OS Support for VLDBs: Unix® Enhancements for the Teradata® Database

John Catozzi

Sorana Rabinovici

AT&T Global Information Solutions
El Segundo, CA 90245 USA
john.catozzi@SanDiegoCA.attgis.com
sorana.rabinovici@SanDiegoCA.attgis.com

Abstract

This paper presents the parallel enhancements which allowed the port of the Teradata Database from TOS, a proprietary 16-bit Operating System, to an SVR4 Unix system. It gives an architectural overview of how the Teradata Database solves the main VLDB problems: performance and reliability. We will present the transition from the Database Computer DBC/1012 nodes (Interface Processors-IFPs and Access Module Processors - AMPs) to the virtual processors (vprocs), which run concurrently in a collection of SMP nodes. We also present the Parallel Database Environment (PDE) add-on package to Unix that makes this possible. We will discuss the results of our performance enhancement work and the directions for the future.

Introduction

Twelve years ago, the Teradata DataBase Computer or DBC became the world's first massively parallel computer for database processing. That computer system, operating on a large collection of uni-processor Intel x86 nodes with a proprietary operating system (TOS), pioneered the MPP relational database market. Today there are over 400 sites with these systems. In this paper we describe this original system in overview, then present the work which was involved in re-implementing the Teradata database software on today's modern generation of large-scale massively parallel systems where each node of the system is a powerful Symmetric Multi-Processor (SMP) computer running the Unix operating system. This work was very challenging and presented some unique problems. We present here some overall architectural concepts of this work along with a couple of implementation highlights. We then show the importance of the OS interface to the performance of the database with the results of our efforts to maximize the performance of the Teradata database software in this environment.

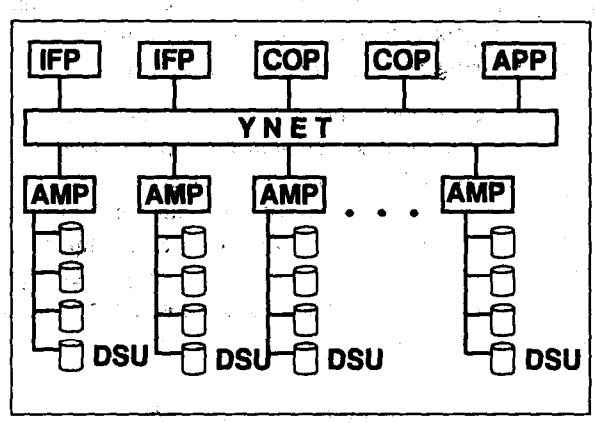

Figure 1: The Original DBC

The Teradata DBC/1012

The DBC is a database management system. Its main elements are Interface Processor nodes (IFPs) and Access Module Processor nodes (AMPs) connected by a proprietary network (the YNET). Each of these elements is an x86 uni-processor with 8 to 16 megabytes of memory running a proprietary operating system (TOS) in 16 bit mode. The IFP node provides the connection to clients through an IBM channel or ethernet interface. An AMP has up to 10 gigabytes of attached disk. TOS together with the Teradata Database software fully parallelizes all functions among these simple nodes. A DBC basic configuration is presented in Figure 1.

In a DBC, data is represented as a collection of tables,

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

relational in nature, spread over all the configured disks, according to a hash algorithm.

The IFPs receive user requests (queries) and translate them in internal request steps that are forwarded over the YNET to the AMPs. They also coordinate the responses that come from the AMPs and present the results to the users. The AMPs receive requests from the IFPs, perform the required data manipulation and send appropriate responses back over the YNET. Since the database tables are evenly distributed across the disks of all the AMPs, the workload is balanced. When an IFP receives a request the parser task interprets it using the data dictionay which contains information about all the databases/tables in the system. It resolves symbolic names and makes integrity checks. The request is then split into a series of data manipulation steps whose execution is monitored by a dispatcher. The dispatcher sends the steps on the YNET toward their destination. As mentioned before, the rows of a table are evenly distributed amongst all AMPs in a system, so that all AMPs could work at the same time on the data of a given table. If a request is for data in a single row (i.e. "prime key" request), the IFP will transmit it toward the AMP on which the data resides. If the request is for multiple rows, the steps will be forwarded to all participating AMPs.

