in: SFB-Bericht 23/94, University Kaiserslautern, 1994

Performance Evaluation of the Remote Cooperation System in PRIMA

Michael Gesmann
University of Kaiserslautern
Federal Republic of Germany
email: gesmann@informatik.uni-kl.de

Abstract

In order to achieve acceptable performance when executing queries in enhanced database systems, e.g. for engineering and knowledge-based applications, intra-query as well as inter-query parallelism are required. Therefore, systems exploiting different kinds of parallelism need an environment offering suitable implementation capabilities. This paper describes different mechanisms for supporting parallel query execution in enhanced database systems. In order to provide efficient intra- as well as inter-query parallelism in our enhanced database management system, we introduce the basic communication system, called Remote Cooperation System. Thereafter, we present alternative implementations and finally present an analysis of their performance behavior.

We implemented two different system architectures which will be compared in more detail. In order to get a better assessment of theses implementations, we compare our measurements to those obtained with conventional remote procedure call facilities offered by operating systems. The results demonstrate that the Remote Cooperation System is a suitable basic communication system in order to efficiently realize parallel query execution, especially in our enhanced database system called PRIMA.

1. Introduction

During the last years a substantial amount of research has been conducted on enhanced DBMS, also called Non-Standard Database Systems (NDBMS). In contrast to conventional DBMSs, these systems support complex applications, such as design or AI applications. Common requirements in almost all of these new application areas are the modelling and management of complex objects as well as the support of appropriate user interfaces. In order to meet these requirements a large number of data models and query languages have been developed [Mi88, PA86, Mi92] and evaluated in prototype systems. Experience in the interactive use of such systems revealed the utmost importance of responsiveness and performance when executing queries. To achieve a system behavior acceptable for the designer, it turned out that all system components have to be carefully designed and optimized; important results have been reported, for example on different storage structures [SS89, KD93, Te93], query optimizers [Sch93] and system architectures [Pa87, HMMS87].

Most of enhanced applications' work is done on workstations. The NDBMS only serves as a powerful data repository which has to execute a moderate number of cpu-intensive requests on complex structured data consuming a lot of cpu time. On the other hand, the application is typically driven by interactive dialogues of the user. Hence, we concentrate on response time as the primary performance measure and not on throughput. Due to the very long execution paths while constructing complex objects and the inherent parallelism of complex-object processing in NDBMSs, satisfactory response times for queries as the primary performance measure can only be obtained by parallel execution. As opposed to the well-known inter-query parallelism which is mandatory to increase throughput in every multi-user database system, intra-query parallelism becomes interesting for complex-object processing in NDBMS. Opposed to conventional DBMS data distribution and parallel execution of a single operator on these separate partitions is not applicable because of the highly meshed and cyclic structures of complex objects. On the contrary, we need a closely coupled shared everything architecture [HSS89] in order to support parallelism in complex object assembly [KGM91].

In order to allow for suitable intra-query parallelism as an essential design goal in NDBMSs, we need an appropriate runtime environment. Therefore, as done in operating systems, the principals of client/server architectures should be applied to NDBMS development, too. In this framework, the application is considered a client of the NDBMS server and, in turn, when decomposed appropriately, the components of the NDMBS-server form client/server relationships. In addition to such a software structure which facilitates concurrent execution of components, we have to carefully consider the mapping to hardware resources to take advantage of the potential parallelism. In our case, there are three different ways in order to allocate these components to processors:

1. All components are allocated on separate distributed processors (workstations), this is called **distributed allocation**. In this case remote communication between different components has to be performed via messages and the different components may work in parallel on their own processors.
2. All components are allocated on a single processor (workstation), this is called **local allocation**. Here, they can efficiently communicate via shared memory. However, since there is only a single processor, timesharing is possible, but not genuine parallelism.
3. Finally, all components may be allocated on a complex of processors connected via shared memory (shared everything multiprocessor machine), this is called **shared allocation**. Here again, local communication via shared memory can be applied. In this case, the components may run in parallel on the available processors.

Obviously, multiple refinements are possible. For example, typically, the DBMS-application is running on a separate workstation, but, in principle, it is also possible to run the application on the same processor (complex), where the DBMS is running. Furthermore, as will be discussed later, a single process representing one or more components on a processor may not fully exploit its processing capacity (horse power). Especially in the case of synchronous I/O or page faults, a process is forced to wait leaving the corresponding processor idle. For these reasons, it may be advantageous to allocate more than one process to a processor, even in a dedicated and coordinated situation such as DBMS processing. On the other hand, some components are so closely related, i.e. there are frequent calls between them that a process or even a processor boundary between them may seriously degrade performance. Hence, some kind of multi-threading of these services in a single process seems to be favorable in such situations. Here we assume that the overhead for multi-threading is by far less expensive than that of multiprocessing. Furthermore, this approach allows simple integration of application-specific scheduling strategies without intervening in operating systems.

The NDBMS PRIMA [HMMS87] developed at the University of Kaiserslautern is incorporating such a client/server architecture aiming at parallel processing on complex objects. Since this system is running on a network of workstation as well as on a shared everything multiprocessor machine providing all the previously introduced allocation schemes for processes to processors, we need a basic communication system, which transmits requests from a client to a server as well as the results from the server to the client. Therefore, it has to hide aspects of interoperability and heterogeneity in a network of workstations with different hardware running different operating systems. In this paper we focus on RCS's functionality and efficiency, whereas the issues of interoperability and heterogeneity are discussed in [GGS93]. Since conventional remote-procedure-call (rpc) mechanisms offered by operating systems do not support these aspects we could not use operating system services. For these reasons we designed the so called Remote Cooperation System (RCS) which provides an efficient and transparent communication mechanism for its application components and allows the invocation of services by rpcs. In this context, transparency means, the system has to choose the appropriate communication mechanism at runtime invisible for the application. In contrast to conventional rpc-mechanisms only supporting synchronous calls we enhanced the rpc-model by asynchronous calls and transmission of partial results in order to facilitate different kinds of parallelism between clients and servers as well as within a single server.

In this paper, we will examine different implementation alternatives for RCS in order to find the best implementation depending on the underlying hardware architecture and the allocation of components to this hardware. In order to be able to improve the PRIMA system, we first have to optimize the RCS by identifying and removing bottlenecks and inadequate design decisions. Furthermore, understanding RCS's performance characteristics is necessary to understand performance of its applications. Hence, the focus of this paper is a detailed performance analysis of RCS which will give some insights used for further performance evaluation of the PRIMA system.

The next section outlines the requirements RCS has to meet due to the client/server architecture of the PRIMA system and the most important features of RCS. Section 3 presents two different implementation approaches. In Section 4, we describe measurements performed to evaluate RCS and discuss the results obtained. The paper concludes with a short summary of the results.

Related Work

In the literature we find a lot of work on standards for client/server communication. However, most of this work concerns data transfer between client applications and the database server. Contrarily, we decomposed our DBMS into a client/server architecture which induces much more communication events often transferring only small amounts of data. In [BR89, MB91] the authors present the Raid project and investigate its performance characteristics. In order to avoid additional operating system overhead some design decisions on the system's architecture are very similar to our architecture. However, our system is primarily designed for a closely coupled system in order to exploit parallel query execution for nonstandard applications. Therefore, we allow concurrent execution of the same server in multiple processes. Furthermore, we not necessarily have to synchronize communication via shared memory by semaphores but busy-waiting latches are a very attractive alternative not considered in these papers.

Since this paper is not concerned about distributed cooperative work of autonomous, heterogenous, or interoperable systems, as investigated e.g. in [FHM93, MH92, SW93], we will not consider them here any more.

Finally, we have to explain, why we have implemented our own system and not chosen a standard approach (e.g. OSF/DCE, OMG/CORBA). First of all, there are historical reasons. When starting our project in 1988 existing communication facilities (remote procedure call facilities) did not support parallelism at all. Moreover, at that time these standards did not exist. Therefore, we had to implement our own system (i.e. RCS) dedicated to our requirements. Later on, we did not change to other protocols, because the implementation of our PRIMA system was done upon RCS and we did not want to reimplement the already existing components. Furthermore, none of these standards offers the required functionality we need in order to enable parallel query execution as well as to reduce operating system overhead.

