in: Proc. Int. Conf. on Parallel and Distributed Information Systems (PDIS'94), Austin, Texas, 1994, pp. 257-260.

Performance Evaluation of the Remote Cooperation System in PRIMA

Michael Gesmann
University Kaiserslautern
67653 Kaiserslautern, Germany
email: gesmann@informatik.uni-kl.de

Abstract

In order to achieve acceptable performance in enhanced DBMS different kinds of parallelism have to be enabled by appropriate implementation capabilities. The Remote Cooperation System (RCS) offers these capabilities. It is the basic communication system of our enhanced DBMS called PRIMA. In this paper we present different implementations for RCS and an analysis of their performance behavior. In order to get a better assessment of these implementations, we compare our measurements to those obtained with conventional remote procedure call facilities offered by operating systems. The results demonstrate that RCS is suitable for supporting parallel query execution.

1 Introduction

DBMS for enhanced applications (NDBMS) need parallel query execution strategies in order to obtain short response times. Because of the highly meshed and cyclic structures of complex objects, data distribution and parallel execution of a single operator on separate partitions as used in conventional DBMS, are not applicable. Hence, in NDBMS, we need a closely coupled shared everything architecture [1].

In order to allow for suitable intra-query parallelism as an essential design goal in NDBMSs, we need an appropriate runtime environment. Applying the principals of client/server architectures, the NDBMS PRIMA [2] which has been implemented at the University of Kaiserslautern is divided into multiple components interacting as clients and servers. In order to implement efficient and transparent communication facilities we designed the Remote Cooperation System (RCS). In this paper we will examine its performance characteristics. Therefore, we will consider three different ways for allocating the components to processors:

1. **distributed allocation**: All components on separate distributed processors communicating via messages.
2. **local allocation**: All components on a single processor (workstation) communicating via shared memory. However, since there is only a single processor, timesharing is possible, but not genuine parallelism.
3. **shared allocation**: All components on a set of closely coupled processors communicating via shared memory.

Of course, a lot of work on standards for client/server communication has already been done. This work mostly concerns data transfer between client applications and the database server. However, in our implementation much more communication events occur that often transfer only small amounts of data. In order to avoid additional operating system overhead some design decisions on RCS's implementation are very similar to those made in the Raid project [3, 4]. However, our system is primarily designed for a closely coupled system in order to exploit parallel query execution. Therefore, we rely on a different system architecture. Furthermore, we investigate shared-memory synchronization by busy-waiting latches not considered in other papers.

The next section outlines the requirements RCS has to meet and describes the most important features of RCS. Section 3 shortly presents different implementations. In Section 4, we describe measurements performed to evaluate RCS and discuss the results obtained. The paper concludes with a short summary.

2 RCS - conceptual view

In this section we shortly introduce the functionality and some realization issues of RCS. More detailed presentations can be found in [5] and [6].

Based on an operation decomposition, we developed a client/server model that defines the framework for database processing in PRIMA; in principal, a client can issue requests for **services** to be performed by dedicated **server processes**. A set of server processes processing specific services is called **server**. The execution of a requested service by a server is called **task**. Hence, multistage requests to services offer capabilities for pipelining. Furthermore, execution of independent concurrent requests in different server processes exploits additional parallelism.

On principal, to achieve any parallelism we have to separate task invocation and asynchronous result reception by the functions *Remote_Service_Invocation* and *Get_Service_Result*. In order to realize pipelining RCS needs a function *Reply_Part_Of_Task* which transmits partial results produced so far.

In order to perform communication between clients and servers RCS has to deal with different mechanisms, e.g. messages between different hosts and shared memory locally on a host. This has to be done transparently for the application programs. Furthermore, when mapping services to processes, we have chosen a so-called single-process multithreading implementation where a single server process executes all tasks addressed to a service implemented there. Whenever a waiting situation occurs, the server has to break the task's execution and to choose another task for its execution. This approach avoids unnecessary operating system overhead (process creation, administration, deletion) and enables us to enforce our own application dependent scheduling strategies. In order to prevent unnecessary context switches we can integrate multiple services within a single process. Furthermore, we enable static replication of server processes which allows tasks to be executed in any of these processes and after an interruption, they can be continued in any available process.