Teradata on Unix/PDE

A multitude of factors drove us to the development of a new system to replace TOS and to support the Teradata database software into the future. Among them were the emergence of 32-bit mode, supporting larger physical and virtual memory sizes, open systems trends, and the availability of the SMP node in which multiple CPUs, sharing memory, concur in solving the workload presented to the node. The decision was made to build upon AT&T's Fault resilient MP-RAS SVR4 Unix as the base operating system for the new version of Teradata. To this basic operating environment we added extensions to support a parallel environment, to provide a single system view, to support a parallel debugger, and to make it easier to port the Teradata database software from TOS. We call these extensions Parallel Database Environment or PDE. The general relationship of this software in the system is shown in Figure 2.

We will introduce PDE by describing two of its fundamental concepts - cliques and virtual processors. The system is made up of large, powerful SMP nodes arranged into cliques and each running multiple virtual processors. These nodes are connected by a scalable inter-connect called the Bynet.

A collection of processor nodes connected to shared external data storage is termed a clique. Cliques are the fundamental physical building blocks of this system.

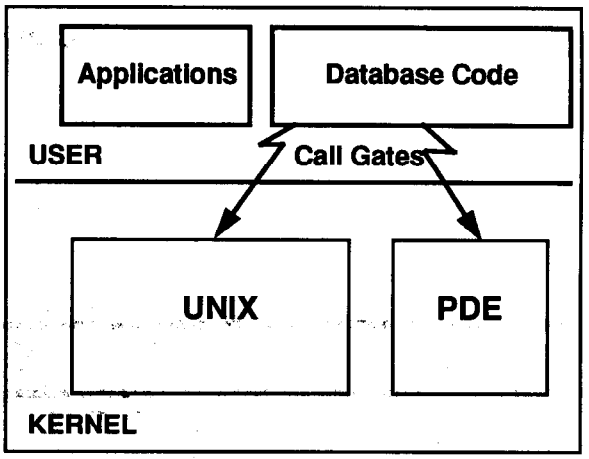

Figure 2: New System Software

Under normal operation, each node is assigned exclusive use of a subset of the shared disks. In case of node failure, the disks would be assigned to the remaining nodes in the clique, thus maintaining full availability of all the data. A general diagram of the system is shown in Figure 4.

Given the power of the SMP node, it is necessary to provide for intra-node parallelism as well as the inter-node parallelism of the DBC. We accomplish this by running multiple instances of AMPs and/or IFPs in a node. Each such instance constitutes a virtual processor (vproc).

The virtual processor concept adds a level of abstraction between the multi-threading of a work unit, and the physical layout of the computing system allowing us to host a shared nothing database on an MPP platform made up of shared memory SMP nodes. This concept results in better control over the degree of parallelism and provides for higher system availability without undue programming overhead in the application. Each virtual processor is given its own private logical disk space, called a virtual disk (vdisk). The vdisk may actually be a conglomeration of several physical disk drive units. Vdisks can be accessed by any processor node within a clique allowing a vproc to be started and run anywhere within a clique.

The Teradata database code commits all transactions to disk. In order to improve performance and at the same time provide for the possibility of a node failure we use the high bandwidth interconnect to keep the modified segments that belong to one node in a backup node within the same clique.

PDE provides the ability to run multiple virtual processors on a processor node. Each virtual processor is isolated from the others on the same node. There is no shared context between vprocs on the same node. This enables the concept of location transparency where the application is unaware of the physical location of a vproc. This becomes an important consideration allowing the system to operate in the presence of a failed node.