2. The Remote Cooperation System - Conceptual View

In this section we present the decomposition of our NDBMS PRIMA into a set of components each implementing different services. Thereafter, we describe useful types of parallelism in complex-object processing in order to motivate some extensions of rpc-facilities for RCS which support these kinds of parallelism. Finally, we discuss some general realization issues for RCS.

2.1 PRIMA Architecture and RCS

The PRIMA system is implemented as a DBMS kernel architecture consisting of an application-specific layer and an application-independent kernel. The application-specific layer embodies functions needed for a particular application. For all aspects of data management these functions are, in turn, supported by operations of an application-independent complex-object interface provided by the DMBS kernel. This kernel system is responsible for all tasks of data and meta-data management, of mapping these structures to storage, as well as for transactions control. To obtain modularity and data independence, we have designed the kernel architecture as a hierarchically layered system structure [HMMS87]:

- The data system implements operations on complex objects offered at the interface of the kernel. It consists of a Compiler to translate, an Optimizer to optimize, and an DML-Execution to execute application queries on complex objects. These operations are internally transformed to operations on simple objects.
- The access system transforms these operations on simple objects to operations on blocks of the available storage structures. For this purpose, it refers to the available storage structures and access path structures.
- The storage system is responsible for the buffer and file management.
- Separate components of the system incorporate transaction and meta-data management services.

Based on this modularization and 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**. In the following, a set of server processes processing a specific service is called **server**. Of course, the service may appear as client to other services. But since every client in the kernel also operates as service, we do not distinguish between service and client as well as between server and client; therefore, we always call them service and server, respectively. Figure 2.1 shows

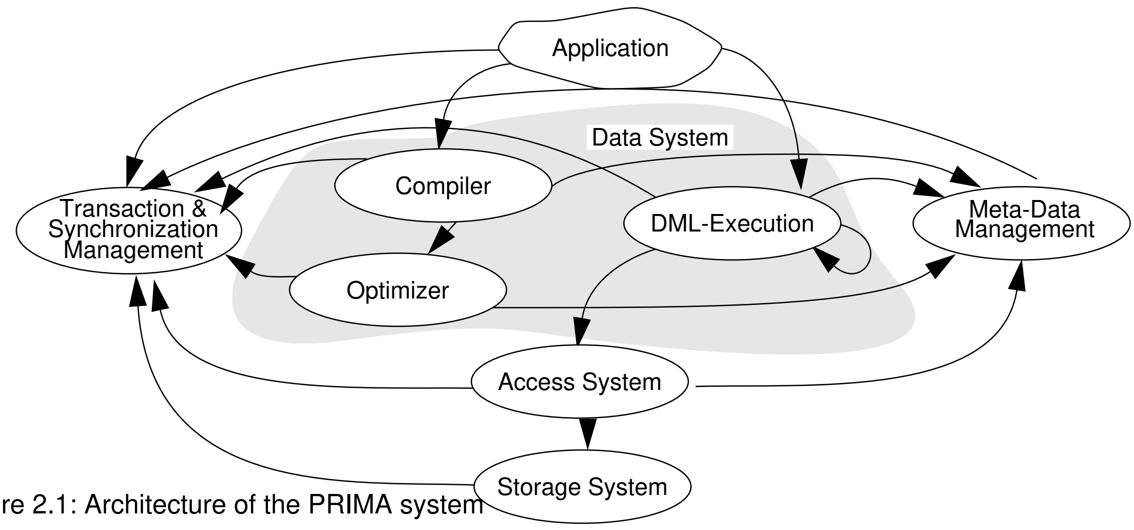

Figure 2.1: Architecture of the PRIMA system

the simplified decomposition of the system consisting of the services mentioned. The arrows indicate possible client/server-relationships. Apparently, the system is much more complex, but the view provided here is sufficient to discuss the most important properties of RCS, i.e. perform communication between clients and servers and administration of occurring tasks.

2.2 Types of Parallelism in PRIMA

In this subsection we shortly outline some types of parallelism in complex-object processing. As will be shown, we can generalize these mechanisms in our architecture.

Before executing a specific application query on a database, the Compiler and Optimizer transform it into a query evaluation plan (QEP) which consists of nodes representing elementary operators each producing a set of complex objects when executed by the DML-Execution server. In the case of sophisticated queries, such a QEP may become very complex that is, it consists of a large number of operators. In general there are two very obvious possibilities for exploiting parallelism while executing such a QEP. First of all, there is the inter-operator parallelism which appears in parallel execution of independent operators and in pipelining between dependent operators of the QEP. These mechanisms are useful in order to construct multiple components of a single complex object or to construct multiple complex objects in parallel. Furthermore, we can achieve intra-operator parallelism when invoking multiple tasks for a single operator of the QEP. In this case, every task produces a subset of the resulting complex objects. In order to realize these kinds of parallelism, the DML-Execution server internally has to perform multiple tasks in parallel.

These mechanisms, so far described only for the DML-execution server, are applicable in the whole system when replacing operators by services in the previous discussion. Then, we generalize pipelining to client/service parallelism when executing a client-task in the client and its invoked tasks in servers in parallel. Of course, pipelining is more specific, because it captures the data produced by the server for a client, whereas client/service parallelism does not include any dataflow. Inter-operator parallelism corresponds to inter-service parallelism when executing several tasks in different servers. Finally, parallel execution of several tasks in a single server, called intra-service parallelism, accords with intra-operator parallelism.

2.3 Conceptual View to RCS

Since rpc-facilities [DYN, SUN] only offer a synchronous interface, they cannot enable the previously introduced types of parallelism while executing a single query. In order to exploit parallelism, we need the following two extensions of conventional rpc-facilities.

In principal, to achieve any parallelism we have to separate task invocation and result reception. The functions *Remote_Service_Invocation* and *Get_Service_Result* achieve this separation. After invoking a task, the actually performed task may continue its execution, for example, it can create further tasks. It will accept the results asynchronously. On one hand, this extension enables inter-service and intra-service parallelism in starting multiple tasks before receiving their answers. On the other hand, it is a first prerequisite for pipelining, i.e. simultaneously executing a parent and its child task while the child produces input, to be processed by the parent task.

In order to realize pipelining RCS needs the following second extension to conventional rpc-facilities. In addition to the function *Reply_Task* which sends the final result of a child operator to its parent operator, we need a function *Reply_Part_Of_Task* which transmits a partial result which has been produced so far. On the client side, the function *Get_Service_Result* needs a flag indicating whether the received result is complete or not.

2.4 Realization Issues

RCS has to deal with different communication mechanisms. Communication between different hosts, e.g. between the application program on a workstation and the database system kernel on a multiprocessor machine, has to be done via messages. Of course, this is more expensive than communication via shared memory where sophisticated network protocols, copying of data, and failure handling can be avoided. RCS is responsible for choosing the appropriate mechanism transparently for the application program.

So far, services have been treated as abstract concepts. In order to map them to specific processing units offered by the operating system (i.e. the processes), there are two possible alternatives. First, using a so-called multi-process single-threading approach where every individual task may be executed in its own server process, dynamically created for this purpose. This approach introduces two main problems:

- First of all, there are the enormous costs for process administration, for process creation, process deletion and context switches. Obviously, there are at least two context switches produced by a single service invocation. Additional context switches arise if a task has to wait for events or for further (partial) results of a subtask. Note, we consider complex object processing which in turn is very cpu-intensive, we expect a lot of context switches when exploiting parallelism. Therefore, even if single context switches do not effect response times remarkably, the vast number of context switches leads to worse performance.
- The second problem when using a multi-process approach is that we cannot influence scheduling strategies without changing code of the operating system.

Lightweight processes (LWP) implementing sequential parts of a task called threads overcome the first problem because some implementations can perform context switches and process creations almost as fast as procedure calls [ALL89]. However, they are not available on all commercial (OS) products. For example, the DYNIX operating system on SEQUENT machines realizes LWPs by normal processes, e.g. creating a process takes about 25 ms. But even if LWPs were available the second problem still remains. Furthermore, LWPs are usually used for fine-grained parallelism with little exchange of data. Frequent waits and data exchanges require an efficient handling of events and partial results and it is not clear, how efficiently LWPs can handle these problems.

Alternatively, a so-called single-process multi-threading implementation, avoids these disadvantages. Here, all task executions of a service may be executed within the same server process. Whenever a waiting situation occurs, the server has to perform a task switch, i.e. to break the execution of the actual task and to choose another task for its execution. This approach avoids unnecessary context switches in the operating system which is considered much more expensive than our own task switches inside a process. Furthermore, this realization enables us to easily apply server and possibly application-specific scheduling strategies rather than leaving scheduling to the operating system.

