, the file parameters M and m, and the size of the objects are LH*LH parameters.

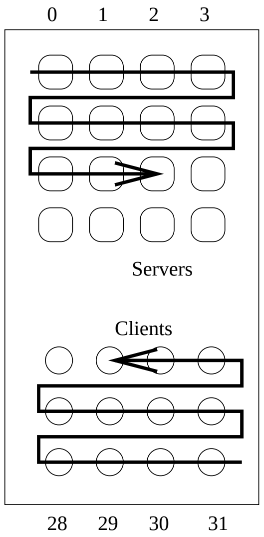


Figure 8.1: Allocation of servers and clients.

8.2.1 Scalability

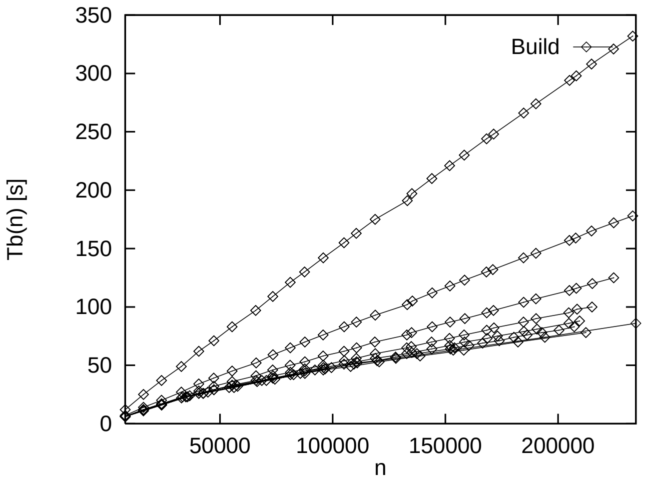


Figure 8.2: Build time of the file for a varying number of clients.

Figure 8.2 plots the elapsed time taken to constitute the test LH*LH file through n inserts'; $n = 1, 2..N$ and $N = 235.000$, performed simultaneously by k clients $k = 1, 2..8$. This time is called *build time* and is noted $Tb(n)$, or $Tb^k(N)$ with k as a parameter. In Figure 8.2, $Tb(N)$ is measured in seconds. Each point in a curve corresponds to a split. The splits were performed using the concurrent splitting with the dynamic control and the bulk shipping. The upper curve is $Tb^1(n)$. The next lower curve is $Tb^2(n)$, and so on until $Tb^8(n)$.

The curves show that each $Tb^k(n)$ scales-up about linearly with the file size n. This is close to the ideal result. Also, using more clients to build the file uniformly decreases Tb^k, i.e.,

$$k' > k'' - > Tb^{k'}(n) \leq Tb^{k''}(n),$$

for every n. Using two clients almost halves Tb, especially $Tb(N)$, from $Tb^1(N) = 330$ sec to $Tb^2(N) = 170$ sec. Building the file through eight clients decreases Tb further, by a factor of four. $Tb(N)$ becomes only $Tb^8(N) = 80$ sec. While this is in practice an excellent performance, the ideal scale-up should reach k times, i.e., the build time $Tb^8(N) = 40$ sec only. The difference results from various communication and processing delays at a server shared by several clients, as discussed in previous sections and in what follows.

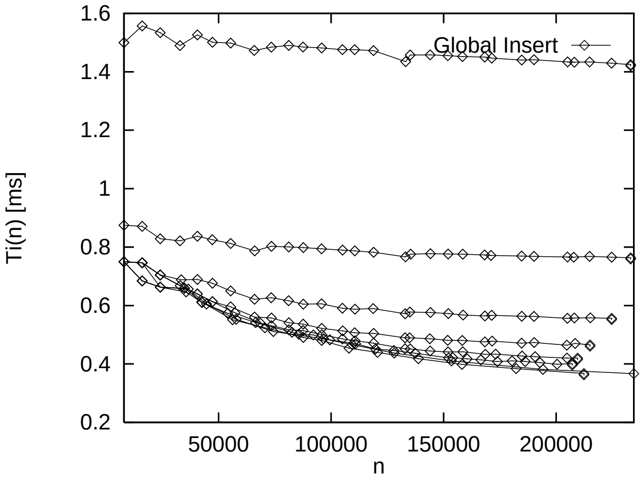


Figure 8.3: Global insert time measure at one client, varying the number of clients.

Figure 8.3 plots the curves of the global insert time

$$Ti^k(n) = Tb^k(n)/n[msec].$$

Ti measures the average time of an insert from the perspective of the application building the file on the multi-computer. The internal mechanics of the LH*LH file is transparent at this level including the distribution of the inserts among the k clients and several servers, the corresponding parallelism of some inserts, the splits and so on. The values of n, N and k are those shown in Figure 8.2. Increasing k improves Ti in the same way as for Tb. The curves are also about as linear, constant in fact, as they should be. Highly interestingly, and perhaps unexpectedly, each $Tb^k(n)$ even decreases when n grows, the gradient increasing with k. One reason is the increasing number of servers of a growing file, leading to fewer requests per server. Also, our server and client node allocation schema decreases the mean distance through the net between the servers and the clients of the file.

The overall result is that Ti is always under 1.6 msec. Increasing k uniformly decreases Ti, until $Ti^8(n) < 0.8$ msec for all n, and $Ti^8(N) < 0.4$ msec in the end. These values are about ten to twenty times smaller than access times to a disk file, typically over 10 msec per insert or search. They are likely to remain forever beyond the reach of any storage on a mechanical device. On the other hand, a faster net and a more efficient communication subsystem than the one used should allow for even much smaller Ti's, in the order of dozens of microseconds [LNS96] [LNS94].

Figure 8.4 plots the global throughput $T^k(n)$ defined as

$$T^k(n) = 1/Ti(n)[i/sec](insertspersecond).$$

The curves express again an almost linear scalability with n. For the reasons discussed above, T^k even increases for larger files, up to 2700 i/sec. An increase of k also uniformly increases T for every n. To see the throughput scalability more clearly, Figure 8.5 illustrates a plot of the relative throughput

$$Tr(k) = T^k(n)/T^1(n),$$

for a large n, $n = N$. One compares Tr to the plot of the ideal scale-up that is simply $T'r(k) = k$. The communication and service delays we spoke about clearly play an increasing role when k increases. Although Tr monotonically increases with k, it diverges more and more from $T'r$.

