ESTEDI

European Spatio-Temporal Data Infrastructure
For High-Performance Computing

IST Project IST-1999-11009

Commission of the European Communities

VLDB Workshop
Supercomputing Databases

Roma, Italy 14. September 2001

Pontificia Università Urbaniana

at the VLDB 2001 Conference

Parallel Query Support for Multidimensional Data

Dipl.-Inf. Karl Hahn

(FORWISS, TU-München)

hahnk@forwiss.tu-muenchen.de

Contents

1 Introduction

1.1 Motivation

A common characteristic of multidimensional data is its enormous size. Analysing such large data requires a lot of resources, especially CPU power and disc space. In order to speed-up simulations High Performance Computing Centres use parallel computers, e.g. symmetric multiprocessing machines (SMP) or workstation clusters.

RasDaMan is a database system for multidimensional data but was not originally designed to use more than one processor to execute queries. In consideration of the fact that most queries on multidimensional data are computational queries [Rit99], i.e. the response time of the queries depends on the CPU and not on the I/O (CPU-bound vs. I/O-bound), it is obvious that this is a bottleneck of the RasDaMan system.

In this paper we will describe the implementation of parallel query processing in RasDaMan using the Message Passing Interface (MPI) as the communication interface. We concentrate on SMP computers (computers with several CPUs) because this basically is the most frequently used parallel hardware architecture of the HPC partners. Nevertheless, the demonstrated concepts can also be used with workstation clusters as MPI works on a wide range of hardware, e.g. parallel supercomputers or Linux clusters (”Beowulf” machines).

The rest of the paper is organised as follows: in chapter 1.2 we will shortly present an overview of parallelism in database systems. In chapter 2 we will focus on the implementation of parallel query processing in RasDaMan, first showing the original query processing then explaining the differences to a parallel query execution. The Message Passing Interface (MPI) is the communication protocol used for inter-process communication. We will give an explanation of the communication primitives and the messages in chapter 3. Chapter 4 summarises the achievements and gives topics for the future work.

1.2 Overview Parallelism

In the scientific literature one can find different possibilities for parallel query execution in database systems. We can distinguish between data and pipeline parallelism on the one hand, and inter-operator and intra-operator parallelism on the other hand:

- Data parallelism: the same operation is done by several processes, each process using a sub-partition of the data. This kind of parallelism is frequently used in relational database systems, e.g. a full table scan can be performed by several processes by partitioning the table.
- Pipeline parallelism: one process does computations on a data stream while another process is still producing the data stream. E.g., in a relational database system, a sort operation produces output which can be used by the next operation while the sorting process is still in progress.

- Inter-operator parallelism: different operations on the data are processed by different processes. E.g., in relational databases, the left subtree and the right subtree of a join operator can be processed by different processes.
- Intra-operator parallelism: an operation on data is processed by more than one process. This is mostly a kind of data parallelism. E.g., data partitioning for a scan operator in relational database systems.

An excellent description of parallel query processing in relational database systems can be found in [DG92].

In this paper we describe the implementation of data parallelism for RasDaMan. In chapter 2 we will see that the implemented concept includes both, inter- and intra-operation parallelism, because different parts of the query are designated to different processes (inter-operation p.) but the (same) operations which represent the expensive part of the query are distributed to different processes for each multidimensional data object (intra-operation data p.).

2 Parallel Query Processing

In order to present parallel query processing, we will first briefly explain query processing in the original RasDaMan server (without parallel query processing) in chapter 2.1. This will best show the changes that had to be made to allow for parallel execution.

2.1 Original Query Processing in RasDaMan

RasDaMan uses a query language which is derived from standard SQL, called RasQL. The simplified structure of such a RasQL query is

```
SELECT <operation on data>
FROM collection 1, ..., collection n
WHERE <condition on data>
```

Internally, the RasDaMan server builds up a query tree to process the query (Figure 1). The query execution follows the open-next-close protocol (iterator concept) that is well known in database technology. First, the method `open()` is invoked on the root node α. In a post-order traversal, the method invocation is propagated through the query tree while initialising stream inputs, collection iterators, and other resources. Then, method `next()` is invoked repeatedly on the root node which again is propagated in a post-order traversal through the complete tree. Each time the method completes, this bottom-up process returns one element of the result collection. It indicates the end of the evaluation process through an exception. At the end, method `close()` is called to clean up resources allocated during execution.