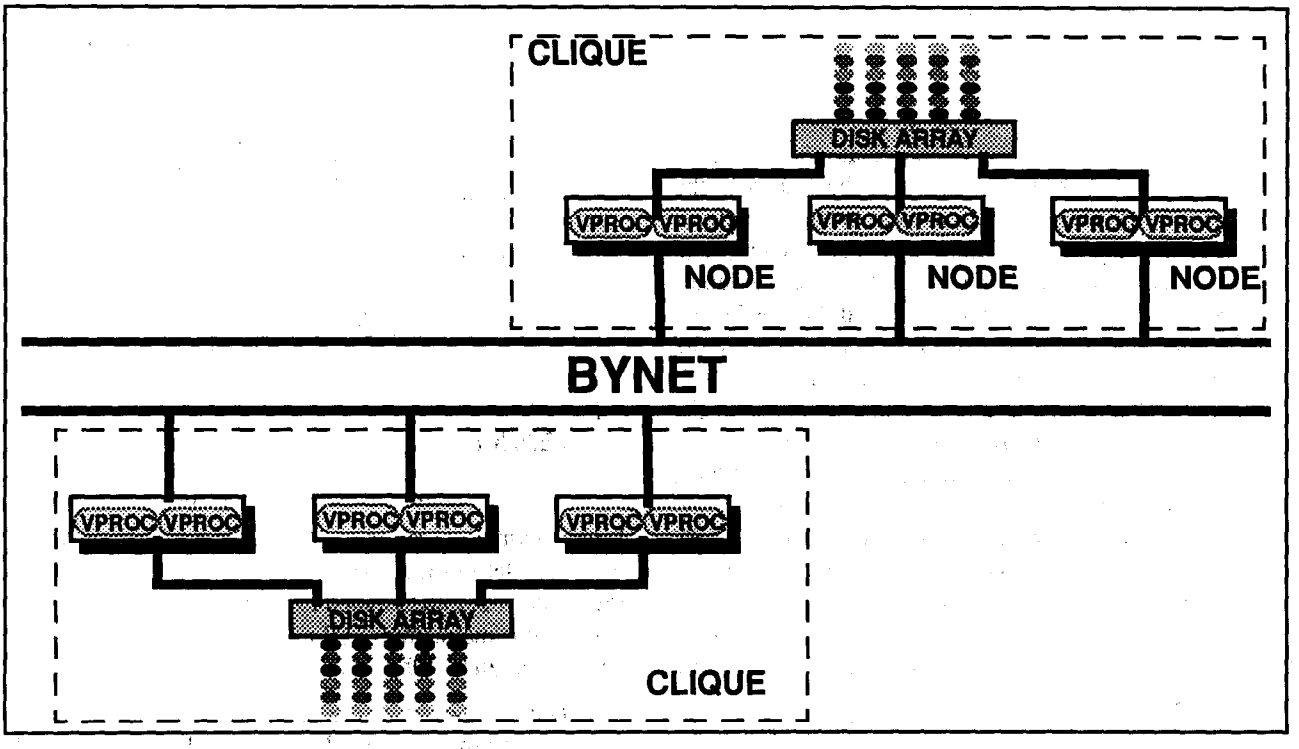

Figure 4: New System Configuration

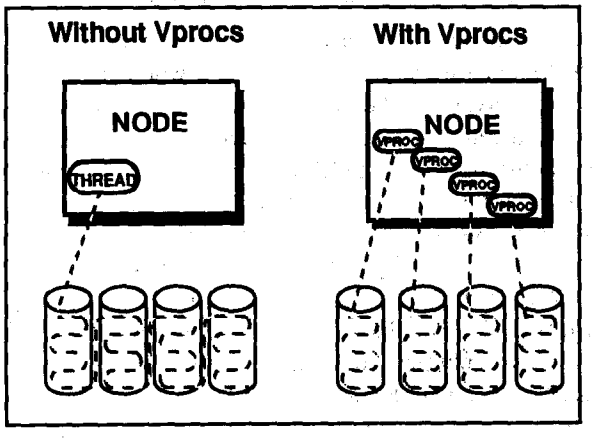

Figure 3: Increased Parallelism Provided By VPROCs

The introduction of cliques and vprocs brings additional parallelism and higher availability to the system. Two factors contribute to the additional parallelism:

1. Disk Utilization: If a processor node has multiple storage devices (disk drives) attached, a single thread of execution might occupy only one of those devices at a time, leaving the others under-utilized or even idle. With vprocs, the degree of parallelism is increased to include up to one thread per disk, rather than one thread per node.
2. CPU Utilization: Having multiple vprocs per node allows increased parallelism for a single query enabling it to utilize all of the available CPUs.

