1990

Experiences with Parallelizing a Distributed Database System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Janche Sang

Yin-he Jiant

Report Number:
90-1047

EXPERIENCES WITH PARALLELIZING A
DISTRIBUTED DATABASE SYSTEM

Bharat Bhargava
Janche Sang
Yin-he Jiang

Experiences with Parallelizing a Distributed Database System†

Bharat Bhargava, Janche Sang, Yin-he Jiang

Computer Science Department
Purdue University
West Lafayette, Indiana 47907

Abstract

Raid is a distributed database transaction processing system. This research is an attempt at porting and running the various modular component of the Raid software on parallel processors and conduct a series of measurements that measure the performance of transaction processing. The parallel version of Raid called P-Raid is implemented on Sequent/Symmetry which is a shared memory multiprocessor system. This architecture allows us to increase efficiency by running several Raid servers (such as concurrency controller, atomicity controller) concurrently, thus achieving parallelism at the server function level. To implement parallel version of RAID, several changes in the different layers of the system were required. They deal with communications library and system interface. We found that the multiprogramming level of five, resulted in a thirty percent improvement in the throughput.

1. Introduction

There are two orthogonal directions of research in transaction processing systems. Distribution deals with availability and parallelism deals with performance. This study has the focus on finding the inherent parallelism in distributed database systems and

utilizing it to enhance performance.

Our approach is to parallelize an existing system called RAID[1]. RAID is a distributed database transaction processing system and is designed to be robust and adaptable. It is designed to run under the UNIX system on a network of SUN workstations. In order to gain experience and understanding how a distributed database system can be parallelized, we have migrated RAID system from SUNs to parallel machines . We call the parallel version of the system P-RAID. In this paper, we present our experiences with the changes in the RAID software and make measurements to compare the performance of the distributed and parallel version.

In RAID system, the functions of transaction processing are divided into software modules called servers. The roles of the servers in the RAID system are the User Interface(UI) - the front end invoked by the user to process queries,, the Action Driver(AD) - execute the transaction, the Atomicity Controller(AC) - manages two commit phases of transaction processing to ensure global serializability, the Replication Controller(RC) - maintains consistency of the replicated copies of the database, the Concurrency Controller(CC) - ensures local serializability at a given site, and the Access Manager(AM) - provides write access to the local database. These servers construct a RAID site which has a replicated database. Figure 1 shows the pattern of server communications in RAID. Each box in the figure represents a RAID server. Arrows represents service requests from one server to another(some arrows represent more than one service request.) Unboxed server names represent servers on other sites.

The implementation of servers in RAID used one process for each server and a general-purpose communications protocol for interprocess communication. RAID servers need to communicate both with servers at the same site and with servers at other sites. A high level, layered communication package provides a clean, location transparent addressing between servers. RAID communication facilities are implemented on top of UDP/IP, the Internet Universal Datagram protocol. The communication path of RAID is shown in Figure 2(a).

However, RAID's server design has performance problems which have been discussed in [2]. There are two techniques of reducing this overhead in RAID:

- Merge several servers into the same process rather than use a separate process for each server. Context switching can be reduced because there are fewer processes and communication costs can be reduced because of using shared memory instead of UDP. In [2], we merged the servers on one site into a single process and improved the system performance by up to 20 percent on a 4-site system. The
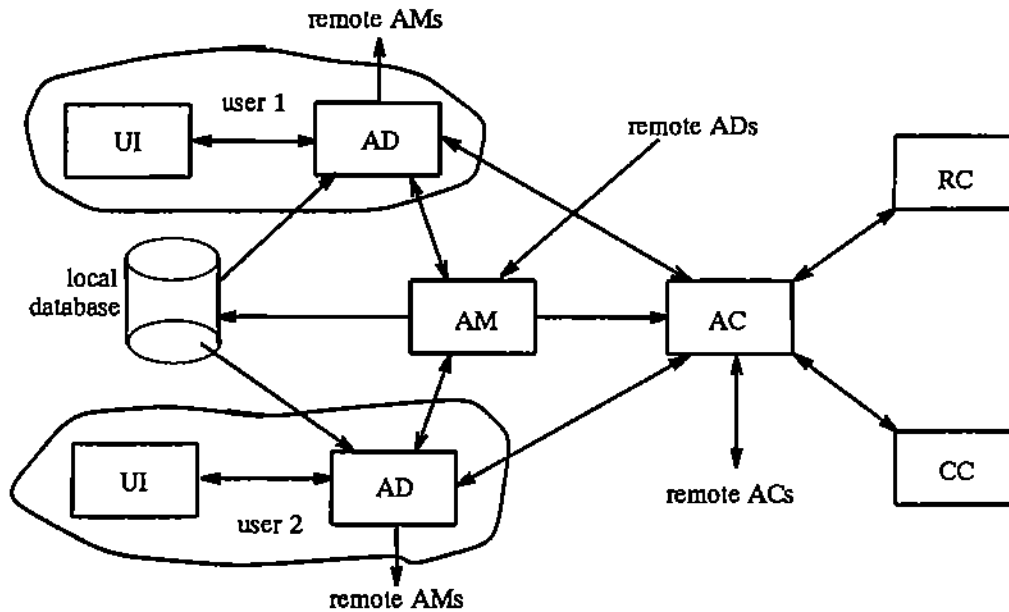
Figure 1: The organization of a RAID site

physical organization of the second version RAID is shown in Figure 2(b). Unfortunately, merging all four servers also forces the servers to run synchronously. The loss of concurrency is the major disadvantage in this scheme.