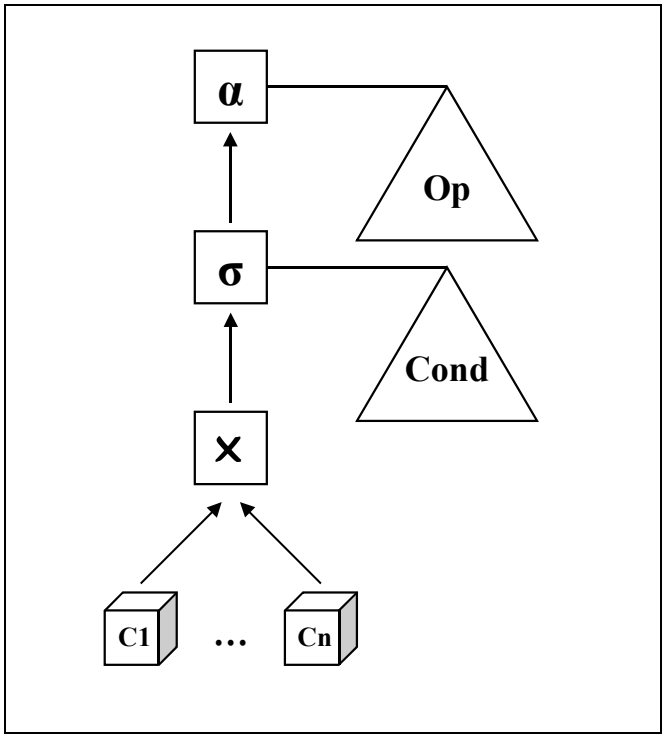

Figure 1: Internal query tree for a RasQL query

The iterators using the open-next-close protocol in Figure 1 are

- cross product x, representing the FROM clause of the RasQL statement. It delivers the cross product of all multidimensional objects of all collections C1 x C2 x ... x Cn. The number of returned elements is |C1| x |C2| x ... x |Cn|.
- selection σ, representing the WHERE condition of the RasQL statement. The condition tree consists of multidimensional operations (e.g. geometric operations, induced operation, etc.). A `next()` returns the next multidimensional data object for which the condition tree evaluates true.
- application α, representing the SELECT operation of the RasQL statement. The operation tree executes multidimensional operations on the resulting multidimensional data. These are e.g. geographical operations like trimming or section, or aggregation operations like average.

It should be pointed out that the multidimensional data is not loaded at the bottom of the query tree (only identifiers for the data are returned by x) but demand-driven during the evaluation of the condition tree and the operation tree. Therefore, these operations of the query tree (α and σ) are the most expensive sections.

2.2 Distributing the Computational Work to Processes

Distributed processing of RasQL queries requires different processes and communication between them, e.g. to exchange requests or intermediate results. In order to avoid performance problems while evaluating a query the

processes do not fork during query execution but at start-up time of the RasDaMan server (see Figure 2). I.e., at start-up time of the RasDaMan server we create several processes which reside in memory waiting for requests. As we will see, we run 2 processes for administration tasks and an arbitrary number of processes for the computational work. In order to utilise CPU resources on the one hand, and avoid unnecessary swapping of processes on the other hand, we recommend n+2 processes with n being the number of processors.