This is illustrated in Figure 3.

To accomplish better availability we allow for vproc migration. In the case of a node failure, all the vprocs assigned to the failed node are restarted on the remaining nodes of the clique. The resultant configuration, shown in

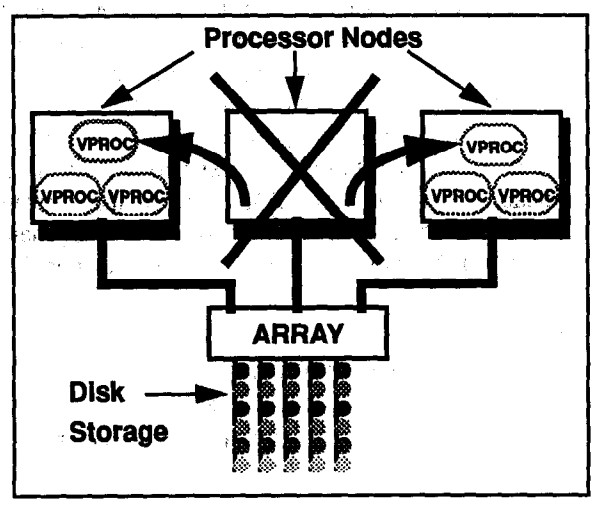

Figure 5: Clique Recovery - VPROC Migration

Figure 5, operates with less processing power, but the par-

allel application still has the same number of vprocs and retains full access to all data storage.

Our implementation of virtual processors is at the operating system data structure level, and in its interconnect software. By providing the virtual environment at the data structure level, the partitioning and isolation of vprocs can be provided at a much lower performance cost than traditional virtual machine implementations.

The Unix/PDE system keeps track of each task/process and the virtual processor to which it belongs. Whenever a task is scheduled its vproc is active, so that allocation of per vproc resources is unique to the vproc. This minimizes the interference from the other tasks running in the system. Allocated names, for example, are unique within each vproc. Unix/PDE assigns vprocs to processor nodes at system start-up time based on a load-balancing algorithm. At that time a translation table of vproc <--> node is created and stored. When a message is sent to a mailbox at a particular vproc the interconnect software looks up the vproc --> node translation in the table, sends the message to the designated node, and the recipient then routes it locally to the appropriate mailbox in the proper vproc.

Implementation

Unix/PDE provides to the database code the same type of services that TOS provides. Each Vproc runs approximately 100 tasks, which frequently map/unmap database segments. The segments of the database are sometimes heavily shared. There is also a class of objects called Global Distributed Objects toward which all the processes on a node have visibility, and which are kept in sync across the system. This functionality is implemented as an add-on at the kernel level because of performance, security, better control, and debuggability.

The main SVR4 features around which we designed PDE are SVR4's object oriented approaches to process scheduling and virtual memory management. We introduced a new scheduling class (Unix usually provides Time Sharing, Real Time, etc.) for the tasks working for the database. This allowed us to get control on events like sleep, setrun, preempt, wakeup, etc. This is extremely important to the implementation of the parallel debugger. We also introduced a virtual segment driver, which allowed us to manage the database blocks. A virtual segment driver allows the definition of the action to be taken when noticeable events occur, for a given virtual range (e.g faults, change of protection, acquiring and dropping of maps, etc.) Using this facility we were able to implement all the mapping and interlocking mechanisms required by the database code, without paying a high overhead price.

There are many innovations included in PDE, but in this paper we will describe just two of them: the shared map segments and the node flush synchronization through shared disk. The shared map segments are important for the handling of the database shared segments and for the GDOs. The node flush mechanism is a critical component for implementing cliques.

It is a well known fact, that in Unix, sharing data or text between a large number of processes is both memory and time consuming. In order to alleviate this problem, we introduced a new type of segment, the shared map segment. The basic concept is to use a shared page table to improve the management of the shared physical memory.