- Run the servers on different processors concurrently. It is obvious that context switching time can be eliminated entirely. In addition, system throughput could be improved due to parallel processing of transactions.

We use the second approach to build a parallel RAID system.

The parallel machine selection is an important issue. There were two different kinds of MIMD machines available to us. One was Ncube/7 and the other was Sequent/Symmetry. Ncube/7 has a host processor and 128 node processors. Each node processor has its own memory and communicates with each other by using message passing scheme. However, the operating system VERTEX running on the node processor is a subset of UNIX. Porting RAID on the Ncube/7 suffered from the lack of compatibility of system calls. The system calls which were used in Raid but not supported by VERTEX have to be simulated, modified, or even eliminated. This requires a major effort to build the system. For example, UDP/IP is not offered in VERTEX. Communication packages in RAID was modified so that messages to or from remote servers were through host processor. Similar problem occurs in I/O. So we have to simulate it by passing messages to the host which has the ability to do the I/O. Another problem

encountered is the C Language on Ncube is not compatible with the one on SUN, such as, Ncube/7 does not allow to return a *struct* data from a function, enumeration variables have different allocation sizes, etc.. Besides, Ncube/7 was unstable and unreliable, improper message passing and memory accessing could cause system hang easily. Our progress on the Ncube machine was quite slow and was discouraging.

The Sequent/Symmetry incorporates from 2 to 30 identical processors and a common memory. This architecture has the benefits of resource sharing and communication among different processes. Most important of all, it runs DYNIX operating system which is a 4.3BSD UNIX descendant. Since the Sequent/Symmetry is a shared-memory multiprocessor, we can also modify the communication library in RAID to use shared memory to transmit internal messages instead of UDP. A message is called internal if both the sender and receiver are at the same site and same machine. Thus, our goals for the modification for the new version RAID were to:

(1) Allow multiple tasks to be run simultaneously.

(2) Servers use shared memory to transmit internal messages.

(3) Keep RAID's characteristics unchanged as much as possible while transferring RAID to the Sequent/Symmetry.

The next section presents the design of P-RAID. Section 3 describes Sequent and DYNIX features and some implementations strategies in detail. In Section 4, we devised an experiment to compare both RAID and P-RAID. Performance measurements are analyzed in Section 5. Finally, Section 6 gives some conclusions and indicates the future work.

2. P-RAID Design

P-RAID reduces system overheads and communication costs by running servers concurrently and by using shared memory to transmit internal messages. In P-RAID, the communication package and system interface were changed, while the conceptional organization of RAID is unchanged.