3 RCS - implementation view

Now, we outline two different implementations of RCS used in our system. An initial implementation called old RCS (oRCS) uses datagram sockets as basic communication mechanism in every process. Since this implementation did not perform very satisfactory we implemented a new RCS (nRCS, Fig. 1) which should avoid oRCS's sophisticated protocols with time-out mechanisms and acknowledge messages guaranteeing correct transmission of data. Furthermore, the receiver should not handle incoming data asynchronously. This allows continuous task execution. Communication via shared memory should be enabled whereever possible, but communication across host boundaries is handled by a dedicated communication process (CP) which can run on a separate processor in case of a multiprocessor machine. W.r.t. internal communication a CP behaves like any other server process. Of course, on a single processor this implementation causes multiple additional context switches in the case of remote communication. Appropriate data structures in the shared memory taken from and released to preallocated pools support task management. The main problem in task management is to synchronize the access of multiple concurrent processes to shared data structures. A suitable solution should enable as much parallelism and at the same time should keep synchronization overhead as low as possible. For these reasons, we implemented nRCS using three different synchronization mechanisms to evaluate and compare their influences on RCS's performance, i.e. conventional semaphores, busy waiting latches, and finally latches releasing processors if requests cannot be satisfied immediately.

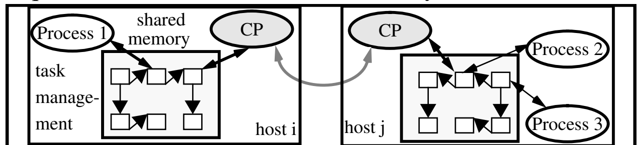

Figure 1: Architecture of nRCS

4 Measurements and results

In order to get a quantitative evaluation for our RCS implementations we describe measurements and their results. A comparison with rpc-facilities of operating systems, which are the only suitable mechanisms we have for comparison, allows a better assessment of RCS.

The aim of our measurements was to get real performance data. Furthermore, we wanted to compare nRCS to oRCS, and to rpc-facilities. Finally, these measurements should yield a better understanding of nRCS's operation.

In the shared allocation case we used a SEQUENT Symmetry (S27) with 8 processors (6MHz Intel 80386) running DYNIX V3.0.18. Every processor had a cache of 64 kbytes. The processors and the shared memory were connected by a 64-bit system bus with a channel bandwidth of 80 megabytes per second. Furthermore, we examined the distributed allocation and the local allocation case, using workstations (33 MHz SUN SPARC ELC) with SUN OS 4.1.3, connected via Ethernet. All measurements were executed in exclusive mode in a conventional environment, i.e. neither the hardware nor the operating system had been modified for our measurements. We measured response times for synthetically generated tasks. Usually, task parameters and result values were empty, servers only accept tasks and immediately reply. Note, that even if the result value was empty the server still had to reply the task. In all following figures, the y-axis shows the average response time for a single task in milliseconds, i.e. we measured the times from the first task invocation until all tasks had been executed and divided this elapsed time by the number of totally executed tasks.

Initial measurements compared the available communication mechanisms (rpc, oRCS, nRCS) with each other using the described allocation schemes (distributed, local, shared). Since oRCS and nRCS offer some features rpc-facilities do not have, we subsequently compared oRCS and nRCS in more detail. Finally, since shared allocation is the preferred one in our NDBMS application we examined performance characteristics of nRCS in this case in further detail.

Comparison of rpc, oRCS, and nRCS

In the distributed and in the local case, busy waiting latches have not been used. Due to the restrictions of rpc-facilities which do not allow concurrent invocation of tasks the implementations using nRCS and oRCS performed synchronous task invocations, too. Furthermore, we have been forced to vary the number of client processes in order to achieve parallelism. With distributed allocation, only the server process on one side and the client processes on the other side were distributed, i.e. all client processes were running on a single processor. The results are illustrated in Figure 2. Note, due to the variation in processor power, the measured times on the multiprocessor and on the workstations cannot be compared directly.

In the shared allocation case nRCS with semaphores behaved worst (Fig. 2a, nRCS, semaphores) since semaphores require kernel calls causing unnecessary context switches in waiting situations. Moreover, nRCS with busy waiting latches (Fig. 2a, nRCS, latches) outperformed all other implementations including the rpc-facility. With more than one client process the server became the bottleneck. Therefore, response times did not decrease any further.

In the local allocation case (Fig. 2b) rpc outperformed all RCS-implementations. Furthermore, rpc and oRCS response times did not decrease with increasing client parallelism because the processor was always running a client or a server process even in the case of a single client process. The bad response times for nRCS with a single client and the drastically improving behavior with higher degrees of client parallelism were surprising. Further investigations turned out that the SUN OS signals waking up sleeping processes were responsible for this bad performance. When executing the same measurement on the SEQUENT, running the DYNIX operating system, with only a single processor switched on, yielded about constant response times, not shown in the figures.