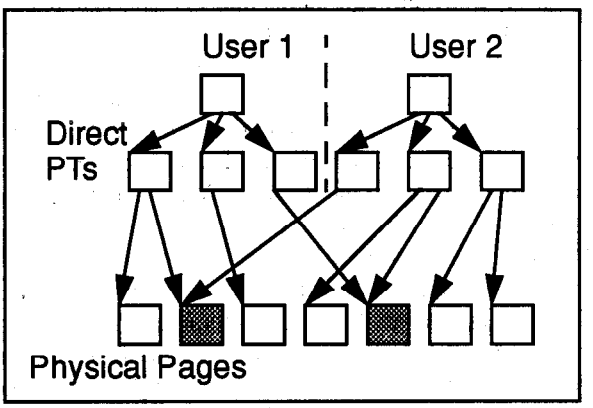

Figure 6: Classic Shared Memory

In a system with hundreds of processes sharing in read only mode significant amounts of data (e.g. shared libraries, and in the PDE case, Global Distributed Objects) the fact that we use a single page table to gain access to the shared data, saves hundreds of pages for the system. It is

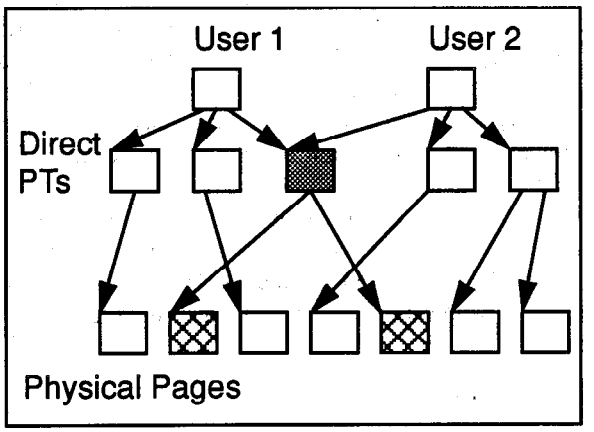

Figure 7: Shared Page Table

also much easier to track the shared data, and eventually to change it, since there is only one mapping to change and not hundreds. In a typical configuration, we have an 8-way SMP node, with 10 to 14 vprocs and consequently around 1400 processes. In this case, using a shared map segment, reduces the amount of memory needed for page tables by 5.6 MB for each 4MB of shared memory. This can add up to a savings of over 100-200MB during typical operation. Another advantage of this schema is that the chance of hitting the page table in the cache and not having to go to

memory for it is much improved. The overhead of the hardware layer of the operating system, which walks all the mappings of a physical page, checking the referenced and modified bits, in order to maintain consistency, is also very much reduced. This solution allowed us to implement the functionality needed for the Global Distributed Objects, with very small overhead, and with substantial savings of memory.

Another innovative technique was used for synchronizing the flushing of database segments by the primary or the backup node, through shared disk tokens. As explained before, the database uses a node in the same clique (backup node) to keep a copy of the modified segments of a primary node. This allows us to use low latency interconnect I/O in order to implement a write cache, saving the performance penalty of the much slower disk I/O when committing updates to the database. When an exceptional condition occurs such as a node crash, power failure, etc. PDE has to ensure data integrity, and for this reason either the primary or the backup, but not both, have to flush their modified segments to disk. It is preferable to flush the primary, because it has the most recent data. To make sure that only one node flushes, without relying on the Bynet which may be in an error condition, the PDE uses shared disk tokens to implement a persistent storage semaphore and a set of flags to control this operation. We have defined a synchronization segment on a shared disk, and rely on the fact that we can write a primary or backup token in two different sectors of this segment and then atomically read them both. The tokens have values like: FLUSHING_INIT, FLUSHING_WANTED, FLUSHING_INPROGRESS, FLUSHING_DONE. Each node will write its token, starting with FLUSHING_WANTED, atomically read both tokens, and depending on the token combination, flush or watch the other node flushing. This guarantees that as long as one of the nodes is functioning a valid state will be created on the disk. Following this flush, when the system restarts, the areas of disk which were being handled by the failed node can be reassigned to any other node with shared access to the disk storage.