The architecture described so far has two weak spots which may be resolved as described below. First, since task invocations between two different services may still cause context switches if the invoked task has to be executed in another process we can combine multiple (closely cooperating) services within a single program, e.g. the DML-Execution and the Access System. Hence, if every process is running on its own processor which is possible due to the relatively small number of different processes almost no context switches occur on a multiprocessor. This is the case because none of the services except the Storage System perform I/O and therefore they do not voluntarily release the processor. Note, in the further discussion, servers and server processes may include multiple services.

The second weak spot concerns restricted parallelism. Every service is embodied by a single server process. This of course does not allow parallelism within a single server. In order to overcome this problem we allow static replication of server processes (static multiprocessing), i.e. a specific program may run in multiple processes. Tasks may be executed in any of these processes and after an interruption caused by a waiting situation, they may be continued in any available process.

Finally, since we do not have shared memory between hosts [Me92] available and we do not want to implement sophisticated protocols to exchange state information about servers between hosts, servers are realized in a set of processes that do not span hosts. In order to enable parallel execution inside the server the chosen host might be a multiprocessor. Note, we intentionally distinguish between processors and hosts which may be single- or multiprocessor machines.

To sum up, this architecture allows all possible kinds of parallelism (client/server parallelism, server/server parallelism, pipelining) and it does not limit the available degree of parallelism which is determined by the number of processors. The allocation of services to server processes and the distribution of the servers over the available hosts is predefined, before starting the system, in a so-called configuration. Moreover, this architecture simplifies load balancing drastically, because it is reduced to a task scheduling problem in RCS which has to determine the next task to be executed.

3. The Remote Cooperation System - Implementation View

So far, we have only described the functionality of RCS. In this section, we outline two different implementations of RCS used in our system. Starting with an initial implementation called old RCS (oRCS) experiences showed that the interface of RCS is suitable for realizing the system. At that moment having the functionality needed was more important than performance. However, it turned out that it was very difficult to maintain oRCS. Some faults in the implementation which did not allow us to run the same service within multiple server processes on a single host led to another implementation called new RCS (nRCS). We will discuss different implementation alternatives for nRCS which may yield distinct performance characteristics. The measurements described in Section 4 are carried out with both systems and with different alternatives.

3.1 Initial Implementation

Since the PRIMA system is developed on a network of SUN workstations and since for reasons of load distribution processes had to run on different machines, we unconditionally needed mechanisms for communication across different sites from the very beginning of the project. Therefore, datagram sockets are used as basic communication mechanism. Figure 3.1 illustrates the implemented architecture for a set of processes running on two hosts.

At the time of its initialization, a client process establishes connections to all its service processes. These connections are maintained during the whole lifetime of the system because it would be too expensive to create new sockets and connections on each task invocation and result transmission. Parameters to be transferred are copied into a linear bytestream which then has to be disassembled into packages if the bytestream is too long to be sent at a time through the stream. Additionally, sophisticated protocols with timeout mechanisms and acknowledge messages have to be used to guarantee correct transmission of data. On the other

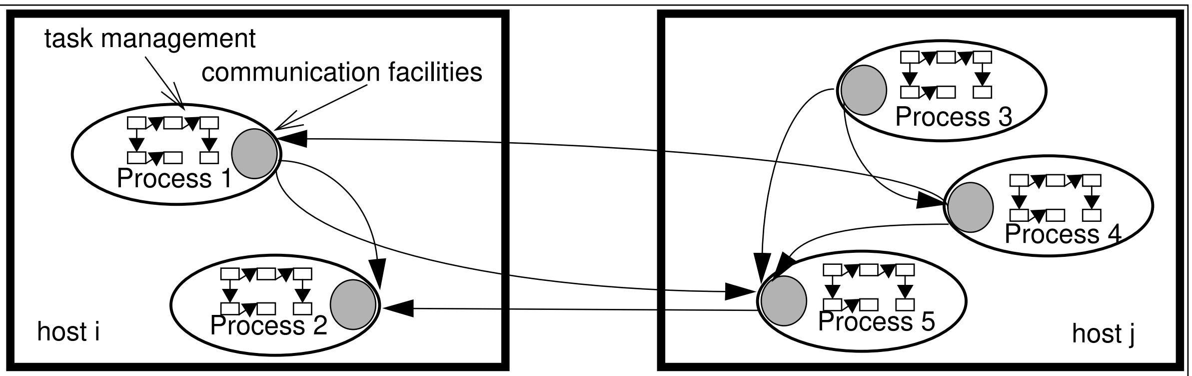

Figure 3.1: Architecture of the initial implementation of RCS (oRCS)

side, the receiver has to handle incoming data asynchronously. As a consequence, every received package causes an interrupt which will be processed immediately by an interrupt routine.

Another serious disadvantage of this implementation is that the data structures for task management are kept in a process' local storage. Hence, tasks cannot migrate from one process to another one which unnecessarily restricts the achievable real parallelism.

3.2 Improved Implementation

Our second implementation pays more attention to performance issues. Due to the processing characteristics, outlined in the introduction, nRCS was designed with special emphasis on shared allocation of NDBMS's components on a closely coupled multiprocessor system. But of course, it should run in a distributed network of workstations as well because the application running on a workstation and the NDBMS server have to communicate across the network. For these reasons, communication via shared memory should be enabled whereever possible, but communication across host boundaries should also be supported. Since the initial implementation has pointed out the functionality of RCS to be satisfactory and since we did not want to change the code of our system, the interface of RCS remained unchanged. As an important requirement, RCS should offer all runtime facilities described in Section 2, especially all mechanisms to assign services to processes.

3.2.1 Communication Mechanisms

In order to implement adjusted communication mechanisms we examined two different architectures:

- A first proposal considered a separate handling of local and remote communication in every process. In this case, local messages have to be exchanged via shared memory whereas remote communication has to be performed by message-based mechanisms to be included in every server process (c.f. Fig. 3.1). Hence, RCS internally has to distinguish between the two communication modes, and the code of RCS becomes more sophisticated. Furthermore, the size and complexity of the processes increases. In order to prevent blocking of the sender, the external communication again has to be performed asynchronously. Summarizing, this approach would again lead to some problems already encountered in the oRCS implementation.
- The second architecture is illustrated in Figure 3.2. As an essential property, it handles the entire communication across host boundaries by a separate communication process (CP). W.r.t. internal communication a CP behaves as any other server process. The main task of a CP is to accept service calls from local clients for remote servers, to send the parameters to the remote processor, to receive the answer, to accept tasks from and to send answers to remote clients. Apparently, this approach avoids the disadvantages of the first proposal. First of all, since CPs do not have to process tasks and since we do not assume them to be a bot-

tleneck, they can communicate synchronously, because waiting situations do not disturb system's performance behavior. On the other hand, synchronous communication simplifies its implementation drastically. Furthermore, we have a homogeneous view inside RCS, this means that communicating with a local or a remote server is the same, nRCS always only manipulates shared data structures which are then read by CP if necessary. After receiving a result message, CP puts the result into these data structures. Therefore, process size is smaller than in the first proposal because processes only have to implement local communication and they are not concerned with remote communication. Beside avoiding the disadvantages, this architecture has the following salient features. Because it does not have to take care about reliable transmission of the parameters, decomposition of data into small packages, composition of result messages, i.e. a lot of communication overhead is saved, the client process can continue its work much earlier. Furthermore, sophisticated protocols for remote communication do not burden the execution of a local task.