With distributed allocation (Fig. 2c), the rpc-mechanism again outperformed both RCS implementations. Compared to local allocation, the overhead for double task creation and result transmission in nRCS across host boundaries caused about double response times. Due to the signals behavior, the vast number of context switches did not effect the results noticeably.

Further comparison of oRCS and nRCS

Different degrees of parallel task invocations have been investigated in the following measurements consisting of single client and server processes. In the shared allocation case, nRCS with latches led to much better results than oRCS and nRCS with semaphores (Fig. 3). Since waiting situations did not occur in this environment, both implementations with latches did not differ except in the case of sequential execution. Due to task-management overhead response time increased linearly. Figure 3b shows the results on the SUN workstations. Because of less context switches oRCS behaved better with distributed allocation than in the case of local allocation. Contrarily, nRCS produced better results in the local case because context switches could be saved. In nRCS with distributed allocation, an increasing number of parallel task invocations led to a decreasing number of context switches between CP, client, and server processes. Here, in a single process activation multiple tasks or results could be processed. Therefore, the overhead caused by operating-system signal handling decreased.

In order to evaluate the performance of task invocations inside a single process a client created tasks for a server implemented in the same process. Since there were no context switches necessary, we used the implementation with latches keeping the process running. Hence, the performance behavior was similar in principal (Fig. 4). However, since the SUN processor is much more efficient, it gained shorter response times. With separate measurements we compared these results with the situation where client and server were allocated to two processes. As expected, on the SEQUENT the implementation with separate processes behaved better than the one-process implementation because the load could be distributed over two processors. On the other hand, the one-process implementation led to better results on the SUN workstation because there were no context switches after task invocations.

Detailed analysis of nRCS with shared allocation

Using as few processes as possible keeps the cost for context switching low, whereas a growing number of processes facilitates load balancing. Therefore, we investigated varying numbers of client and server processes.We executed measurements with one, three, five, and ten client and server processes, respectively. Compared to the case of five clients and servers measurements with more processes always produced worse response times. Generally, if the number of processes exceeded the number of available processors (in our case 8) context switches between the participating processes led to worse response times (Fig. 5) It was not useful to add multiple server processes to a single client because the client is the bottleneck. This results from the task-management implementation where more work is done at the client-side. Adding more clients yielded better performance for lower levels of parallel task invocations since clients could work in parallel on their own processors. With a higher degree of parallelism this performance degraded because of the task administration overhead in the server process. Note, every process produced the indicated number of parallel tasks.

5 Conclusions

In order to optimize mechanisms supporting parallel execution of queries on complex objects in our NDBMS PRIMA we introduced RCS which provides a client/server communication mechanism. Compared to conventional remote procedure call facilities it offers an extended functionality which enables additional kinds of parallelism needed in complex object processing. Implementations using different communication and synchronization mechanisms have been presented. nRCS is particularly designed for closely coupled multiprocessor systems which are best suited for NDBMSs. In order to get an accurate understanding of the performance characteristics of the implementations, we performed a couple of measurements. Summarizing their results, nRCS fulfills the requirements of our application. Especially in the case of shared allocation on the multiprocessor system, it achieves better response times than oRCS. A comparison with the remote procedure facilities which provide a much simpler interface persuades us that nRCS is a suitable basic communication system in order to efficiently realize our PRIMA system. We observed different performance behavior for various numbers of client and server processes, but due to of the simplistic scenario a general statement about performance characteristics in these cases is not possible. This effect has to be explored in more detail in a more complex application consisting of multiple levels of client/server-relationships.

6 References

[1] Härder, T., et al: Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations, 3rd Annual Parallel Processing Symposium, Fullerton, CA, 1989, pp. 564-578

[2] Härder, T., et al: PRIMA - A DBMS Prototype Supporting Engineering Applications, 13th VLDB 1987, Brighton, United Kingdom, 1987, pp. 433-442

[3] Bhargava, B., Riedl, J.: The Raid Distributed Database System, IEEE Trans. on Software Engineering, Vol. 15, No. 6, pp. 726-736, June 1989

[4] Mafla, E., Bhargava, B.: Communication Facilities for Distributed Transaction-Processing Systems, IEEE Computers, pp. 61-66, August 1991

[5] Gesmann, M., et. al: A Remote Cooperation System Supporting Interoperability in Heterogeneous Environments, Int. Workshop on RIDE-IMS, Vienna, 1993, pp. 152-160

[6] Hübel, C., et al.: Using PRIMA-DBMS as a Testbed for Parallel Complex-Object Processing, Int. Workshop on RIDE Trans. and Query Processing, Tempe, Az., 1992, pp. 38-45