Performance work

It was no surprise that initially the new system performed worse than the current TOS based system. A serious effort was put into fixing the performance bottlenecks. In doing this we created different workload profiles to characterize the system behavior. Among them were: concurrent DSS workloads, consisting of a number of concurrently running complex DSS query streams, join workloads with row redistribution, full file scans, etc. Improving the performance of a complex system is like peeling an onion. The removal of each bottleneck unveils the problems lurking below. Over the period of six months we were able to constantly peel away layers of the performance problem onion. This work resulted in *one and a half orders of magnitude* improvement in the performance of the database software under realistic workloads. Most of the performance bottlenecks were related to the inherent differences in the two environments: the uni-processor small memory of the DBC/1012 vs. the 8-way SMP and 2GB of memory of the modern Unix server. This "fatness" of the node requires that careful attention be paid to the areas of flow control and scheduling of work, system overhead, and efficient access to scarce system resources through fine grained locks.

We built tools to observe the hot points in the system (e.g. excessive lock spinners) and to find out the cause of excessive idle time (pinpointing the sleep reasons of "busy" tasks - tasks working on behalf of a given query). Through this work we improved the performance step by step. Sometimes the gain of a particular improvement was small, but some specific steps were big winners.

One of our first observations was that having huge physical memory (up to 2 GB) did not automatically ensure better performance. The cache of the database segments grew proportionally with installed memory, and the techniques to store and retrieve from that cache became inadequate(e.g. the number of segments on one hash bucket grew to ~600). So, our first step was to increase by an order of magnitude the number of hash buckets for the database segments, and to change the locking protocols used to access them in order to provide for finer locking granularity. We also improved the algorithm used to randomize the distribution of these segments on the hash buckets. Combined, these actions reduced considerably the locking contention in the memory management interface.

In order to provide for maximum throughput of the database under intense workload conditions, the Teradata software relies on a work flow control scheme which shuts down the arrival of new work when the resources of a vproc have reached a certain level of congestion. By improving the granularity at which we controlled this work flow we were able to increase the overall performance of the system.

The nature of the database activity causes the processes working on behalf of a query to go through many transitions of the nature: 'running/waiting for I/O/running'. In the case of a multi-processing system it is important to run a task on the same CPU as often as possible in order to maximize the usefulness of information in the CPU's cache. This was enabled by code in the scheduler to track and maintain CPU affinity for each task.

Another major gain came from the elimination of the Unix OS control over the lower level hardware mapping of the database memory objects. The standard UNIX code

goes to great lengths to keep track of the usage of memory pages in order to optimize its demand paging scheme. Since PDE controls the caching of the database segments, these pages are not subject to demand paging and this overhead is unnecessary.

Other performance gains were realized from not zeroing the scratch segments for the database (the database code is aware and initializes only the needed areas), compacting the messages, avoiding as much as possible locking contentions, and improving on the tracing system lock granularity.

Some additional gains came from changes to the database code to operate more efficiently in this new SMP/MPP environment. For example the merge join loop was optimized and the overhead of row redistribution was considerably reduced by batching up to 4KB of rows to a destination.

The result of our performance work is illustrated in the performance progression charts - Figures 8 and 9. These