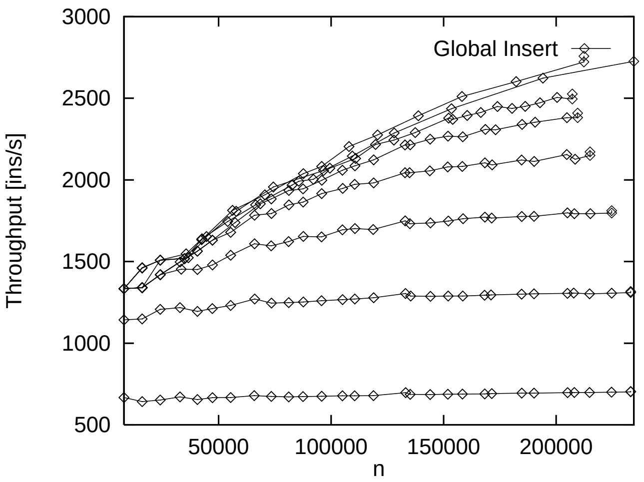


Figure 8.4: Actual throughput with varying number of clients.

For $k = 8$, $Tr = 4$ which is only the half of the ideal scale-up. It means that the actual throughput per client,

$$Tc^k(n) = T^k(n)/k,$$

also comparatively decreases to half of the throughput T^1 of a single client.

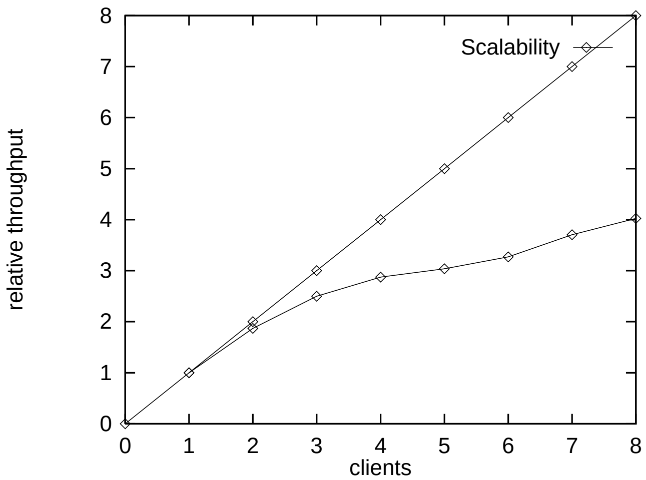


Figure 8.5: Ideal and actual throughput with respect to the number of clients.

Figure 8.6 and Figure 8.7 show the comparative study of the dynamic and the static split control strategies. The plots show build times, with $Tb'(n)$ for the static control and $Tb(n)$ for the dynamic one. The curves correspond to the constitution of our example file, with $k = 1$ in Figure 8.6 and $k = 4$ in Figure 8.7. The plots Tb are the same as in Figure 8.2. Figure 8.6 and Figure 8.7 clearly justify our choice of

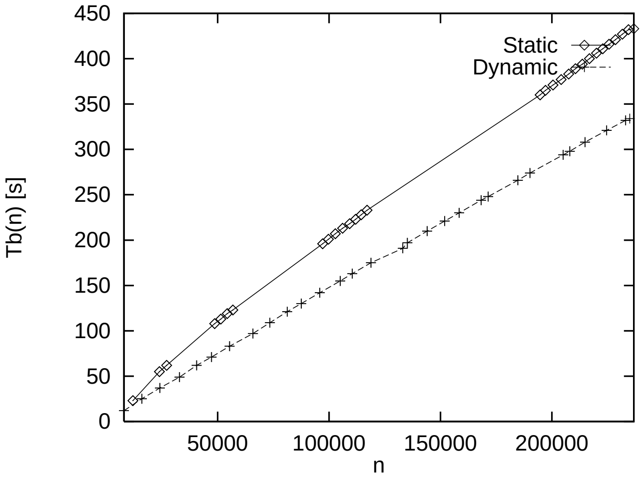


Figure 8.6: Comparison between Static and Dynamic splitting strategy, one client.

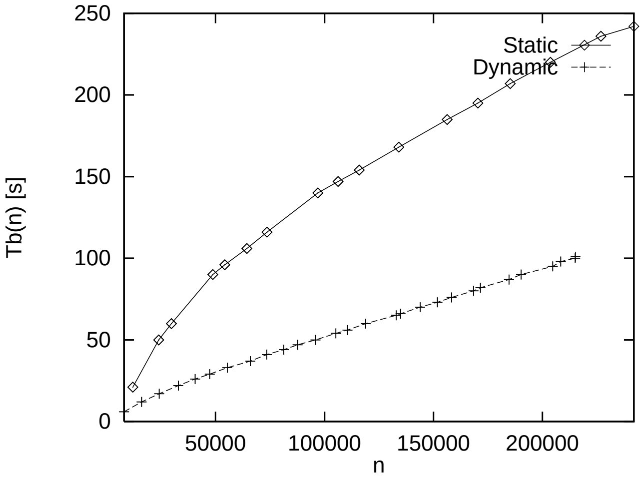


Figure 8.7: Comparison between Static and Dynamic splitting, with four clients.

the dynamic control strategy. Static control uniformly leads to a longer build time, i.e., for every n and k one has $Tb'(n) > Tb(n)$. The relative difference $(Tb' - Tb)/Tb$ reaches 30% for $k = 1$, e.g. $Tb'(N) = 440$ and $Tb(N) = 340$. For $k = 4$ the dynamic strategy more than halves the build time, namely from 230 to 100 sec.

Note that the dynamic strategy also generates splits generally more uniformly over the inserts, particularly for $k = 1$. The static strategy leads to short periods when a few inserts generate splits of about every bucket. This creates a heavier load on the communication system and increases the insert and search times during this period.

8.2.2 Efficiency of Concurrent Splitting

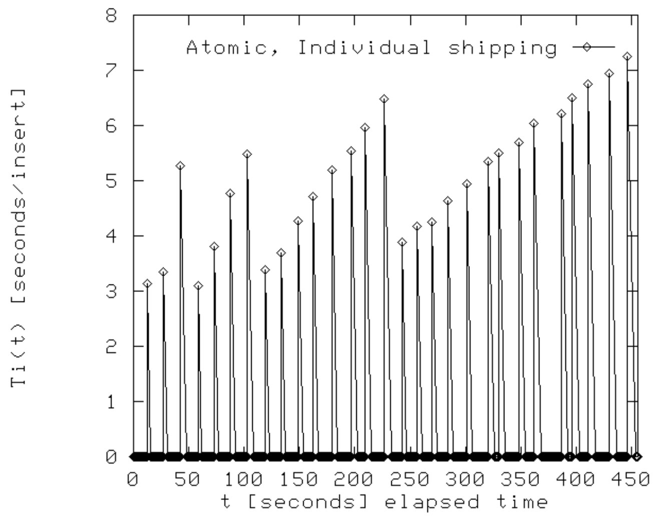


Figure 8.8: Efficiency of individual shipping.