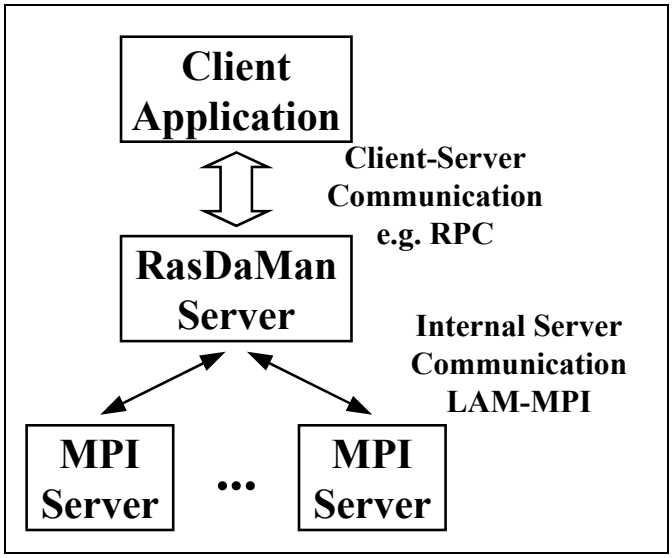

Figure 2: New parallel architecture of the RasDaMan server

The parallel RasDaMan server divides 3 classes of processes, i.e. each process class reflects a part of the overall query tree that can be executed independently from other processes:

- RasDaMan master: this process holds the server-client communication (RPC[1] or HTTP[2]). It distributes the queries and the work to all other (internal) processes. The internal processes communicate with the master process using the MPI (Message Passing Interface, see chapter 3.1) protocol.
- internal tuple server: this process manages the tuples of multidimensional data objects. This process is required to assure a central administration of the multidimensional data objects for all processes.
- internal worker processes: a number of processes which do the computational work. Receiving a tuple from the tuple server (invocation of next) these processes evaluate the condition tree and the operation tree on this tuple.

Figure 3 shows the query tree which was prepared for parallel query processing by the parallelizer module which is invoked after the optimisation process. The original query tree is divided to three parts which are processed by different process classes. The parts are connected with send and receive nodes. These nodes represent the transmission of intermediate results using the MPI protocol.

[1] RPC: Remote Procedure Call

[2] HTTP: Hypertext Transfer Protocol

We will now explain the specific task of the three process classes, and what part of the query tree they have to compute during query evaluation.

2.2.1 RasDaMan Master Process

This process holds the client-server communication and manages load-balancing during query execution. The following requests are distributed to the internal MPI processes:

- Start of RasDaMan server: the master process starts several internal MPI processes[3]. Parameters (e.g. from the command line) are given to these processes.
- Stop of RasDaMan server (shutdown): the master process sends a shutdown request to the MPI processes and exits.
- Open database: a message with the required parameters is sent to the processes. Parameters include the calling client, the user name, the database name, etc.
- Close database
- Execution of a query: The task of the master process is to distribute the query to the internal processes, collect all results from the internal worker processes, and transmit these results to the client application (e.g. RView) via the RPC or the HTTP protocol. The working processes (1:n communication) and the pending results are internally stored. Each time a worker process returns a valid result, another next request is sent to this process. A close request is only sent to the internal processes if the master received a NULL value (no more results) from all communication partner processes. In Figure 3 we can see that the query tree for this process only consists of a receive node.

2.2.2 Internal Tuple Server

The tuple server administers the multidimensional data objects. In order to avoid uneven computational loads on the different worker processes the multidimensional objects are distributed to the workers on demand by this process, not in advance at query initialisation. Receiving a next request the tuple server process accesses the underlying relational database system (only object identifiers are read not the whole objects), and sends the next tuple of object IDs to the calling process.

[3] we recommend n+2 processes with n being the number of processors (see chapter 2.2)

2.2.3 Internal Worker Processes

We typically run n worker processes with n being the number of the CPUs on the computer where RasDaMan is started. In contrast to the master process and the tuple server which are designed for management tasks, the worker processes do the actual computational work.

We decided not to compute the application node and the selection node of the query tree on different processes because this would avoid the usage of transient parts of the multidimensional data, i.e. memory-cached objects of the data read from the relational database system. In Figure 3 you can see that both, the selection tree and the application tree are evaluated by the worker processes. A worker process retrieving a next request gets the next tuple of multidimensional data from the tuple server and evaluates the selection and application on it.