However, this architecture obviously induces a lot of context switches in a network of single processor machines, i.e. a shared-nothing architecture. Note, in oRCS, a task creation and result transmission across host boundaries may be executed without any context switch if both processes, client and server, are active. In our improved architecture, however, we need at least four (!) context switches (for example: Process1(client) -> CP1, CP2 -> Process5(server) -> CP2, CP1 -> Process1(client)). All these context switches are unavoidable, because on each processor there are always two processes involved in the communication. Furthermore, we actually have two local task invocations (client -> CP1, CP2 -> server) and two local result transmissions (server -> CP2, CP1 -> client) in addition to the two messages across the net, which may deteriorate response times. We will further investigate this long-winded behavior and measure its influence to the system in Section 4. Nevertheless, this observation does not hold on multiprocessor machines where CP can run on an individual processor which may avoid additional context switches between CP and application processes. For example, if in our previous example Process1 and CP1 on the client side as

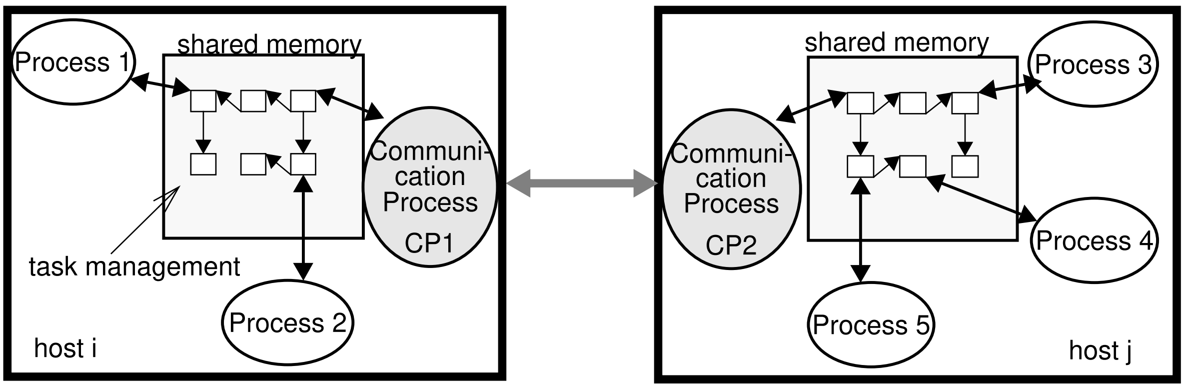

Figure 3.2: Architecture of the improved implementation of RCS (nRCS)

well as Process5 and CP2 on the server side are running on different processors, a service request of Process1 to Process5 would be executed without any context switch.

Note, in this context we transmit parameters from a source (client, server) to a destination (server, client, respectively). These parameters will not be used any more on the source site. Contrarily, distributed shared memory allows access to the same object by different processes on multiple hosts which is not required here. However, since message passing via the communication process is not visible to the application program and since parameters are allocated in local shared memory, the advantages [Me92] of easy programming, location independency, and persistence of parameters after a process termination apply to our implementation, too.

3.2.2 Task Management

Since we are realizing a single-process multi-threading approach, a server has to cope with scheduling and management of all tasks requiring a specific service. Therefore, RCS has to manage all tasks a server has to execute. Efficient management of appropriate data structures c.f. Figure 3.3) is crucial for the systems performance. Since explanations of observed performance characteristics (c.f. Section 4) will sometimes refer to handling of these data structures we will present them in the following:

- The root of our data structure is a server vector. Since each service in the system is identified by a unique number, every entry in this vector can address one service.
- A service may be realized by multiple processes which all have the same type, i.e. they correspond to the same program. A process_type_control_block (ptcb) represents this program. The ptcb contains information about the services, this process type implements. Initially, each entry of the server vector refers to the ptcb of the communication process. Apparently, ptcbs represent process types but not the processes themselves. This simplifies task administration drastically as will be seen later. On the other hand, this means that a service on a local host cannot be included in more than one process type. Hence, we cannot combine services

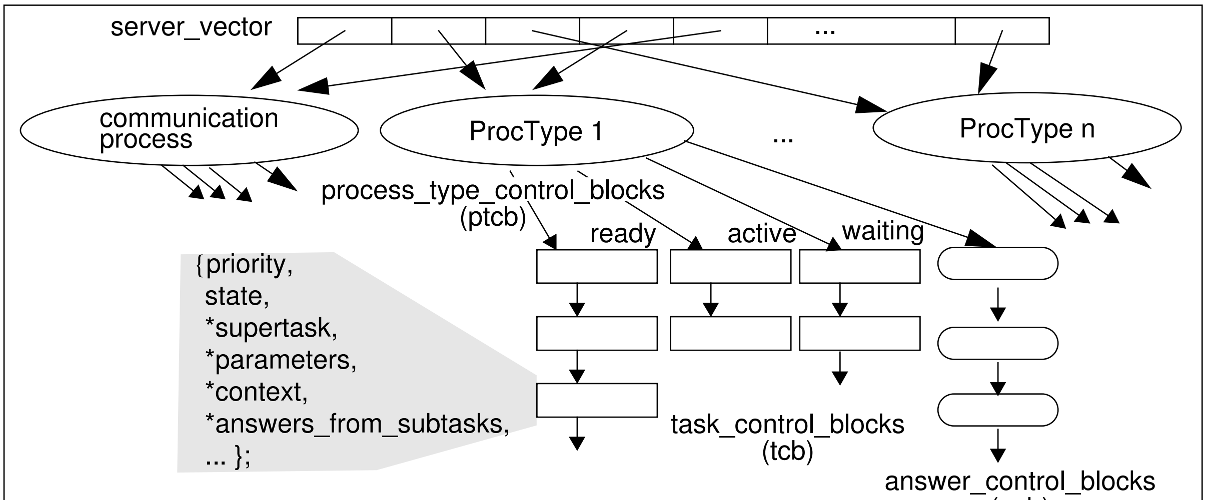

Figure 3.3: Data structures for task management

arbitrarily to server processes. But this restriction holds only for a single host, on different hosts we can integrate a service in different process types. Since it does not make sense to start a server process more than once on the same processor (because this causes unnecessary context switches) such a design decision is not a restriction in a distributed system.

- Task_control_blocks (tcb) represent tasks to be executed. They contain information about the task's parameters, its state, its supertask, a list of received answers from its subtasks and a pointer to its context which stores the actual state and local variables of a task after an interruption for its future continuation. tcbs are allocated in lists of the addressed process type. For each state a task may enter (ready, active, waiting), there is a separate list.
- Answer_control_blocks (acb) contain received answers. For performance reasons, a server will initially connect them to a list in the ptcb, and while scheduling tasks on the client side they will be assigned to their corresponding tasks. Obviously, this procedure causes more overhead on the client side than on the server side which could also reschedule tasks after connecting an answer to its supertask. But in real applications we expect that task execution on the server side takes longer than task preparation and answer processing on the client side. Furthermore, a client task often creates multiple subtasks and in order to get short response times we have to increase throughput for the server tasks. Therefore, the server should be able to process further tasks as soon as possible.

In order to save storage operations, tcbs and acbs are taken from and released to a preallocated pool. The same mechanism is used in some efficient lightweight process implementations [ALL89]. To be able to install multiple processes offering the same service and to allow migration of tasks between processes, these data structures have to be kept in shared memory. Furthermore, to enable shared use at low cost access to them has to be synchronized by the participating processes. Possible synchronization mechanisms are discussed in the following subsection.

3.2.3 Synchronization Mechanisms

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 as possible 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 performance:

- First, we used conventional semaphores offered by almost all operating systems. For the following three reasons, semaphores are only useful in order to lock infrequently accessed shared data with few access conflicts:
  - operating systems usually only allow for a restricted set of semaphores,
  - they are very expensive because they always cause operating system kernel calls,
  - they always cause context switches when the requested resource is locked.

Since separate locking of multiple small data units is too expensive, in this case, larger synchronization granules (e.g. a whole task list instead of single tcbs), which lock all data units at a time, have to be locked with a single semaphore. Otherwise, synchronization overhead would be too high. On the other hand, large synchronization granules restrict the possible amount of parallelism between the running processes. In contrast to these characteristics, the functions of RCS usually only need a short time to manipulate frequently accessed data. For these reasons, semaphores do not seem to be an appropriate synchronization mechanism on multiprocessor machines.

- In a second approach, we exchanged the semaphore operations by latches. Latches are simple atomic test-and-set operations to be issued at the application level. If a latch cannot get the requested resource, it loops in a busy waiting manner until the request is granted. If the number of processes does not outnumber the available processors, we expect that busy waiting does not waste resources, because no other process could use the processor in the meantime. Furthermore, the number of latches is not restricted and their execution and creation is much cheaper than that of semaphores. In our case, bus contention due to latch operations [An90] is not an as important problem as with fine-grained parallelism where relatively more work is done in critical regions.
- The third mechanism is a variation of the second one. Since indefinite busy waiting loops of test-and-set operations on a mono-processor host are senseless, we use latches, but we release the processor if a request cannot be satisfied immediately. Note, that in nRCS always at least two processes share the processor of a mono-processor machine (one application process and the CP).