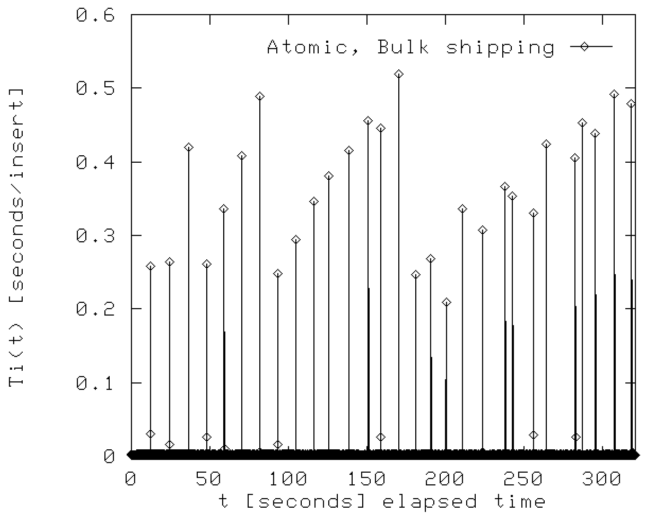


Figure 8.9: Efficiency of bulk shipping.

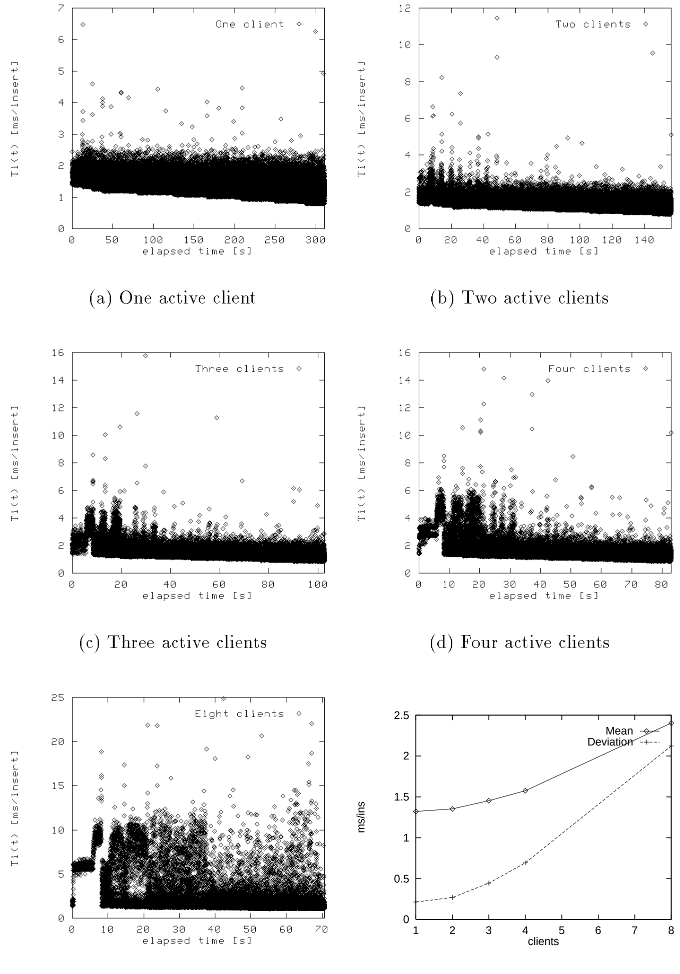
Figure 8.8 shows the study of the comparative efficiency of individual and bulk shipping for LH* atomic splitting (non-concurrent), as described earlier. The curves plot the insert time $Ti^1(t)$ measured at t seconds during the constitution of the test file by a single client. A bulk message contains at most all the records constituting an LH-bucket to ship. In this experiment there are 14 records per bulk on the average. A peak corresponds to a split in progress, when an insert gets blocked till the split ends.

The average insert time beyond the peaks is 1.3 msec. The corresponding Ti's are barely visible at the bottom of the plots. The individual shipping, shown in Figure 9a, leads to a peak of $Ti = 7.3$ sec. The bulk shipping plot, Figure 8.9, shows the highest peak of $Ti = 0.52$ sec, i.e., 14 times smaller. The overall build time $Tb(N)$ also decreases by about 1/3, from 450 sec (Figure 8.8), to 320 sec (Figure 8.9). The figures clearly prove the utility of the bulk shipping.

Observe that the maximal peak size was reduced according to the bulk size. This means that larger bulks improve the access performance. However, such bulks also require more storage for themselves as well as for the intermediate communication buffers and more CPU for the bulk assembly and disassembly. To choose the best bulk size in practice, one has to weigh all these factors depending on the application and the hardware used. However, no bulk message contain more that one LH-bucket currently, but the LH*LH algorithm can be extened to ship more buckets[1]. A more attractive limit is the size of the buffer to be used, rather than the number of records.

Figure 8.10 shows the results of the study where the bulk shipping from Figure 8.9 is finally combined with the concurrent splitting. Each plot $Ti(t)$ shows the evolution of the insert time at one selected client among k clients; $k = 1..4, 8$, concurrently building the example file with the same insert rate per client. The peaks at the figures correspond again to the splits in progress but they are much lower. For $k = 1$, they are under 7 msec, and for $k = 8$ they reach 25 msec. The worst insert time with respect to Figure 8.9 improves thus by a factor of 70 for $k = 1$ and of 20 for $k = 4$. This result clearly justifies the utility of the

[1]This might not be so attractive, since the algorithm gets to be slightly more complicated and the bulk messages too big.



(a) One active client

(b) Two active clients

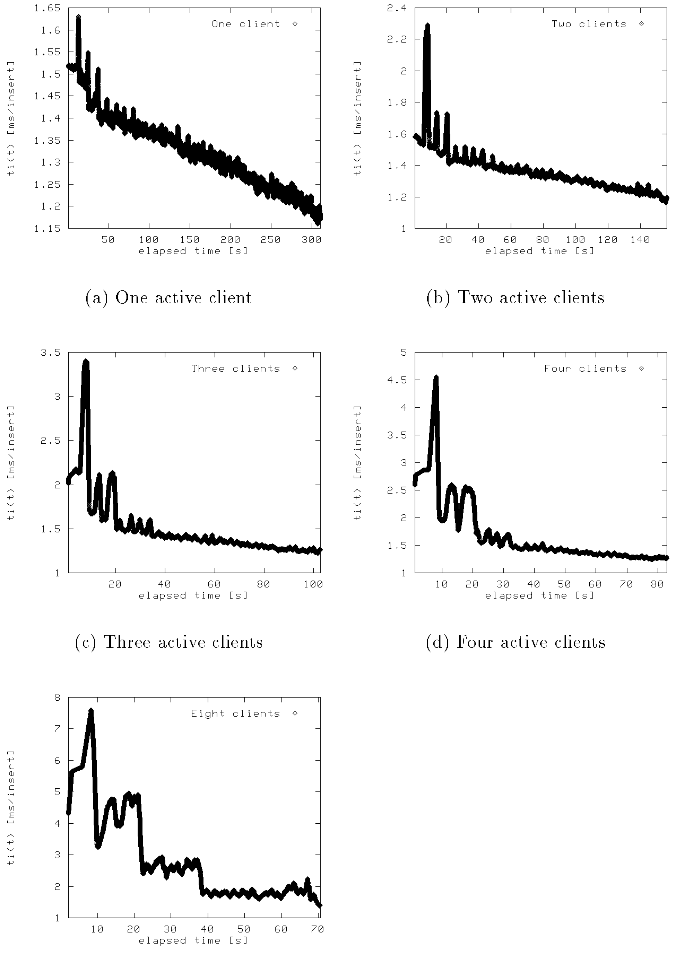
(c) Three active clients

(d) Four active clients

(e) Eight active clients

(f) Average, std. deviation

Figure 8.10: Efficiency of the concurrent splitting.