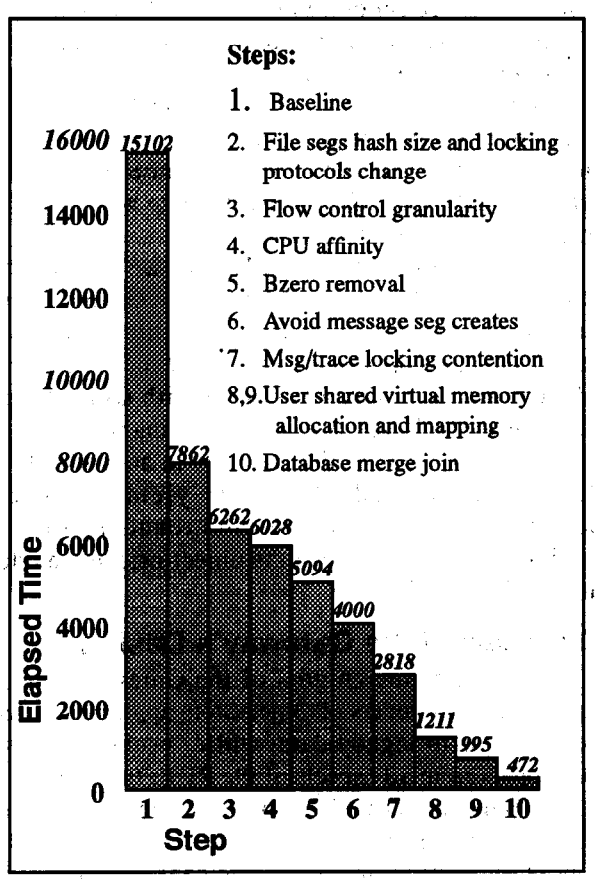

Figure 8: Redistribution Joins Performance

figures show the reduction in elapsed time that was realized as each of the performance enhancements was put into place for two of the standard query workloads.

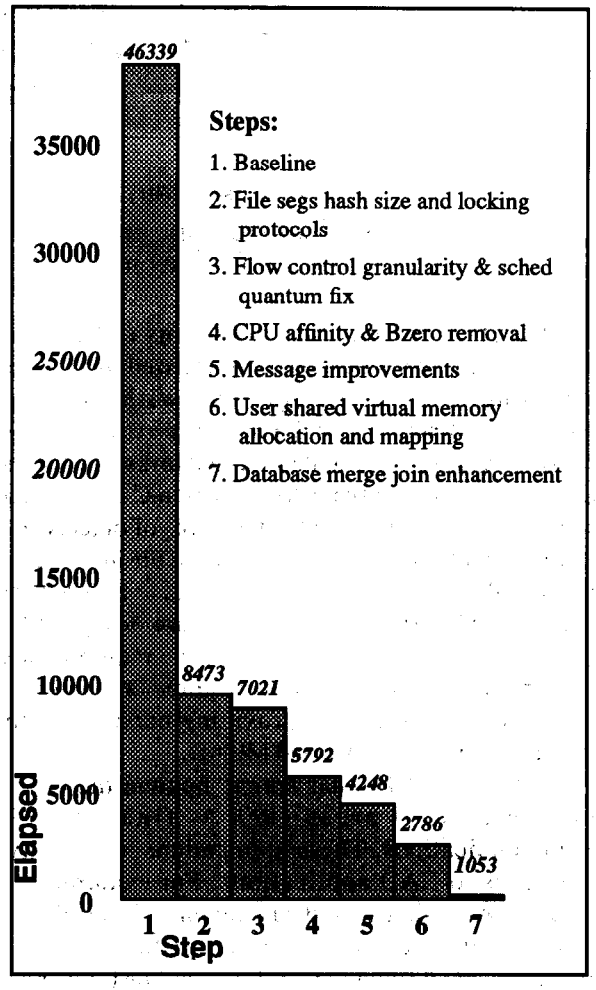

Figure 9: Concurrent DSS Performance

Summary

The scalable, massively parallel, Teradata database has been successfully ported from the DBC/1012 with its uni-processor, limited memory nodes to a new scalable MPP system using large SMP nodes running the Unix SVR4 MP-RAS operating system. By building on the object oriented features of SVR4, viz the scheduling classes and segment classes, we have created an architecture which provides for scalable intra-node parallelism through vprocs and for high availability through the organization of the nodes into cliques.

We have demonstrated a way to support a scalable shared nothing architecture on a large shared memory, multi-processing node that results in high performance by minimizing the system overhead for common functions and ensuring maximum concurrence for the system service software through fine grained locking of scarce resources. The experience that we have gained from this effort will serve us well in the future as we look to other platforms and the Windows NT operating system.