4. Measurements and Results

So far, we have described the functionality and implementation of RCS. Furthermore, we have sketched the expected performance characteristics from an abstract point of view for different implementations. This section describes measurements and their results in order to get a quantitative evaluation. For this purpose, a comparison with rpc-facilities offered by operating systems allows a better assessment of the system. Before we present the results, we give a description of the kind of the measurements, the specific environment, the parameter settings, and the measurement procedure.

4.1 Measurement Basics

There are three main objectives for our measurements. First, we want to get real performance data of our system. Second, we want to compare nRCS to oRCS and to rpc-facilities of the operating systems used; these are the only suitable mechanisms for comparison we have.

Finally, the various kinds of measurements enable us to better understand the operation of nRCS and its performance characteristics.

As already sketched, two different hardware architectures have to be exploited. With the first architecture we investigate the 'shared allocation' case. For this purpose, we use a SEQUENT Symmetry (S27) [SEQ] which is a closely coupled multiprocessor system offering shared memory for communication and task management with 8 processors (6MHz Intel 80386) running DYNIX V3.0.18. Every processor has a cache of 64 kbytes. The processors and the shared memory are connected by a 64-bit system bus with a channel bandwidth of 80 megabytes per second. Furthermore, we examine 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 described here are executed in a conventional environment, i.e. neither the hardware environment nor the operating system are modified for our measurements. Of course, the machines were exclusively used by our measurements. Furthermore, we ensured that foreign network traffic was negligible by using a bridge physically separating the two participating workstations of our project from others.

We measured response times for synthetically generated tasks because this is the key measure of our application. In order to explain the observed effects, it turned out that further measurements using operating system facilities like time [SUN] or ptime [DYN] had to be accomplished to obtain more information about resource utilization. Their results are only described when necessary. Preceding measurements with gprof [SUN] guaranteed that there are no fatal errors inside the implementations of RCS.

The effect of parallelism was examined by varying two key parameters, the number of processes used and the number of simultaneously invoked tasks within a single process. In every measurement, repeated up to 10 times to be sure that really no external effects influenced the results, we executed 10000 tasks. Usually, task parameters and result values are empty, servers only accept tasks and immediately reply. Note, that even if the result value is empty the server still replies the task. Figure 4.1 shows simplified server and the client program skeletons.

The illustrated performance figures show average values. In all graphs presented, 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.

4.2 Measurement Results

The initial measurements compare all three available communication mechanisms (rpc, oRCS, nRCS) using the described allocation schemes (distributed, local, shared) for processes to processors (Sect. 4.2.1). Since oRCS and nRCS offer some features rpc-facilities do not have, we subsequently present a more detailed comparison of oRCS and nRCS again using all allocation schemes (Sect. 4.2.2). Finally, we investigate performance characteristics of

```
server_program()
{
  Init_Server(measure_server);
  while (1) {
    Accept_Task(&task_parameters);
    Reply(result_parameters);
  }
}
```

```
client_program(no_of_tasks, parallelism)
{
  no_of_invoked_tasks = 0;
  gettime(starttime);
  for(i=1;(i<=parallelism)&&(i<=no_of_tasks); i++){
    Remote_Service_Invocation(measure_server,
                                              task_parameters);
    no_of_invoked_tasks++;
  }
  while (no_of_invoked_tasks<no_of_tasks){
       Get_Service_Result(&result_parameters);
       Remote_Service_Invocation(measure_server,
                                              task_parameters);
    no_of_invoked_tasks++;
  }
  for(i=1;(i<=parallelism); i++){
    Get_Service_Result(&result_parameters);
  }
  gettime(endtime);
  avg_task_execution_time=(endtime-starttime)/no_of_tasks;
}
```

Figure 4.1: Structure of Programs implementing Clients and Servers

nRCS with shared allocation in further detail considering features oRCS and rpc do not offer (Sect. 4.2.3).

4.2.1 Comparison of rpc, oRCS, and nRCS

The measurements started with a comparison of rpc, oRCS and different implementations of nRCS using semaphores, busy waiting latches, and latches causing context switches. In the distributed and in the local case, busy waiting latches have not been used (c.f. Section 3). Due to the restrictions of rpc-facilities which do not allow concurrent invocation of tasks we are forced to vary the number of client processes in order to achieve parallelism. Since we use a very small number of client processes only (1 to 8), overhead due to task administration is not significant. Of course, this would change with higher degrees of parallelism, automatically leading to increased response times. In order to compare the measurements, the implementations using nRCS and oRCS perform synchronous task invocations, too. In the distributed allocation, only the server process on one side and the client processes on the other side are distributed, i.e. all client processes are running on a single processor. The results are illustrated in Figure 4.2a for the shared allocation on a multiprocessor system, in Figure 4.2b for the local allocation, and in Figure 4.2c for the distributed allocation. Note, due to the variation in processor power, the measured times on the multiprocessor and on the workstations cannot be compared directly.

The measurements with shared allocation yield several remarkable results. First of all, as already mentioned, semaphores (Fig. 4.2a, nRCS semaphores) require kernel calls and cause unnecessary context switches in waiting situations such that this implementation behaves worst. Second, as expected, the implementation with busy waiting latches (Fig. 4.2a, nRCS latches) outperforms all other implementations of nRCS. Finally, response times of nRCS are better than those of the rpc-facility.

The larger response times in the case of a single client using rpc and oRCS arise from context switches in the server process after sending the answer because it has to wait for the next task. With an increasing degree of parallelism the server requests keep the process running which more and more avoids voluntary context switches. In nRCS, increased response times in the single-client case are caused by the overhead in the client (c.f. Section 3.2.2); by increasing