(a) One active client

(b) Two active clients

(c) Three active clients

(d) Four active clients

(e) Eight active clients

Figure 8.11: LH*LH client insert time scalability.

concurrent splitting and our overall design of the splitting algorithm of LH*LH.

The plots shown in Figures 8.10a to 8.10e show the tendency towards higher peaks of Ti, as well as towards higher global average and variances of Ti over $Ti(t)$, when more clients build the file. The plot in Figure 8.10f confirms this tendency for the average and the variance. Figures 8.10d and 8.10e also show that the insert times become especially affected when the file is still small, as one can see for $t < 10$ in these figures. All these phenomena are due to more clients per server for a larger k. A client has then to wait longer for the service. A greater k is nevertheless advantageous for the global throughput as was shown earlier.

Figure 8.10 hardly illustrates the tendency of the insert time when the file scales up, as non-peak values are buried in the black areas. Figure 8.11 plots, therefore, the evolution of the corresponding *marginal client insert time* Tm^k. Tm^k is computed as an average over a sliding window of 500 inserts plotted, as can be seen in Figure 8.10. The averaging smoothes the variability of successive values giving the black areas in Figure 8.10. The plots $Tm^k(t)$ show that the insert times not only do not deteriorate when the file grows, but even improve. Tm^1 decreases from 1.65 msec to under 1.2 msec, and Tm^8 from 8 msec to 1.5 msec. This satisfactory behavior is due again to the increase in the number of servers and to the decreasing distance between the clients and the servers.

The plots show also that $Tm^k(t)$ uniformly increases with k, i.e.

$$k'' > k' \rightarrow Tm^{k''}(t) > Tm^{k'}(t),$$

for every t. This phenomenon is due to the increased load of each server. Another point of interest is that the shape of Tm^k becomes stepwise, for greater k's, with insert times about halving at each new step. A step corresponds to a split token trip at some level i. The drop occurs when the last bucket of level i splits and the split token comes back to bucket 0. This tendency seems to show that the serialization of inserts contributing most to a Tm^k value occurs mainly at the buckets that are not yet split.

The overall conclusion from the behaviour illustrated in Figure 8.11 is that the insert times at a client of a file equally shared among k

clients is basically either always under 2 msec, for $k = 1$, or tends to be under this time when the file enlarges. Again this performance shows excellent scale-up behavior of LH*LH. The performance is in particular greatly superior to the one of a typical disk file used in a similar way. For $k = 8$ clients, for example, the speed-up factor could reach 40 times, i.e., 2 msec versus $8 * 10$ msec.

8.3 Curiosity

To test the performance of LH*LH on the Parsytec machine, we made a large number of batch runs, where numerous different data sets were collected. Using advanced scripts, we automatically filtered, calculated different scaleup curves, and finally plotted them to files. Many of the plots contain so many measure points that gif-files had to be used instead of postscript drawings, which decreased the size of such a plot from around 1 MB to 30 KB.

During the process of analyzing the data the AMOS system [FRS93] was experimentally used. Data was imported and the AMOSQL [KLR+94] query language could be used to construct plots on-the-fly. As an example of such a query, we plot the throughput curve from 1 server to 8 servers:

```
create function Speed(Measure m)->real as
        select operations(m)/Elapsed(m);

plot((  select servers, maxspeed
        for each integer servers, real maxspeed
        where maxspeed =
          maxagg(Speed(FSer(servers,
                       FQM("W",
                       FPC(32,
                       Measures())))))
        and   servers = iota(1, 8),
        "X-Servers, Y-MaxSpeed, Write, PC=32");
```


First, we declare a derived function `Speed(measure)` that takes a measure point and calculates the number of `operations(measure)` per second `Elapsed(measure)`. Then we `plot(bag of <x,y>, header string)`. The actual plotting is done by automatically calling the gnuplot program. The bag of tuples `<x,y>` is constructed through an AMOSQL query, where `server` is the number of servers, plotted along the x-axis, and `maxspeed` is the maximum `Speed` for that number of servers writing "`(FQM("W", ...))`", using 32 nodes "`(FPC(32, ...))`".

8.4 Conclusion

Switched multi-computers such as the Parsytec GC/PowerPlus are powerful tools for high-performance applications. LH*LH was shown to be an efficient new data structure for such multi-computers. Performance analysis showed that access times may be in general of the order of a millisecond, reaching 0.4 msec per insert in our experiments, and that the throughput may reach thousands of operations per second, over 2700 in our study, regardless of the file scale-up. An LH*LH file can scale-up over as much of distributed RAM as available, e.g., 2 Gbytes on the Parsytec, without any access performance deterioration. The access times are in particular an order of magnitude faster than one could attain using disk files.

Performance analysis also confirmed various design choices made for LH*LH. In particular, the use of LH for the bucket management, as well as of the concurrent splitting with the dynamic split control and the bulk shipping, effectively reduced the peaks of response time. The improvement reached a thousand times in our experiments, from over $7sec$ that would characterize LH*, to under 7 msec for LH*LH. Without this reduction, LH*LH would likely to be inadequate for many high-performance applications.

Chapter 9

LH*LH Implementation

We will now go through the inner workings in the data server and the data client. This involves how messages are sent, addressed, received and then dispatched to the actual data structure. We start with the initialization of the system, and then we describe the client. As far as the client is concerned we discuss what it does, why and how. Then we turn to the server continuing the discussion. Communication choices are explained mainly in the Parsytec environment. We will briefly mention experience from an early implementation in the PVM environment [MSP93], too.

9.1 The System Initialization

The first thing that is done during the initialization of the system is that the nodes calculate their logical machine address. This is currently an integer number; in a larger environment this would require a mapping from this number to a more physical number, for example an IP-address. In PVM a simple protocol is used to calculate this number in a negotiating processes with the other nodes, whereas on the Parsytec the number is given by the operating system. This number is not directly used by the LH*LH data structure but instead there is a mapping for each LH*LH file from a logical data server number to a (machine) node logical number. This is referred to as *Server Mapping*. How such mappings can be implemented is discussed in Section 9.4.