2.1. Changes to the communications library

Figure 3 shows communication path of P-RAID. The servers AC, AM, CC, and RC share a common memory for internal communication. Each server maintains a list of buffers for incoming internal messages, as shown in the Figure 4. Messages in the list are read in First-In-First-Out order. The message list structure is almost the same as we did in [2], in which only one message list is used. Both the buffers and pointers for
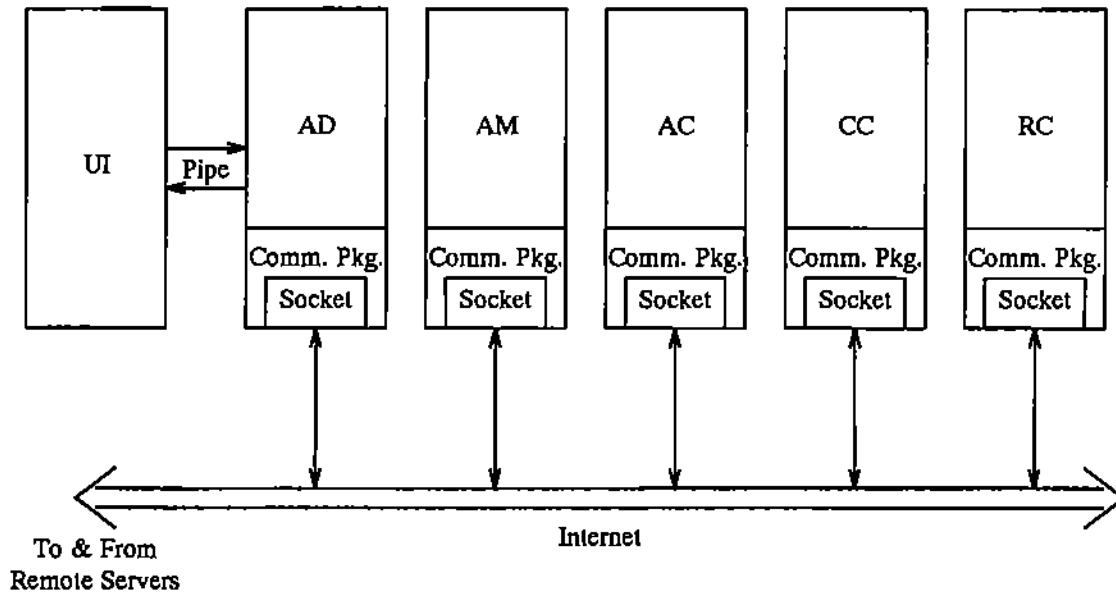
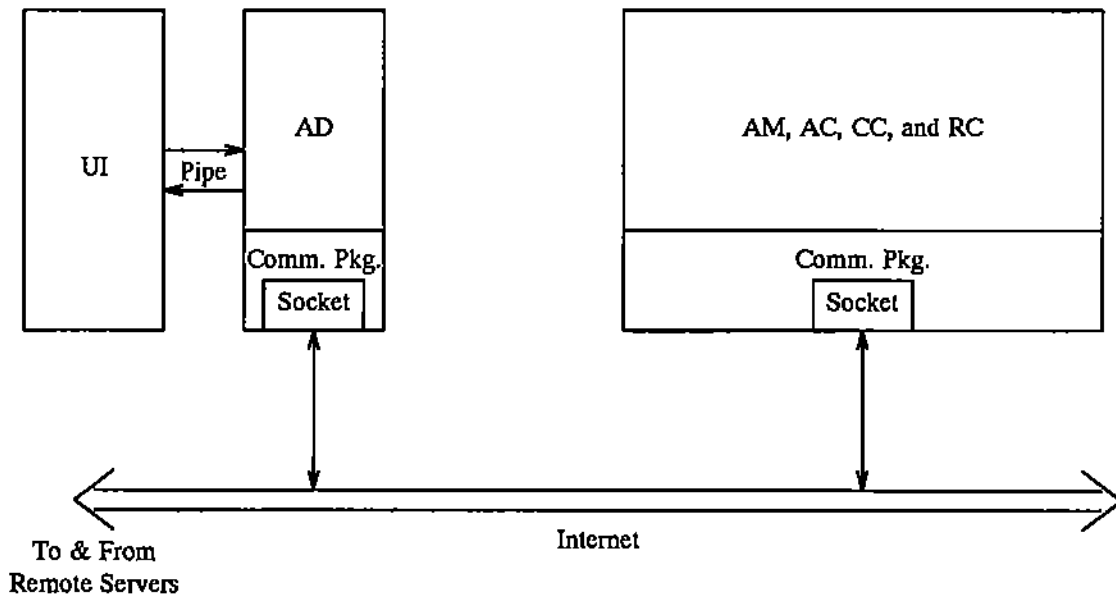Figure 2(a): The RAID communication path
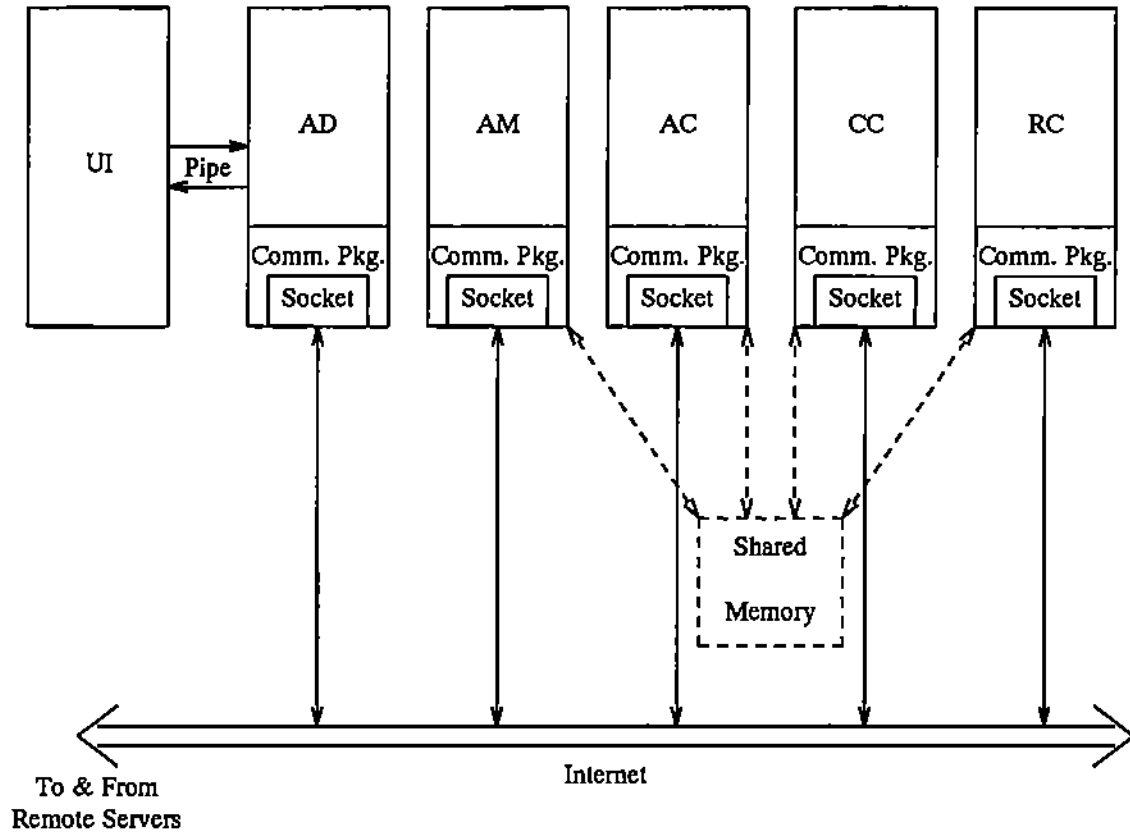
Figure 2(b): The merged-server RAID structure