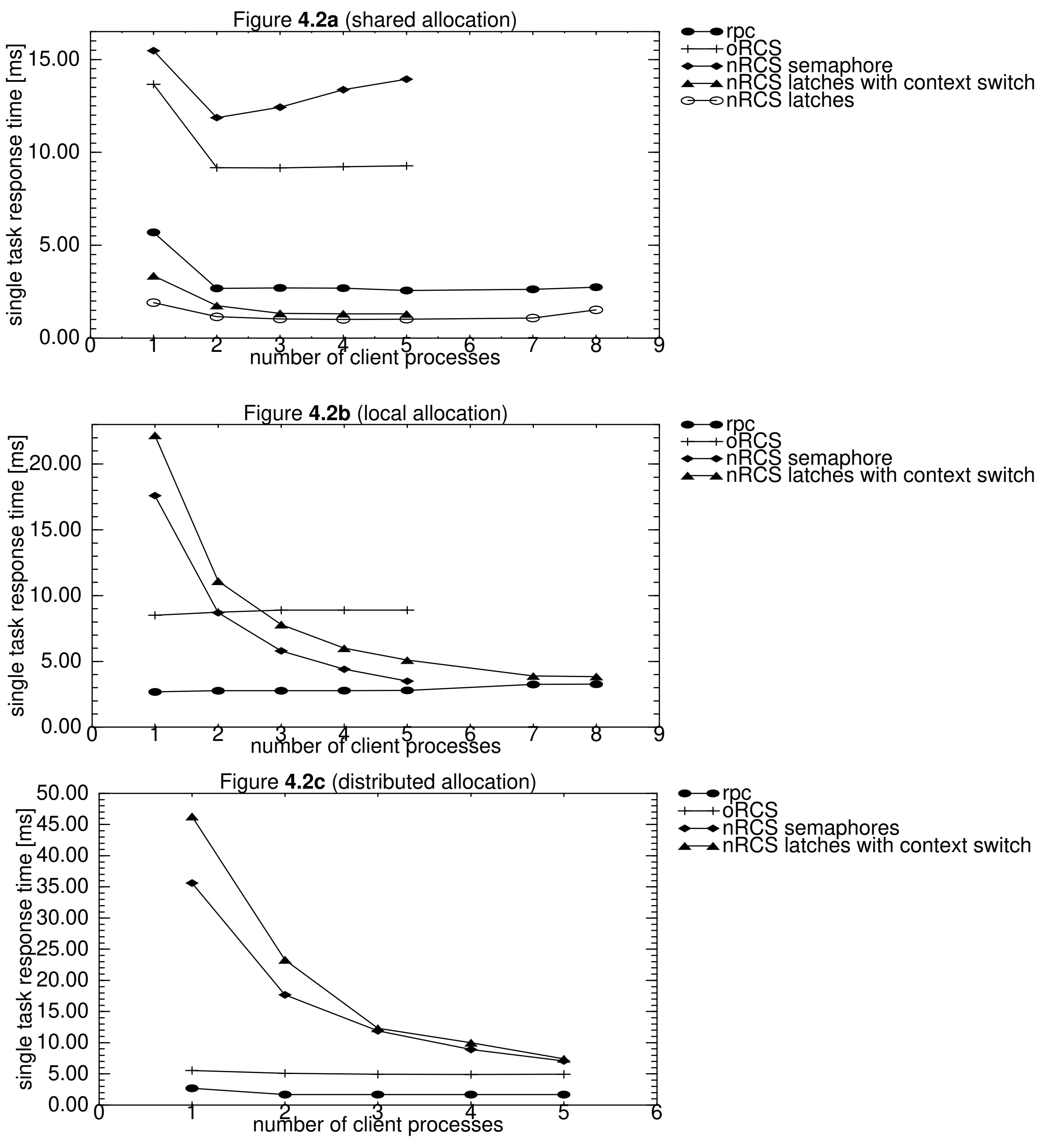

Figure 4.2: Comparison of rpc, oRCS, and nRCS (1 server process, n client processes)

the number of client processes this overhead is distributed over more processes thereby reducing the average response time.

In the local allocation case (Fig. 4.2b), as expected, in the rpc and oRCS implementation response times do not decrease with increased client parallelism because the processor is always running a client or a server process even in the case of a single client process. But the results for nRCS are surprising, because we did not expect the bad response time observed for a single client and, furthermore, the drastically improving response times with higher degrees of client parallelism. A quick look at our measurement environment does not deliver any convincing explanation.

Further investigations turned out that the SUN OS-signals for waking up sleeping processes were responsible for such a bad performance. After receiving a signal processes cannot be activated immediately because the SUN OS previously has to reschedule them. With an increasing degree of parallelism, this problem alleviates because in this case the other client processes can use the time to the next scheduling. In order to validate these arguments, we executed the same measurement on the multiprocessor machine, running the DYNIX operating system, with only a single processor switched on. The results of these measurements, illustrated in Table 4.1, confirmed the expected behavior. This means that if task execution times increase due to real task processing, which is omitted in our measurements response times, they do not enlarge in the nRCS case with few client processes because the processing time then can exploit the waiting times of the processes. Contrarily, in the rpc and the oRCS case they increase even

#of clients	response time in [ms]
1	14.9
2	15.2
3	16.1
4	16.5
5	18.0

Table 4.1: nRCS, single processor, DYNIX operating system

in the case of few client processes, too. Therefore, in this case response times of nRCS on one and rpc and oRCS on the other hand approximate in the same way as observed for multiple client processes which degrade the disadvantage of nRCS in these situations. As already observed in the local allocation case in the distributed allocation case (Figure 4.2c), the rpc-mechanism again outperforms both RCS implementations. In comparison to the local allocation, the overhead for double task creation and result transmission inside RCS (c.f. Section 3.2.2) across host boundaries causes about double response times for both nRCS implementations. Due to the behavior of signals, the vast number of context switches with the communication manager does not effect the results noticeably.

Finally, in contrast to the shared allocation case, we see that the nRCS implementation with semaphores behaves better than the implementation with latches which can be explained by the implementations of the latches. Since there is no user call available to release the processor and we do not want to change the operating system, we cause the process to sleep for at least one time slice (10 ms on a SUN) which in contrast to semaphores causes longer waiting times. This effect diminishes if there are multiple processes on the same processor, because other processes may use the waiting time.

4.2.2 Further comparison of oRCS and nRCS

Since both, oRCS and nRCS, are capable of asynchronous task invocation and of integrating clients and services into a single process, we evaluated and compared performance characteristics for both implementations using these features. Again, we analyzed all three process allocation schemes.

Parallelism

Different degrees of parallel task invocations have been investigated with the following measurements consisting of only one client and one server process. In the shared case, nRCS with latches leads to much better results than oRCS and nRCS with semaphores (c.f. Fig. 4.3a). Since waiting situations do not occur in this environment, both implementations with latches do not differ except in the case of sequential execution. Due to task management overhead response time increases linearly. It exceeds response time of sequential execution at about 200 in nRCS and at about 100 parallel tasks in oRCS.

Figure 4.3b shows the results on the SUN workstations. First of all, because of less context switches oRCS behaves better in the case of distributed allocation than in the case of local allocation. Contrarily, nRCS produces better results in the local case because context switches can be saved. In nRCS with distributed allocation, an increasing number of parallel task invocations leads to a decreasing number of context switches between the communication processes and the client and server process, because multiple tasks or result may be transferred or processed during a single process activation. Therefore, the number of waiting situations due to necessary process rescheduling (c.f. 4.2.1) decreases and the difference for the response times of the local and the distributed nRCS case increases. Furthermore, nRCS even outperforms oRCS in both cases with higher degrees of parallel task invocations.

Embedded Task Invocation

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 are no context switches necessary, we used the implementation with latches keeping the process running. The same environment was used on a SUN workstation and on the SEQUENT machine, hence, the performance behavior is similar in principal (c.f. Fig. 4.4), but since the SUN processor is much more efficient, it gains shorter response times. Separate measurements compared the situa-

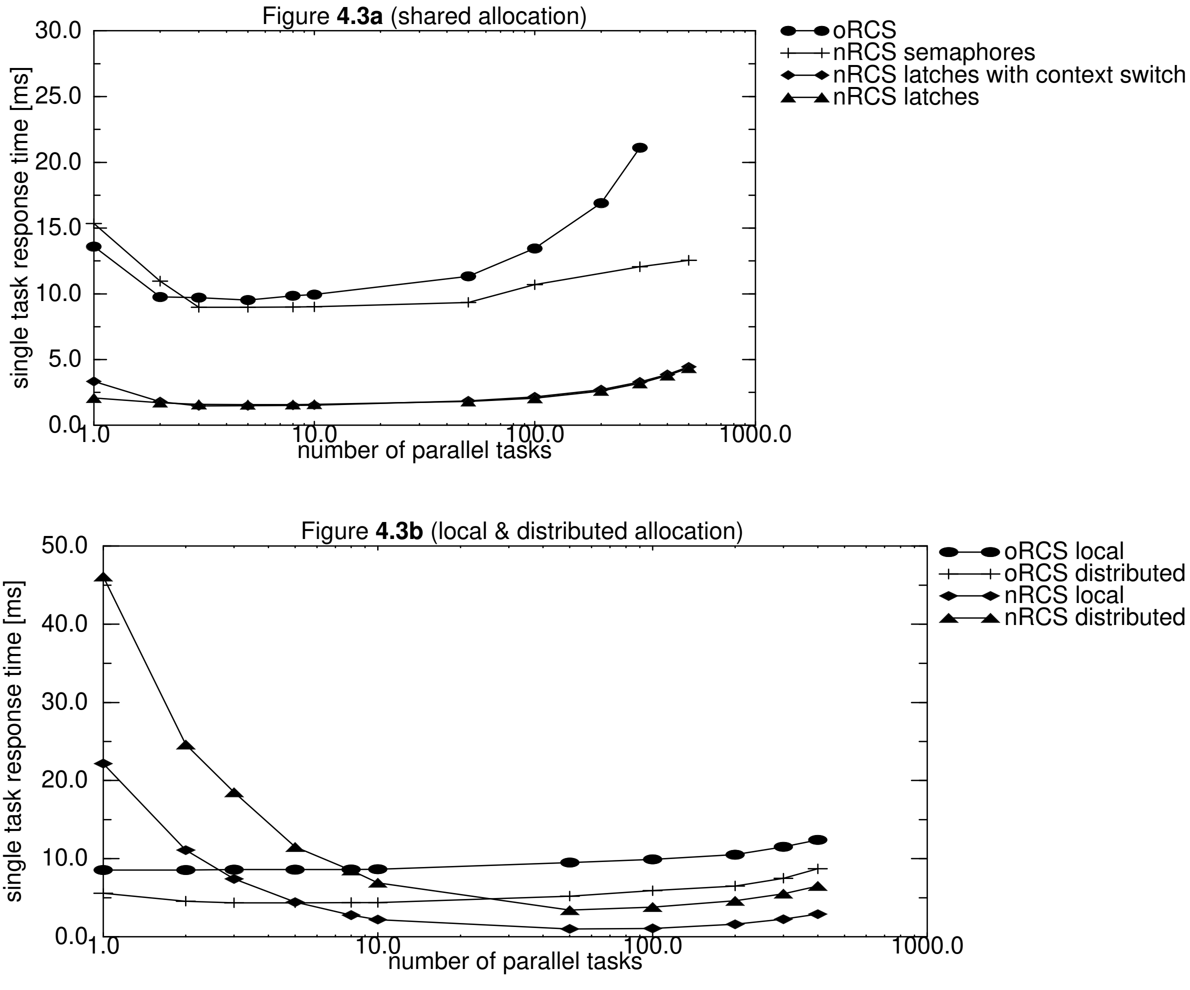

Figure 4.3: Comparison of oRCS and nRCS (1 server process, 1 client process)