In the Parsytec environment, during this phase we set up communication links between all nodes in all directions. This to be able to use the fastest means of communication. There are actually two links set up in both directions. One for request and data communication, and one for control messages. Examples of control messages include forcing a split to a new server, and counting the servers. The reason for this is that some messages should have a higher priority, and there is no way to receive these, in a simple way, on a link without accepting all messages and then having to store them. The control messages are always checked and executed first, and they should be sparse compared to other communication. This also allows us to receive request and data messages with entry control flow, implemented using a limited receive buffer. This means that the server node can hold a limited number of received requests in a queue and process them one by one. It not only allows smoother operations on the client side if a server at some point of time happens to receive more requests than usual, but it also prohibits overloading a server.

Experience shows that, for example the PVM communication package has no other limit than available memory on the amount of messages it will receive without the receiver program code asking for any messages. This is not good for several reasons. First, memory overflow can easily occur at the receiver when clients are sending requests faster than the server can handle them. Second, the probability that clients at a node send messages to the wrong server increases since the IAM will not be sent before the server has handled the request. In the PVM environment there are no guarantees that any communication blocks[1]; the PVM communication package may (and so it did) receive messages till it exhausts memory, even if the program at the receiving end does not ask for any messages, i.e., executes a receive operation; this also increased the number of IAMs since requests queued up.

Both of these aspects are implemented in an interchangeable layer where the specific code for a machine and operating system environment resides.

It is currently assumed that all the data clients and data servers run

[1]That is, the send operation returns only when the data has been received at the other end, e.g. synchronous communication.

on "equal" computers. By this we mean that they form a networked multi-computer, that is, a homogeneous environment as regards communication software and libraries. However, to be able to work in a heterogeneous environment is just a matter of changing a well-defined layer in the software in such a way that it behaves compatibly.

9.2 The Data Client

A data client is an execution thread. It accesses data stored on clusters of data servers in (LH*LH) files and this can be extended to other distributed (scalable) datatypes as well. Several data clients can cooperate in fulfilling the same goal, i.e. a search can be split by a program into several data clients that use the data client interface that then access, process and collect the information and then send it to the requester. There can be several data clients on one node using different threads. All these can then jointly benefit from the same Image Adjust Messages that other data clients at the same node received.

The data client uses the *Data Client Interface* with which it can access any LH*LH file stored in the network computer. A file is identified by a `TableID` and a `ServerList`; the latter contains the Server Mapping mentioned earlier, and the former is a unique integer number that identifies the distributed file. When a file is opened, a pointer to a handle structure is returned. This handle identifies the file and stores the current client image state of that file. Thus, if the same handle for the file is used by more than one client thread, the updates will benefit all of them. This handle is then used in all calls to the data client interface.

9.2.1 Function Outline

When a client performs an insert operation on a file, the following operations will occur internally in the data client interface. First, the pseudo-key is calculated using a hashing function. Using the pseudo-key and the client image for that file which is stored in the data accessible through the file handle, the client calculates the logical number of what it believes is the appropriate data server that should store the data.

This number is then mapped to the machine node number. A message is then assembled containing the identity number of the file, the key, the data as a blob of bytes, its associated size in number of bytes, and the client image[2]. This message is then directly sent to the calculated destination server.

In the Parsytec environment the synchronous communication and our entry flow control ensure that the message will not be transferred before the server actually issues a receive. After the message has been received by the calculated destination server, the data client code directly returns to the caller if it was an insert operation. Otherwise, if the operation requires an answer from a server, it blocks and awaits the message. It returns when the answer is received. The client can then directly issue a new operation on the distributed file. Several client threads can individually communicate with the file without disturbing each others' operations.

9.2.2 Image Adjust Messages

The client image variables should be adjusted when the client makes an addressing error. It is the responsibility of the LH*LH file servers to send a message that corrects the client. In LH* this message is sent by the final receiving server when it identifies the that the message has been forwarded to it. However, in LH*LH we have chosen to let a server that forwards the request also send the update message to the client. There are some advantages and disadvantages with this approach. The number of image adjust messages (IAMs) will increase for new clients, since forwarding can occur a couple of times and each of them will yield a new IAM. On the other hand, since each forwarding takes a while, the client will get the message back and can then adjust its image *before* the final recipient server has processed the request, thus the next request of a highly active node will use a more appropriate destination. This will not only reduce the delay before the client gets a more updated image, it can also increase the throughput of the clients on the same node when more servers are employed, and thus the whole distributed file will benefit from this. The individual servers then become less loaded

[2]Actually in our case only the level of the LH*LH file is sent.

and forwardings can be avoided. A possibly more serious drawback is that the client has a better image of the file *before* its previous request has reached its final destination[3]. However, the same thing can occur, even if the IAM is sent only when the request has reached its final destination[4]. To avoid this problem, the servers could be required to always send back an acknowledge message when the operation has been totally completed, and then requiring the clients to wait for it. This is, however, an inefficient solution that limits the throughput of the clients, and still the same problem can occur between different clients performing joint work.

In LH* it is proposed to send the IAM *piggy-backed* on the reply message, which is not done in LH*LH where we do not use reply messages when there is no data to be returned[5]. The IAM message is sent using asynchronous communication primitives on the Parsytec Machine. This solves some problems, but can also raise others as discussed below. The encoding of an IAM also includes the file identity number. This enables the data to be directed to the appropriate client file image.

Waiting for a message

When using the communication links on the Parsytec, there is no way when receiving or waiting for messages to choose among messages or probe messages before they have been received. The link is a communication channel that works like a pipe in that messages can be sent from one originator only to the other end, the destination. A client accessing data on an SDDS will by neccessessity have to expect answers from *any*[6] sender. The result is that it has to listen to all links and receive all messages to actually get the message it is waiting for. To avoid this, and to be able to code a message so that it can be selected, one has

[3]For example, a client inserts some data. This yields an IAM that is sent back to the client. The client sends a request for the same data, but to a more correct address, we are now not guaranteed that the data has been inserted at this address. It could have been delayed, and still be waiting to be forwarded.

[4]The actual scenario is similar to the previous one, but involves another request. This request is sent before the search, and triggers an IAM. If the previous insert still has not reached its destination, the search may produce an unexpected result.

[5]Remember that the link communication on the Parsytec is safe.

[6]At least from any server.

either to explicitly code a two-way communication protocol that negotiates by sending extra information or set up a communication link especially for this type of messages. The latter was our choice of implementation. Each link requires a certain amount of buffer space and will thus be associated with some cost. The total number of links to be established will then be linear with respect to the number of different message types and quadratic to the number of nodes. One alternative on the Parsytec is to use the mailbox communication primitives. They are, however, not guaranteed to be delivered, limited in size and slow. But they are an alternative when the number of messages is relatively small, their sizes limited, and when the loss of one individual message will not cause any real harm. This is the case for the IAMs.

9.2.3 Suggested Improvements

When a file is opened, the handle returned should be a handle to the already existing structure, if the file has already been opened. This is, however, just a matter of programming style since the variables containing the handles can be shared or copied.