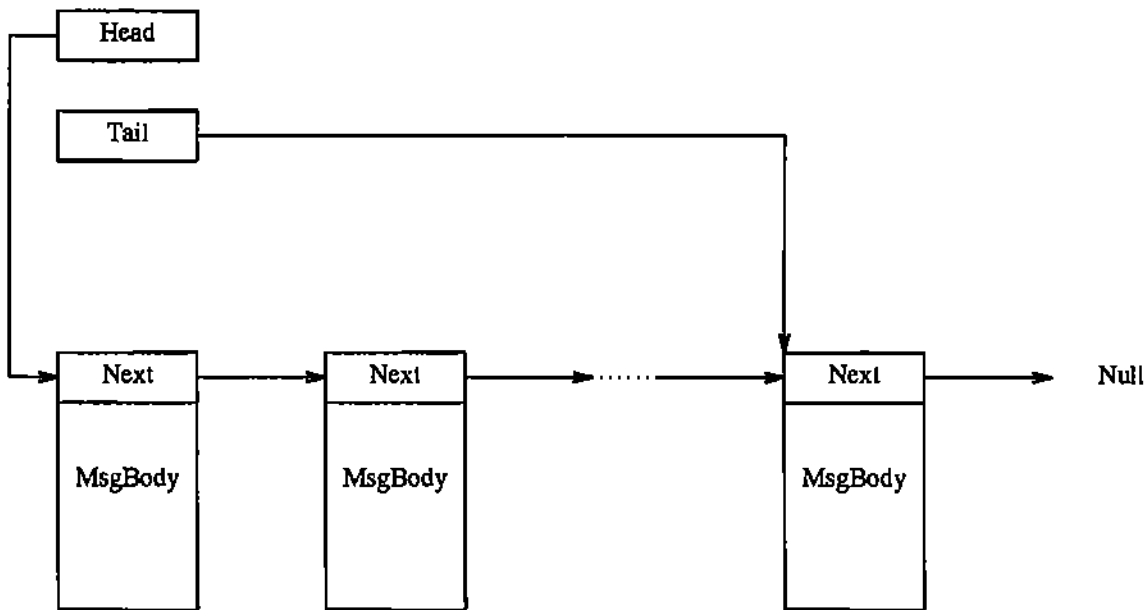Figure 3: The P-RAID communication path

Figure 4: The message list structure

these buffers are stored in the shared data area because other servers may access them. Each server can send an internal message to the other server by appending the message to the receiving server's queue. Once the message has been processed by the receiving server, the buffer is freed and returned back to shared data area for later use.

If the message destination were in the internal server, the message would be sent through shared memory; otherwise, it would be delivered via UDP. Although there are two kinds of message destination, the same calling interface was used for internal and external messages. Therefore, the changes in communications library are transparent to server and it requires no changes to the server codes.

Since each message list can be accessed by more than one server simultaneously, a lock is used for each list to guarantee mutual exclusion. Each server must get a lock of the list in order to send or receive a message. Further, each server only holds one resource(i.e. a list) until it finishes the operation. The server does not acquire additional resources while it is in the critical section. Thus, one of the deadlock necessary conditions *"Hold and Wait"* (see [5]) does not exist. Therefore, the deadlock problem will not occur.

We did not adopt the communication kernel design in YACKOS[4]. The reasons are as follows. First, servers in P-RAID need to use UDP to communicate with external servers, while YACKOS is only built on shared memory. We still have to write UDP codes if we call YACKOS communication routines. This will cause different calling interfaces to servers. Second, since RAID has already provided a layered communication with its own naming space, the modifications to the RAID code is only limited to some layer which will be simpler than changing most parts if using YACKOS. Third, YACKOS uses fixed buffer length. This is not suitable for RAID because message length in RAID is variable.

2.2. Changes to the system interface

A process called **starter** is used to let servers AC, AM, CC and RC be able to access a common memory. The starter is created at system initial stage. Then, it can be forked to be four processes, each executing a RAID server code as it is created individually. Therefore, this change is transparent to the remote site of RAID. The starter is designed to be configurable at run time with any combination of servers. In addition to those advantages discussed in [2], this flexible design of starter let it be extendible to allow more multiple ACs, CCs, RCs, or AMs could be in a RAID site in the future.

3. Implementation

This section briefly outlines the features of the Sequent/Symmetry machine that are relevant to the implementation of P-RAID. We describe the modifications to the different components of the RAID software.