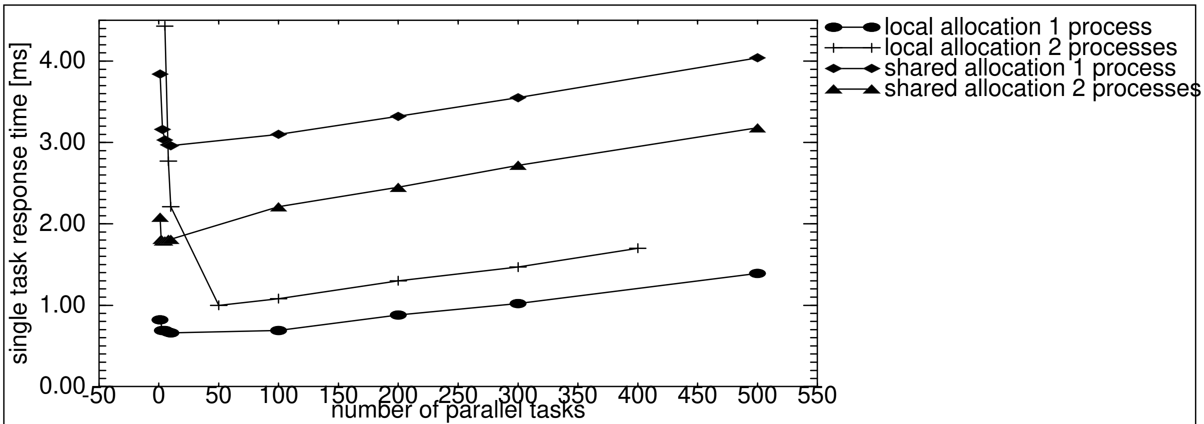

Figure 4.4: Evaluation of nRCS (embedded task invocation)

tion where client and server were allocated to two processes. The corresponding results are summarized in Figure 4.4. As expected, the implementation with separate processes behaves better than the one-process implementation on the SEQUENT because the load can be distributed over two processors. On the other side, it is clear that the one-process implementation leads to better results on the SUN workstation because there are no context switches after task invocations. In Figure 4.4, we skipped the peak value for task execution at very low levels of parallelism with two processes in the case of local allocation (see for example Fig. 4.2b).

4.2.3 Detailed analysis of nRCS in the case of shared allocation

Since nRCS is the only system providing all required capabilities, and since it turned out to be the best system in our environment with shared allocation, we investigated the performance behavior of nRCS for this implementation in further detail.

Number of Processes

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. For this purpose, we executed the previous measurement with different degrees of parallel tasks on the multiprocessor architecture with one, three, five, and ten client and server processes, respectively. Measurements with more processes always produced worse response times as compared to the case of five clients and servers. Generally, if the number of processes exceeds the number of available processors (in our case 8) context switches between the participating processes lead to worse response times. The important results are shown in Figure 4.5. First of all, we see that it is not useful to add multiple server processes to a single client because due to its overhead the client is the bottleneck. Adding more clients yields better performance for lower levels of parallelism in task creation because these clients can work in parallel on their own processors. With a higher degree of parallelism this performance degrades because of the task administration overhead in the server process. Note, every process produces the indicated number of parallel tasks. In the nRCS evaluation of Figure 4.5, the best and the worst response times never differ by more than a factor of about 2.

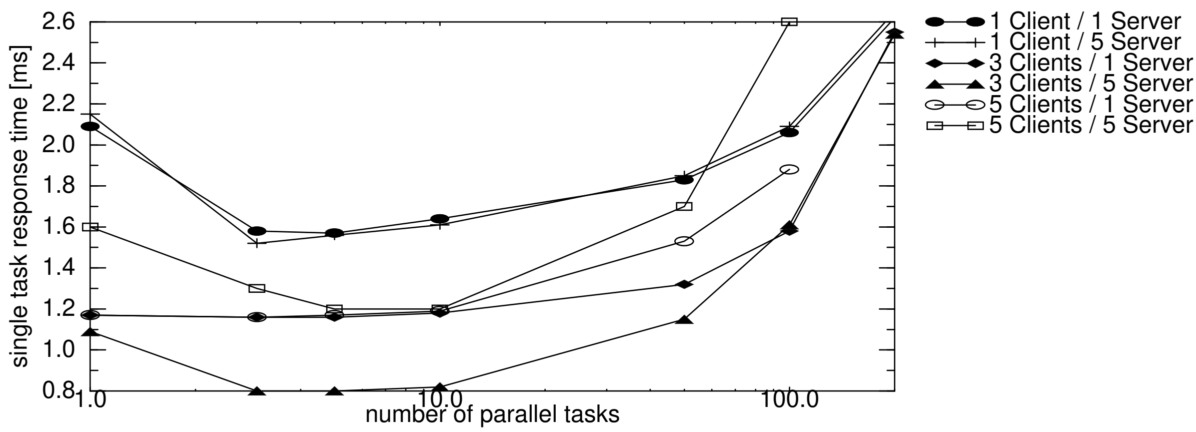

Figure 4.5 Evaluation of nRCS (number of client and server processes)

Task Execution Times

Finally, a simple measurement investigates the impact of increasing task execution times which is done by simply enlarging task administration overhead by a rising number of task switches(c.f. Fig. 4.6). Increasing execution times lower parallelism between client and server in our measurement because now the client is waiting for results while the server is running. This means, as intended with our implementation, the administration overhead on the client side is no longer the bottleneck. Furthermore, these measurements demonstrate the overhead caused by task administration for increasing degrees of parallelism, where the observed response times only depend on this overhead and the execution time of the tasks becomes less important.

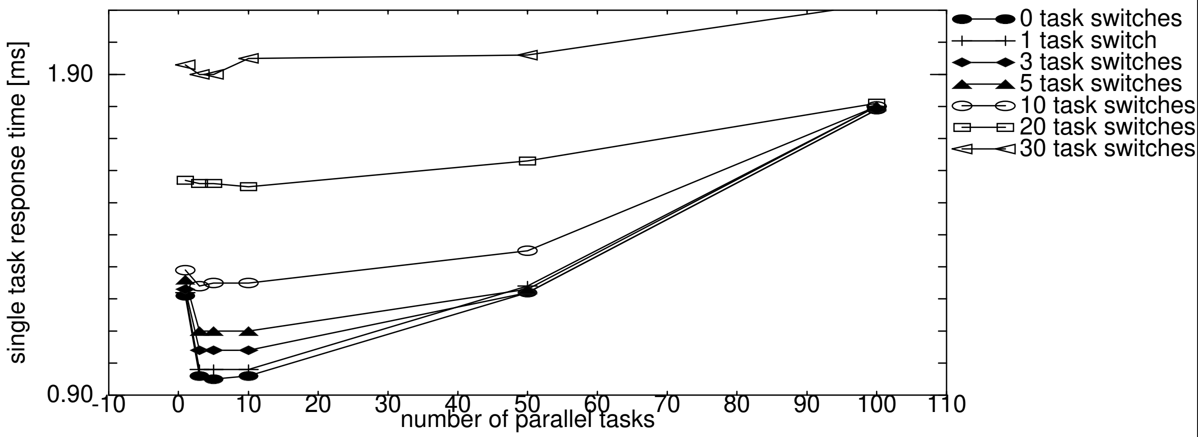

Figure 4.6: Evaluation of nRCS (task execution times)

4.2.4 Summary

Summarizing the results of the described measurements, nRCS fulfills the requirements of our application. Especially in the case of shared allocation on the multiprocessor system, it achieves better response times than the older implementation and even than the rpc-facility which is offered by the operating system and which provides a much simpler interface. nRCS enables us to use all features to utilize parallelism as described in Section 3. Optimal response times were achieved at a relatively small number of 5 to 10 parallel tasks, but up to about 200 parallel client requests response times are satisfactory, which allows to adequately support the kind of parallelism exploited in our system (c.f. Section 2). We observed different performance behavior for various numbers of client and server processes, but because of the simple scenario (except one measurement no real execution time and no waiting situations in the server) a general statement about performance characteristics in these cases is not possible. This effect has to be explored in more detail in a more realistic application consisting of multiple levels of client/server-relationships. Finally, different task execution times occurring in real applications may lead to more sophisticated results.