A key is currently limited to be a null-terminated character string. The hash function calculating the pseudo-key is fixed. It should be definable per file, the data type of the key and what hashing function to use.

For efficiency one would like to send the already calculated pseudo-key in any request having a key. But we choose not to do so, hoping to keep the interface more general. This makes the interface independent of the actual data distribution algorithm. A counter-example is an RP* file that has no "pseudo-key".

Data flow control is hardware/software dependent. On the Parsytec this is enforced by the synchronous link messaging primitives, which probably has a two-way negotiation communication phase, since the message is not received before the receiver asks for it. Generally, in environments such as TCP/IP (sockets or using PVM) we would have to do this ourselves.

The information in the client image that is sent in a request can, as noted, be reduced in some cases. In the LH* algorithm only the level is transferred by the client, and in an image adjust message (IAM) the logical number of the file server is also included. From this information the state, as far as the server knows it[7], can be reconstructed by the client.

An alternative to using the asynchronous communication on the Parsytec machines is to use the control messages link. This should be relatively safe, since the number of IAMs will be limited. This, however, requires the receiver code to know of where to store the information from the IAMs and that it has to keep track of all file handles and their associated data.

9.3 The Server

The data servers store the distributed files. A subset of the data servers cooperate in storing one file. That is, there is one subset of nodes for each file and a specific order of them is called a server mapping that maps from local logical server numbers to physical numbers.

9.3.1 Function Overview

The data server works on messages initiated by a data client. Messages in our Parsytec implementation can be received on either of two links. When a message has been received, an event handler extracts the type of the message. This type that is represented as a number is then used for looking up in a table what function is to be called. The given function is then called. Let us for example assume that we received a `Find` typed message. Then the receiver message-handler function is called with the message buffer as its argument. From this buffer it extracts the key, the data and its associated length and the client image. First, the handle of the bucket stored at this server is retrieved. If such a handle is not found, there is an error in the client addressing mechanism[8]. Then a check is

[7] The server in LH* does not know the actual state of the file, it can only assume that logical servers with a smaller number are of at least the same LH* level as itself, and servers with a higher number can only be assumed to be of a level no higher than itself, presplitting not taken into account.

[8] It might also occur if a distributed file has shrunk and thus the server no longer stores a part of that file. In this case, forwarding information should have been left behind at the node to be used to direct the clients and send them a "special", sort

made as to whether or not the key hashes to this or another server node. This checks the client's image and if it is not the correct server, it will forward the message and send an IAM to the client that requested the operation; then this request is no longer of concern to the server that received it. Eventually, the message will reach the appropriate data server node. When it has been shown that the request concerns the data server that received the request, processing continues there.

LH*LH Workings

Before data normally would be inserted in LH*LH, the server first has to check that it is not currently splitting the file that the request is accessing. If that file is being split, the request might have to be sent, forwarded, to the new server. If so it must be done *without* updating the client, since the *official image* of the file does not yet include the splitting destination. In a more general perspective this goes well in hand with the discussion of the *presplitting of servers* that is presented in [LNS93]. If the file undergoes a split, the key is then checked for whether it would need access to that part of the file that has already been moved. If this is the case, the request is forwarded without adjusting the client's image; nor should the receiving server send an IAM to the clients image[9]. Otherwise, if the data concerns LH-buckets that have not yet been sent, then this data can then be inserted locally into the storage structure.

The tricky part here involves requests that concern data in the LH-bucket currently being moved. If key uniqueness applies, i.e., only one object can be stored for each key, some operations must be handled with care. If an object can be found locally using the key, we overwrite as usual, otherwise we insert it. When this object is shipped to the new server, the same action will take place[10]. Look-ups, however, must first

of reverse, IAM.

[9]It will automatically be updated on the next addressing fault, when the official image of the file has changed.

[10]Actually, data identified by a key could be overwritten several times during the splitting. Either several clients send their request close in time, or some client sends a stream of updates. Then the old data will be overwritten first locally, then when it has been moved, the second insert will not overwrite any data, but it will overwrite later when it is moved. Which of these overwrite inserts that in the end will survive

look in the local data; if a matching key is found we return it, otherwise the request is forwarded to the new server. Deletions are special. We have to delete locally and then forward the request, since we do not have any knowledge about the existence of the data at the new server. If several objects can be stored with the same keys, i.e., it is used as a secondary index, inserts can be made locally; they will be moved later in any case. Look-ups, on the other side have to perform look-up first locally and, independently of the result, forward the request to the new server and then execute it there. Here another complication arises – both the splitting server *and* the new server might have matching data. Then both have to return the answers to the requesting client, which puts higher demands on the implementation on the client which has to be able to receive partial answers from several servers! A more simplistic alternative is to delay the look-up operation until the whole LH-bucket has been transferred, and thereafter forward the request[11]. Deletions will also have to be executed at both places. When data is shipped in the non-unique key variant, it may not overwrite any of the already existing data that matches the key.

It is very important that when forwarding to the new server occurs, no IAM is sent and that when the reply returns to the client, the client does not update its image, even if the client could deduce from the responding server's number that there is (at least) one more server that it does not know about. Our clients only update themselves using information from IAMs. The new server is not allowed to handle any requests from any client regarding the file that is being expanded onto that server. This would bypass the LH*LH concurrency algorithm.

Another important aspect during splitting is the order of the operations and forwardings that we perform. We have to ensure that there is a linearity in time between the forwarding and the shipping of data. This means that we have to assume either that messages are handled at the new server in the order that they arrive and that this is the same order as they were sent from the splitting server[12], *or* we do this linearization

depends on the order that they arrive in at the splitting server.

[11]All requests should be queued then and forwarded in the correct order. The success of this method depends heavily on the time it takes to transfer one LH-bucket.

[12]This means that there has to be full control over or full guarantees on the

ourselves. We take the latter approach. There is a special thread that does the splitting; this thread processes any waiting requests to the data server in-between the shipping operations. Thus messages will be serialized, and sent over the same links. This will guarantee that the order of the messages we send will be preserved at the new server. This solution removes any concurrency problems, but relies on a local queue of requests to be built from requests instead of executing them directly.

9.3.2 Suggested Improvements

When a server receives a request, a check is made as to whether or not it has reached its destination, according to the server itself. If so the LH* algorithm normally does not assume this to be an addressing problem and therefore does not send any IAM. However, from the information given by client image level, it might be deduced that the client needs updating, and the server could then issue an IAM even if there was no addressing error. This, however, is not part of the original LH* algorithm but fits quite well with the LH*LH algorithm and its updating and forwarding mechanism.

When a request is answered by a new server, that is not yet *officially* part of the distributed file, no IAMs are allowed to be sent that could inform the client to know about this new server. One way to implement this, is by masquerading the answer, which means making the server fake its node number to be the same as the server that is splitting. Another solution is to ensure that the clients do not use this information. In either case, no client should be able to send requests to the new server before the shipping is finished at the splitting server. The third, and probably the most secure implementation, would let the server that is splitting act as a proxy for the client and send the request and return the data to the client. This would involve one step of unnecessary communication.

order that messages are assembled, buffered, stored, sent, transferred, received, and stored.