3.1. Sequent DYNIX features

The Sequent/Symmetry available to us has 6 16MHz Intel 386 series processors [3]. All processors share a single pool of memory and operate on a peer basis, executing a single copy of DYNIX operating system. A process in any state can be executed on any processors. DYNIX has a parallel programming library which supports data and function partitioning applications for parallel processing. The system calls provided by the library can help users create sets of processes, schedule tasks among processes, and synchronize processes via shared memory. The following system calls were used:

shmalloc:

Allocate shared memory for data structures whose size is determined at run time.

shfree:

Release data structures previously allocated back to shared memory.

s_init_lock:

Initialize a lock.

s_lock:

Lock a lock. It spins as long as is necessary to acquire the lock.

s_unlock:

Unlock a lock.

Users determine the data that is shared among processes. In DYNIX C programming language, shared variables is declared by adding keyword **shared** to the variable's declaration statement. Accessing a shared memory among processes can be achieved by using the fork() system call. Since the fork() system call creates a duplicate copy of the current process, the child process inherit the parent's complete memory image, including the right to access shared memory.

3.2. Overview of the implementation of P-RAID

Figure 5 shows the pseudo code of the process **starter**. Its main functions are to declare shared variables and to create servers at the P-RAID system initialization stage. The two pointers **Head** and **Tail** are declared to be shared array structures indexed by server type numbers. Thus, a server can access the proper message in a list by the use

of two pointers **Head[wType]** and **Tail[wType]**, where **wType** is a server type number. The **Lock** array is also declared to be shared so that it can be accessed among servers to guarantee mutual exclusion.

After doing some initialization work, the *starter* process forks itself to four (or less; if specified) processes, each of them executing one of AC,CC,RC,and AM servers' functions. From now on, each server is a process just as it is created separately in RAID. The only difference is that once a message has been processed, it needs to check where the message buffer should be released back. Figure 6 shows the additional checking in the server code.

Since RAID is modular, the changes to the communication library were only made to the internals of the **SendPacket** and **RecvMsg** routines. The advantage is that these changes are transparent to the servers. Servers still call the same sending routine **SendPacket** to deliver a message to internal or external servers. Figure 7 is the modified pseudo code of these two routines. In the **SendPacket** routine, it enqueues the message to a proper message list if the destination is internal and use UDP otherwise. The receiving routine, **RecvMsg**, checks in turns for both internal and external message. Once a message (either internal or external) is found, it is returned immediately. One more modification is made to change blocking receive mode to be nonblocking receive mode for receiving external UDP messages. This is because we do not need to wait an external message instead use this time to check an internal message.

We use **s_lock** and **s_unlock** to guarantee only one process can enter the critical section; that is, accessing the shared message lists. For reducing unnecessary overheads, minimizing critical section length is one of our implementation considerations. For example, it is not necessary to let the test statement (testing there is an internal message or not)

    "if (Head[mytypeno] != Null) "

be in the critical section, because there is only one server can extract message from a list. For same reason, we also avoid putting message coping operation in the critical section.

```
#include    <parallel/microtask.h>
#include    <parallel/parallel.h>

/* shared data */
shared slock_t msglock[4];

shared struct in_msg *Head[4];
shared struct in_msg *Tail[4];

        .
        .
        .


main( argc, argv )
{

    Get some options for servers;
    Set siServers[server_type_no].active to be true if the server with
    server_type_no needs shared memory to send/receive internal msgs.

    Initialization step for some servers;

    if (siServers[TYPE_RC].active) {
       if (fork() == 0) { mytypeno = TYPE_RC;  /* TYPE_RC = 3 */
         RCmain();  /* split and create RC server */
         exit(1);
       }
    }
    if (siServers[TYPE_CC].active) {
       if (fork() == 0) { mytypeno = TYPE_CC;  /* TYPE_CC = 2 */
         CCmain();  /* split and create CC server */
         exit(1);
       }
    }
    if (siServers[TYPE_AC].active) {
       if (fork() == 0) { mytypeno = TYPE_AC;  /* TYPE_AC = 1 */
         ACmain();  /* split and create AC server */
         exit(1);
       }
    }
    if (siServers[TYPE_AM].active) {
         mytypeno = TYPE_AM;  /* TYPE_AM = 0 */
         AMmain();  /* split and create AM server */
    }

}
```

Figure 5: Pseudo code for **starter** process

```
ACmain()
{
    InitializeAC();

    while ( TRUE) {

        MsgType = RecvMsg(MsgBody, &RAIDAddr);

        ProcessACMsg( MsgType, MsgBody, RAIDAddr);

        if (InternalMsg(RAIDAddr))
            shfree(MsgBody);
        else free(MsgBody);
    }
}


ProcessACMsg(MsgType, MsgBody, RAIDAddr)
{
    switch(MsgType) {
            .
            .
        /* unchanged */
            .
            .
    }

}
```