5. Conclusions

In order to optimize mechanisms to support parallel execution of queries on complex objects in our NDBMS PRIMA we introduced the Remote Cooperation System 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. On the client side, it offers functions to invoke a service asynchronously and a separate function to receive results. On the server side, results may be transmitted in multiple parts, which allows pipeline parallelism. RCS supports a multi-threading implementation of servers where multiple servers may be allocated to one process. Moreover, to enable parallelism within a particular server, multiple processes may be used to statically replicate such a server.

Two implementations using different communication mechanisms have been presented. The initial implementation (oRCS) only uses stream sockets for communication whereas the actual implementation (nRCS) uses shared memory where possible and a specialized process to communicate across host boundaries. In order to control access to shared data different synchronization mechanisms have been discussed. nRCS is particularly designed for closely coupled multiprocessor systems which are best suitable for NDBMSs due to the highly meshed data structures in complex objects. Data partitioning as used in loosely-coupled distributed DBMS, turned out to be not recommendable.

In order to get an accurate understanding of the performance characteristics of these implementations, a couple of measurements were carried out on a network of SUN SPARC workstations and on a SEQUENT Symmetry S27. oRCS behaves better in the distributed system. When the parallelism is increased, nRCS's performance penalty is alleviated to a certain extent. Contrarily, as intended nRCS leads to better results on the multiprocessor system where latches not causing any context switches turned out to be the best synchronization mechanism. In the distributed case latches releasing the processor if required resources are locked lead to the best results with nRCS.

A comparison with the remote procedure facilities on both systems persuades us that the nRCS implementation is a suitable basic communication system in order to efficiently realize our PRIMA system. The optimal degree of parallelism within a server is about 10 simultaneous tasks, but the measurements showed that only for more than about 200 parallel tasks administration overhead and synchronization conflicts lead to remarkably increasing response times. These characteristics correspond with our hypotheses for our PRIMA implementation, where coarse grained parallelism not producing that many tasks in parallel within a single service should lead to effective usage of resources and good performance. On the other hand with increasing parallelism most of the time is spent in task administration. Therefore, more adjusted data structures in this case can yield better response times.

These results of the described measurements will help to understand the performance characteristics of the PRIMA system which may be obtained by further measurements. In order to do this task monitoring facilities have been integrated into RCS [Hü92]. These tools allow a detailed analysis of specific algorithms as well as of load distribution in the system. In the future, the influence of different scheduling strategies has to be explored in more detail. Moreover, we try to include the communication process into application processes which especially seems to make sense, if there is only a single application process running on a host. Then, we can save context switches between the communication process and application processes to improve the response times with nRCS in the distributed system.

References

An90 Anderson, T.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors, IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 1, pp. 6-16, January 1990

ALL89 Anderson, T., Lazowska, E., Levy, H.: The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors, IEEE Trans. on Computers, Vol. 38, No. 12, pp. 1631-1644, December 1989

BR89 Bhargava, B., Riedl, J.: The Raid Distributed Database System, IEEE Trans. on Software Engineering, Vol. 15, No. 6, pp. 726-736, June 1989

DYN DYNIX 3.0.18 Manual reference pages

FHM93 Fang, D., Hammer, H., McLeod, D.: A Mechanism and Experimental System for Function-Base Sharing in Federated Databases, in Hsiao et al. (Eds): Interoperable Database Systems (DS-5), IFIP, pp. 239-253, Elsevier Science Publishers, 1993

GGS93 Gesmann, M., Grasnickel, A., Schöning, H.: A Remote Cooperation System Supporting Interoperability in Heterogeneous Environments, Int. Workshop on Research Issues on Data Engineering RIDE-IMS, Vienna, 1993, pp. 152-160

Hü92 Hübel, C., et al.: Using PRIMA-DBMS as a Testbed for Parallel Complex-Object Processing, in: Proc. Workshop on Research Issues on Data Engineering: Trans. and Query Processing, Tempe, As., 1992, pp. 38-45

HMMS87 Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. 13th Int. Conf. on Very Large Data Bases 1987, Brighton, United Kingdom, 1987, pp. 433-442

HSS89 Härder, T., Schöning, H., Sikeler, A.: Evaluation of Hardware Architectures for Parallel Execution of Complex Database Operations, in: Proc. 3rd Annual Parallel Processing Symposium, Fullerton, CA, 1989, pp. 564-578

KD93 Keßler, U., Dadam, P.: Benutzergesteuerte, flexible Speicherungsstrukturen für komplexe Objekte, (in German), Stucky, W., Oberwies, A. (Eds), Proc. Datenbanksysteme in Büro, Technik und Wissenschaft, Springer Verlag, pp. 206-225, KGM91 Keller, T., Graefe, G., Maier, D.: Efficient Assembly of Complex Objects, Proc. ADM SIGMOD Conf., Denver, CO, May 1991

Me92 Mehl, H.: Distributed Shared Memory - A Survey -, Report No. SFB124-33/92, University of Kaiserslautern, 1992

Mi88 Mitschang, B.: A Molecule-Atom-Datamodel for Enhanced Applications, (in German), Informatik-Fachberichte 195, Springer Verlag, 1988

Mi92 Mitschang B., et al.: SQL/XNF - Processing Composite Objects as Abstractions over Relational Data, Proc. 9th Int. Conf. on Data Engineering, Vienna, April 1993, pp. 272-282

MB91 Mafla, E., Bhargava, B.: Communication Facilities for Distributed Transaction-Processing Systems, IEEE Computers, pp. 61-66, August 1991

MH92 Manola, F., et al: Distributed Object Management, International Journal of Intelligent and Cooperative Information Systems, Vol. 1, No. 1, pp. 5-42, 1992

Hü92 Hübel, C., et al: Using PRIMA-DBMS as a Testbed for Parallel Complex-Object Processing, in: Proc. Workshop on Research Issues on Data Engineering: Trans. and Query Processing, Tempe, Az., 1992, pp. 38-45.

Pa87 Paul, H.-B., et al: Architecture and Implementation of the Darmstadt Database Kernel System. ACM-SIGMOD, Proc. Int. Conf. on Management of Data, San Francisco, pp. 196-207, 1987

PA86 Pistor, P., Andersen, F.: Designing a Generalized NF2 Model with an SQL-Type Language Interface, Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto,, pp. 278-285, 1986

SEQ Symmetry Technical Summary, Sequent Computer Systems, Inc., 1987

Sch93 Schöning, H.: Query-Processing in Complex-Object Database Systems, (in German), Deutscher Universitäts-Verlag, 1993

SS89 Schöning, H., Sikeler, A.: Cluster Mechanisms Supporting the Dynamic Construction of Complex Objects, Proc. 3rd Int. Conf. on Foundations of Data Organization and Algorithms, Paris, Lecture Notes in Computer Science 367, Springer-Verlag, pp. 31-46, 1989

SUN SUN OS 4.1.3 Manual reference pages

SW93 Schek, H.-J., Wolf, A.: Cooperation between Autonomous Operation Services and Object Database Systems in a Heterogeneous Environment, in Hsiao et al. (Eds): Interoperable Database Systems (DS-5), IFIP, pp. 239-253, Elsevier Science Publishers, 1993

Te93 Teeuw, W.: Parallel Management of Complex Objects, The Design and Implementation of a Complex Object Server for Amoeba, CIP-DATA Koninklijke Bibliotheek, Den Haag, 1993