9.4 Server Mapping

The *Server List* contains the means for mapping the logical server node number of the SDDS, in this case that of the LH*LH-file, to a physical machine number. The LH* algorithm does not concern itself with it. Some ideas are, however, presented in [LNS96]. In LH*LH we currently use a static table, known by all nodes in the multi computer. But the management and distribution of this table itself is a problem potentially of high proportions. Each client accessing the SDDS and each server storing the SDDS has to be able to do this mapping. One easy solution is to state that this mapping table will not grow too large and can therefore be stored at each participating server. When a client makes an addressing error, the missing information could be added to the IAM (Image Adjust Message) and then update the client. This solves the problem of just having a central node storing and delivering the map on request, but it would be a non-scalable solution since the actual allocation of nodes does not scale. This particularly concerns cases when the clients are many and possibly short-lived.

Changes in the mapping will either have to be distributed to the affected clients and server, or using an SDDS-like schema be updated by the (old) server receiving the faulty request. One idea, untested though, is to add another image variable that somehow is a "unique" id of the mapping it has (a simple solution would be a checksum of the mapping data), and if a server detects that the "image" is out of date, it then adjusts it with an IAM, adding the new mapping. If the client accesses a server that no longer has any knowledge of the accessed SDDS file, it can then use a fallback scheme that shrinks the image level by level, till only, in the end, the logical server 0 is contacted. If this fails the client has to "Reopen" the file [13]. Reopening the file might have to use a directory service for finding the SDDS file from its ID.

9.4.1 Autonomous "randomized" mapping

Another idea is to not have the server list fully materialized, but instead use a scheme where the mapping is calculated in a stable and

[13] Accessing a SDDS file requires knowledge at least of the first server's physical address.

reproducible way. This can be done both at the clients and server totally independently. If there are a limited number of machines that can be used and a large amount of files to be stored, a randomizing schema could be used for allocation of new server nodes to the file. This might also provide benefits of randomized load balancing, assuming random creation and growth of the stored files.

Thus, the server list can be seen as a data structure with operations that can reproduce the actual mapping. This is especially needed in a large distributed autonomous environment with many nodes. An example of such an algorithm would be a randomizing function with a seed specific for the distributed file (the id, for example), and with a list of servers participating that are willing to participate in the distribution. It is known that most randomizing functions are not really random. With the same seed the same sequence of numbers can be reproduced; these numbers (usually between 0 and 1) can then be used to remove identities of willing (unused) nodes from the list of all nodes. All clients (and servers) using the same function, seed and so on. will then yield the same sequence. The possibly unwanted property of this method is that this list of "willing" servers has to be maintained or allocated. However, the algorithm does not require a total static list, it can be allowed to grow in chunks and communication can then be minimized.

9.4.2 DNS alike mapping (internet)

Another, more appealing method for internet distribution can be to use the internet DNS databases that provide mapping from a logical name to a physical ip-address. For example, we could register a domain *lhlh.ida.liu.se* under which we let each LH*LH-file be identified by, for example, a unique name, let us say *film*. We then let the first node (logical node numbered zero) in this LH*LH-file be *0.film.lhlh.ida.liu.se* and the second *1.film.lhlh.ida.liu.se* and so on. This provides physical independence, it also scales since the DNS servers employ caching. However, the drawback with this schema is that when the nodes then move, it may take a rather long time before this information reaches out through the distributed database[14].

[14] It may take from a few hours to some days depending on the software used.

Chapter 10

Summary and Future Work

10.1 Summary

In this thesis, we started by describing databases and future application that will use database technology. We identified the need of high-performance, scalability and high availabilty, and this led up to parallel data servers. Dynamic (re-)scalability is achieved by using the new type of data structures, the Scalable Distributed Data Structures (SDDSs). We have described a real implementation of an SDDS, the LH*LH. The behaviour of LH*LH was studied and performance measures were given. It was shown that the local bucket management is an important issue for implementation of high-performance SDDSs. Several thoughts around the implementation and details of problems involved were discussed. Also, potential areas of problems have been identified.

10.2 Future Work

Future work should concern a deeper performance analysis of LH*LH under various conditions. The use of SDDSs in a transactional environment with several interfering autonomous clients working is a particular interesting area. A formal performance model is also needed. In general such models are not yet available for the SDDSs. The task seems of even greater complexity than for more traditional data structures. Also, if the algorithm is to be used for more than one file, a different physical mapping (e.g. randomization) to the nodes should be used for each file to distribute the load. Several different solutions could apply here, but this is an open area. The handling of the list of participating servers for a SDDS-file also needs to be scalable, and this needs further investigation. The ideas put into the LH*LH design should also apply to other known SDDSs. They should allow for the corresponding variants for switched multi-computers. One benefit would be scalable high-performance ordered files. The SDDSs in [LNS94], or [KW94] would seem to be a promising basis on which to aim for this goal.

A particularly promising direction would be the integration of LH*LH as a component of a DBMS. One may expect important performance gains, opening to DBMSs new application perspectives. Video servers seem to be a favorable axis, as it is well known that major DBMS manufacturers are already investigating switched multi-computers for this purpose. The complex real-time switching data management in telephone networks seems to be another domain of interest.

Having a rich set of modeling capabilities in newer database systems, such as object-orientation, gives rise to other questions. Are objects and their attributes to be stored differently depending on their position in the object-type hierarchies? How do we store what objects in an SDDS file? How does this relate to object-oriented querying? User defined predicates? As can be seen there are many interesting issues that will arise.

10.2.1 Host for Scientific Data

To approach these goals, we plan to make use of the implementation of LH*LH for high-performance databases. We will interface it with our research platform, AMOS [FRS93], which is an extensible object-relational database management system with a complete query language [KLR+94]. AMOS would then reside on an ordinary workstation, whereas some data types, relations or functions would be stored and searched by the MIMD machine. AMOS will then act as a front-end system to the parallel stored data. The query optimization of AMOS will have to be extended to also take into account the communication time and possible speed-up gained by using distributed parallel processing. SDDSs other than LH* are also of interest for evaluation, a

new candidate being the RP* [LNS94] that handles ordered data sets.

Bibliography

[AvdBF+92] Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA /DB: A Parallel Main Memory Relational DBMS. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):541–554, February 1992.

[BAC+90] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[BCV91] B. Bergsten, M. Couprie, and P. Valduriez. Prototyping DBS3, a Shared-Memory Parallel Database System. In *Proceedings of First International Conference on Parallel and Distributed Information Systems*, pages 226–234, Miami Beach, Florida, December 1991.

[BD83] H. Boral and DeWitt. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines. In *International Workshop on Database Machines*, volume 3, pages 166–187, Munich, 1983.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[CGK+90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–89, March 1990.