Figure 6: Pseudo code for AC server

```
SendPacket(Msg,Addr)
{
   if (InternalMsg(Addr)) {
      Buf = shmalloc();
      copy Msg to Buf;

      s_lock(Lock[wType]); /* wType is a server type no. in Addr */
      /* critical section begin */
      Insert Buf to the message list by updating Head[wType],
      Tail[wType], and Next link in the list;
      /* critical section end */
      s_unlock(Lock[wType]);
   }
   else {  /* external message */

           .
         /* unchanged */
           .

   }
}

RecvMsg(Msg)  /* Msg is a an address of a pointer to Msgbody */
{
  while (TRUE) {
   if (Head[mytypeno] != Null) { /* an internal msg arrives */

      s_lock(Lock[mytypeno]);
      /* critical section begin */
      Msg = Head[mytypeno];
      Update Head[wType] and Tail[wType];
      /* critical section end */
      s_unlock(Lock[wType]);

      return Msg;
   }
   else {
         check if there is an external Msg or not; /* non-blocking */
         if yes, get it and return;
   }
}
```

Figure 7: Pseudo code for SendPacket and RecvMsg routines

4. Experiments

4.1. Statement of the problem:

This experiment measures the difference in performance of transaction processing in RAID and P-RAID.

4.2. Procedure

Our experiments consist of running a transaction benchmark [6] on several version of the Raid software. The times taken to execute the transactions, and the time spent in the Raid servers AC and CC are measured.

We ran transactions on merged server version of RAID, the P-RAID with UDP, P-RAID system with shared memory. The three experiments are outlined as follows:

- **EXPERIMENT 1:**

Execute merged servers version RAID on Sequent. Since there is only one process for the combined AC, AM, CC and RC servers, the process is obviously executed on a single CPU. This experiment is named MGSV_RAID.

- **EXPERIMENT 2:**

Run AM, AC, CC, and RC as four different processes in parallel on Sequent. However, they still use UDP to communicate with each other. This experiment is named UDP_PRAID.

- **EXPERIMENT 3:**

Execute P-RAID on Sequent. AM, AC, CC and RC servers also run in parallel except that internal messages are transmitted through shared memory. This experiment is named SHMEM_PRAID.

Each experiment is carried out in the following steps:

(1) We creates a process called driver which sets a parameter to define the multiprogramming level.

(2) The driver initiates a number of AD servers based on the multiprogramming level. In addition, a synchronization controller that triggers all AD's at the same time is initiated.

(3) Each AD server executes a transaction that inserts twenty tuples in a database relation. Each AD server waits for a trigger message before it commits the transaction. We use a system call *recvfrom()* to accomplish that. The blocking-receive mode is chosen so that all AD servers could be triggered simultaneously.

(4) The synchronization controller sends a control message to all AD servers via UDP. Each AD server continues its execution to commit the transaction and records the time when it begins and finishes.

(5) The driver terminates the measurement by killing all the processes.

We executed these experiments by varying the multiprogramming level of transactions from one to five. In each case, we repeated the procedure at least twenty five times and took the **best** (shortest) time instead of the average time. This is because Sequent/Symmetry is a public machines in our department and is shared by many users. The long delays must be from the unwanted waiting for CPUs, overheads on unexpected processes being swapped out, etc..

The times measured are only for the cost of committing a transaction. The timing for parsing the database query in our benchmark and its conversion to a transaction and the I/O times to retrieve data are ignored. Servers will become busy when several transactions are in the system. We measured "the elapsed time for committing transactions" and the elapsed time for committing transactions is the period from issuing commit requests till getting all the replies of commit or uncommit. We measured the elapsed time in order to know whether we gain performance improvement from parallel processing or not.

4.3. Data

Table 1 contains the measurements of the experiments stated in the previous subsection. It is clear that experiment SHMEM_PRAID produces the best data. In order to explain the performance times obtained in the Table 1, measurements were done to determine the time spent in each server for a transaction. The execution path for processing a commit request is

$$AC \rightarrow CC \rightarrow AC \rightarrow CC$$

in a pipelined manner. The time used for each stage is shown in the Table 2. Notice that the RC server does not appear in the execution flow because the experiments were conducted in one site only. The AM server is not included since I/O time is not included in the measurement.

Multiprogramming level	1	2	3	4	5
Experiment1 MGSV_RAID	0.09	0.18	0.28	0.43	0.59
Experiment2 UDP_PRAID	0.12	0.18	0.24	0.34	0.47
Experiment3 SHMEM_PRAID	0.09	0.13	0.19	0.31	0.44

Table 1: Elapsed time of transactions execution (in seconds)

Stages	1st:AC	2nd:CC	3rd:AC	4th:CC
Time	0.04	0.03	0.01	0.01

Table 2: The time used in each stage for processing a transaction (in seconds)

4.4. Discussion

For the case of only one transaction, experiment MGSV_RAID and experiment UDP_PRAID have the same elapsed time since both use shared memory to transmit internal messages. Experiment SHMEM_PRAID does not show any speedup due to parallelization for the case of the sequential execution in one transaction. Experiment UDP_PRAID has the worst data in this case due to the use of UDP to communicate with internal servers.

When two transaction run simultaneously, experiment SHMEM_PRAID takes the advantage due to the parallelism. >From Table 2, we note that most of the time spent in processing a transaction request is in the first two stages. Therefore, if overlapping execution is permitted between two successive transactions, the performance could be improved. Figure 8(a) shows the absence of transaction overlap in experiment MGSV_RAID. Note that an earlier transactions is finished before the new transaction is initiated. The time to complete the second transaction is shown as follows.

$$(0.04 + 0.03 + 0.01 + 0.01) + (0.04 + 0.03 + 0.01 + 0.01) = 0.18 \text{ (second)}$$

(waiting time)   (processing time)

A different execution is shown in Figure 8(b). The initiation of the second transaction occurs shortly after the first transaction completes the first stage of the pipeline. Therefore, the expected elapsed time for the second transaction is as follows.

$$0.04 \quad + \quad (0.04 + 0.03 + 0.01 + 0.01) = 0.13 \text{ (second)}$$

(waiting time)   (processing time)

These two numbers are exactly the same as appeared in the Table 1 and show a speedup of 28% ([0.18-0.13]/0.18). Transactions in experiment UDP_PRAID took 0.18 second instead of 0.12 * 2 = 0.24 second also due to the parallel execution of two transactions.

However, the third stage in the first transaction processing may collide with the first stage in the second transaction processing if they both require the AC server. Similar situation occurs in the fourth stages of the first transaction and the second stages of the second one since they compete for the CC server. These collisions are not severe since in the two collision stages, the time spent in one is shorter than in the other one. But, the additional delays appear due to the collisions when three or more transactions arrives. So the times in experiment SHMEM_PRAID are

$$(0.04+0.04) + (0.01+0.01) + (0.04+0.03+0.01+0.01) = 0.19 \text{ (second)}$$

(waiting)   (collision delay) (processing time)

for the third transaction.

Currently, there are only six CPUs on our Sequent machine. Therefore, when more transactions request commit, servers have to compete for CPUs. Thus, additional overhead due to process switching may occur. This is why higher numbers for transaction times than expected are shown in the Table 1. For example, when multiprogramming level is five, the elapsed time for experiment 1 is 0.59 ms which is greater than 0.9*5. However, in each case, the parallel execution such as in experiment UDP_PRAID and in experiment SHMEM_PRAID has a better performance than in experiment MGSV_RAID.

5. Conclusion and Future Work

Our study demonstrates how RAID developed for single processors in distributed environment can be ported on a shared memory multiprocessors system. We have shown the situations when the performance of the RAID system while maintaining modularity can be improved. We suggest parallel processing of transactions and the use of shared memory for communications. We also designed and conducted several experiments to compare the performance of sequential version of RAID and P-RAID. The experiments were devised based on the assumption that the transactions arrival rate is high enough so that several transactions wait for each server.

We experimented with the P-RAID with only one site since we had only one Sequent machine available to us. In the distributed version of P-RAID, the RC server (that manges replicated data) will be included in the execution flow for maintaining consistency of the replicated copies of data. It will cause the increase of both internal and external messages and the degree of overlapping executing transactions, therefore the actual speedup gained by P-RAID may change. This can be analyzed and proved by using the collision vector and reduced state-diagram in [7]. We believe that P-RAID implementation will perform better than the other two implementation because of its message passing mechanism and parallel execution of the servers.

The experiences learned from implementing the PRAID:

- Before migrating large software like RAID to a new system, one should consider the issue of compatibility of system environment that includes compiler and operating system. The target machine's environment should be a superset of the source machine from which we migrate the software. Especially, the new environment's compiler of the language in which the software is written in should support all the features used in the software. The system calls used in the software should not cost excessive effort to implement. calls.

- When trying to utilize the specific computer architecture to improve the efficiency of the code, one tends to make the program machine dependent. This causes problems when the migrating the software to a new environment. For example, Sun workstation environment in which RAID was developed is quite different from the Ncube/7 environment let alone compatible.

In future, we plan to study the feasibility and implementation of different methods to exploit parallelism. They include:

(a) Use multiple copies of servers in each site of Raid. Currently each site has one of these servers. This concept is similar to the design of squads [8]. Squads is a homogeneous group of processes that cooperate to provide a given service. We will study how an increase in the number of a server in a site would affect the performance.
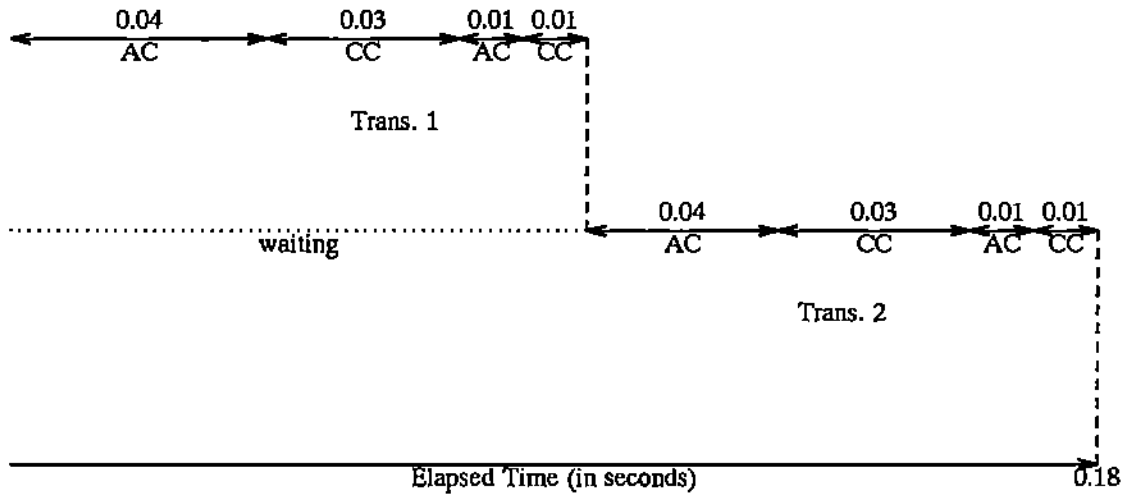
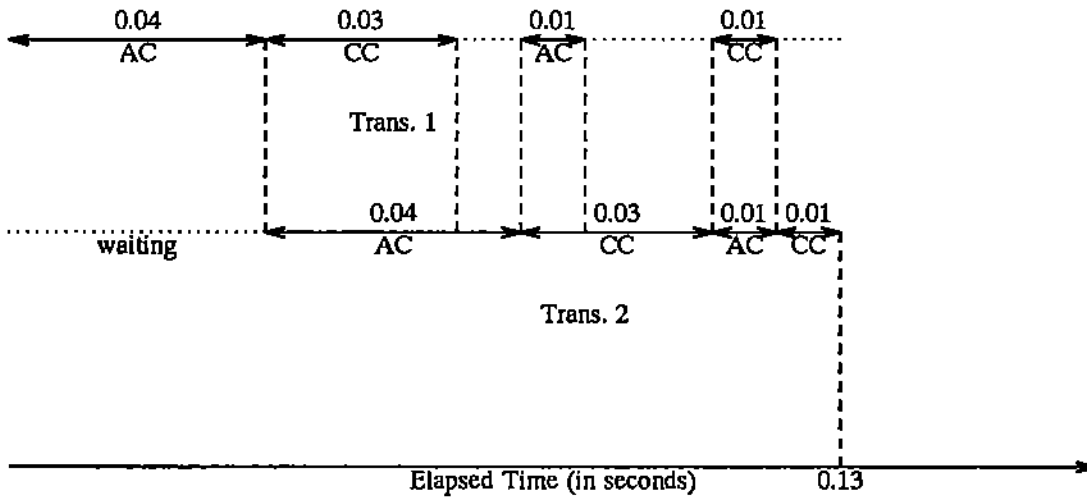Figure 8(a): No overlap of transactions in experiment MGSV_RAID

Figure 8(b): Overlap permitted among transactions in experiment SHMEM_PRAID

(b) Exploit parallelism in some or all of the servers by mapping them onto several processors. This idea breaks the one-process-per-server thinking which Raid has used in the past, but allows greater use of parallelism.

(c) Similar to (b), but we implement servers as lightweight processes (threads), rather that full UNIX processes. This could save both communication and context-switching overhead because the threads' shared memory space can be used for message passing, rather than costly operating system primitives. Many modern operating systems such as Mach[9] and Amoeba[10] support thread primitives.

In each one of above studies, we can measure speedups, and show scalability to large number of processors.

Acknowledgements

We thank C. Koelbel for his help in the initial stages of our design. We thank A. Helal and E. Mafla for their suggestions that led to the design of the experiments.

References

[1] B. Bhargava and J. Riedl, "The Raid Distributed Database System", IEEE Transaction on Software Engineering, Vol. 15, No. 6, June 1989.

[2] C. Koelbel, F. Lamaa, and B. Bhargava, "Efficient Implementation of Modularity in RAID", Proceedings of the USENIX Workshop on Distributed and Multiprocessor System, Fort Lauderdale, FL., Oct. 1989.

[3] Osterhaug, Anita "Guide to Parallel Programming on Sequent Computer Systems", Sequent Computer Systems, Inc., Beaverton, Oregon, 1987.

[4] R. Finkel and D. Hensgen, "YACKOS on a shared-memory multiprocessor", Technical Report No. 125-88, Univ. of Kentucky Department of Computer Science, 1988.

[5] J. Peterson and A. Silberschatz, "Operating System Concepts", 2nd edition, Addison Wesley, 1985.

[6] D. Bitton. D. DeWitt. and C. Turbyfil, "Benchmarking database system:A systematic approach." in Proc. VLDB Conf., Oct. 1983

[7] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing", McGRAW-HILL, 1984.

[8] D. Hensgen and R. Finkel, "Dynamic server squads in YACKOS", Proceedings of the USENIX Workshop on Distributed and Multiprocessor System, Fort Lauderdale, FL., Oct. 1989.

[9] R. Rashid, "Threads of a new system", Unix Review, 4(8), August, 1986.

[10] S. Mullender, G. Rossum, A. Tanenbaum, and R. Renesse, "Amoeba: a distributed operating system for the 1990s", IEEE Computer, May 1990.