[Cor88] Teradata Corporation. DBC/1012 data base computer concepts and facilities. Technical Report Teradata Document C02-001-05, Teradata Corporation, 1988.

[CRDHW74] R. H. Canady, J. L. Rydery R. D. Harrisson, E. L. Ivie, and L. A. Wehr. A back-end computer for data base management. *Communications of ACM*, 17(10):572–582, October 1974.

[Dev93] R. Devine. Design and implementation of DDH: A distributed dynamic hashing algorithm. In *Procedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO)*, 1993.

[DG92] David DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the acme*, 35(6):85–98, 1992.

[DGG+86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A high performance dataflow database machine. In *Procedings of VLDB*, August 1986.

[Dou90] B. Dougherty. Telco's Strategic Importance in Tandem's Success. *Industry Viewpoint*, 1990.

[Du84] H. C. Du. Distributing a database for parallel processing is np-hard. *ACM SIGMOD Rec.*, 14(1):55–60, March 1984.

[FJP90] J. C. French, A. K. Jones, and J. L. Pfaltz. Summary of the Final Report of the NSF Workshop on Scientific Database Management. In *SIGMOD Record*, volume 19:4, pages 32–40, December 1990.

[FRS93] G. Fahl, T. Risch, and M. Sköld. AMOS - An Architecture for Active Mediators. In *IEEE Transactions on Knowledge and Data Engineering*, Haifa, Israel, June 1993.

[IEE92] IEEE. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE, 1992. http://www.SCIzzL.com/.

[KLR+94] J. S. Karlsson, S. Larsson, T. Risch, M. Sköld, and M. Werner. *AMOS User's Guide*. CAE-LAB, IDA, IDA, Deptartment of Computer Science and Information Science, Linköping University, Sweden, memo 94-01 edition, March 1994. URL: http://www.ida.liu.se/labs/edslab/amos/amosdoc.html.

[KLR96] Jonas S Karlsson, Witold Litwin, and Tore Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Advances in Database Technology — EDBT'96*, pages 573–591, Avignon, France, March 1996. Springer.

[KTMO84] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Architecture and performance of relational algebra machine GRACE. In *Procedings of the Intl. Conference on Parallel Processing*, Chicago, 1984.

[KW94] B. Kroll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In *ACM-SIGMOD International Conference On Management of Data*, 1994.

[Lar78] P.Å. Larson. Dynamic hashing. *BIT*, 18(2):184–201, 1978.

[Lar88] P.Å. Larson. Dynamic hash tables. In *Communications of the ACM*, volume 31(4), pages 446–57. April 1988.

[LER92] Ted G. Lewis and Hesham El-Rewini. *Introduction to Parallel Computing*. Number ISBN 0-13-498916-3. Prentice Hall, 1992.

[Lit80] W. Litwin. Linear Hashing: A new tool for file and table addressing. Montreal, Canada, 1980. Procedings of VLDB.

[Lit94] W. Litwin. Linear Hashing: A new tool for file and table addressing. In Michael Stonebraker, editor, *Readings in DATABASE SYSTEMS, 2nd edition*, pages 96–107. 1994.

[LNS93] W. Litwin, M-A Neimat, and D. Schneider. LH*: Linear hashing for distributed files. ACM SIGMOD International Conference on Management of Data, May 1993.

[LNS94] W. Litwin, M-A Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. VLDB Conference, 1994.

[LNS96] W. Litwin, M-A. Neimat, and D. Schneider. LH*: A Scalable Distributed Data Structure. *ACM-TODS Transactions on Database Systems*, Dec. 1996.

[LR85] M. D. P. Leland and W. D. Roome. The silicon database machine. In *Procedings 4th International Workshop on Database Machines*, pages 169–189, Grand Bahama Island, March 1985.

[MSP93] A. Matrone, P. Schiano, and V. Puotti. LINDA and PVM: A comparsion between two environments for parallel programming. *Parallel Computing*, 19:949–957, 1993.

[Näs93] Joakim Näs. Randomized optimization of object-oriented queries in a main memory database management system. Master's thesis, Department of Computer Science and Information Science, Linköping University, 1993.

[Ors96] Kjell Orsborn. *On Extensible And Object-Relational Database Technology for Finite Element Analysis Applications.* Dissertation no. 452, Department of Computer Science and Information Science, Linköping University, 1996.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems.* Number ISBN 0-13-715681-2. Prentice Hall, 1991.

[Par94] Parsytec Computer GmbH. *Programmers Guide, Parix 1.2-PowerPC*, 1994.

[Pet93] M. Pettersson. Main-Memory Linear Hashing - Some Enhancements of Larson's Algorithm. Technical Report LiTH-IDA-R-93-04, ISSN-0281-4250, Department of Computer Science and Information Science, Linköping University, 1993.

[PGK88] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, USA, June 1988. SIGMOD.

[SAP+96] M. Stonebraker, P. M. Aoki, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. MARIPOSA: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, January 1996. http://epoch.CS.Berkeley.EDU:8000/mariposa/papers/s2k-95-63.ps.

[Skö94] Martin Sköld. *Active Rules based on Object Relational Queries – Efficient Cange Monitoring.* Licentiate thesis no. 452, Department of Computer Science and Information Science, Linköping University, 1994.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The Design of XPRS. In *VLDB Conference*, volume 14, pages 318–330, Los Angeles, California, 1988.

[SM96] Michael Stonebraker and Dorothy Moore. *Object-Relational DBMSs: The Next Great Wave.* Number ISBN 1-55860-397-2. Morgan Kaufmann Publishers, INC., San Francisco, California, 1996.

[SPW90] C. Severance, S. Pramanik, and P. Wolberg. Distributed linear hashing and parallel projection in main memory databases. In *Proceedings of the 16th International Conference on VLDB*, Brisbane, Australia, 1990.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated Volume 1*. Addison-Wesley, 1994.

[Tan87] Tandem. Nonstop sql - a distributed high-performance, high-availability implementation of sql. In *Procedings International Workshop on High Performance Transaction Systems*, pages 337–341, Asilomar, Calif., September 1987.

[Tor95] Øystein Torbjørnsen. *Multi-Site Declustering Strategies for Very High Database Service Availability*. Phd-thesis 1995:16, Department of Computer Systems and Telematics, Faculty of Electrical Engineering and Computer Science, Norwegian Institute of Technology, University of Trondheim, Norway, 1995.

[WBW94] R. Wingralek, Y. Breitbart, and G. Weikum. Distributed file organisation with scalable cost/performance. In *Proc of ACM-SIGMOD*, May 1994.

[Wer94] Magnus Werner. A Client-Server Interface for AMOS. Caelab memo, IDA, Department of Computer Science and Information Science, Linköping University, 1994.

[Wer96] Magnus Werner. *Multidatabase Integeration using Polymorphic Queries and Views*. Licentitate thesis no 546, Department of Computer Science and Information Science, Linköping University, 1996.