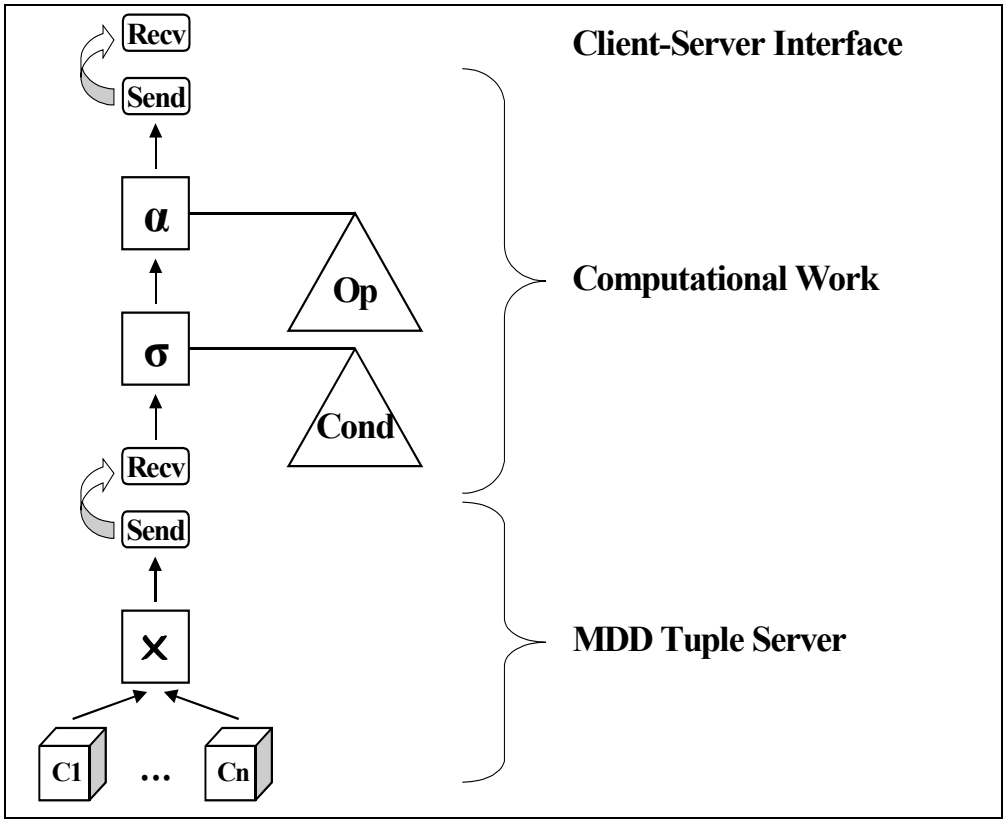

Figure 3: Query Tree after execution of the parallelizer

2.2.4 Example of Internal Communication during Query Execution

We will now give an overview of the complete procedure of parallel query processing. The master process has already started all processes at start-up time (e.g. on a 2-processor computer we typically have one master process, one tuple server process and 2 worker processes). All processes have opened the database connection. Receiving a

query from the client the master process sends a query request to the internal processes (with the query string as parameter). Next, all processes parse the query and build up the optimised query tree[4]. The parallelizer module, invoked after the optimisation process on all processing nodes, extends the query tree with send and receive nodes in order to allow inter-process communication, and assigns each process the relevant part of the overall query tree. Then, the execution of the query is started.

Figure 4 shows the detailed inter-process communication during query execution. The master process first identifies the worker processes and sends a `open()` and `next()` request to these processes. The worker processes passes the `open()` and `next()` on to the tuple server which delivers a tuple of multidimensional objects to the worker processes. If the condition tree evaluates true the worker process sends the result to the master, otherwise the next tuple is requested from the tuple server (see (2) in Figure 4: we have no result here). Whenever the master process receives a valid result it sends another next request to the reporting process. The master process must not make any assumptions about the order of the incoming results, as next requests can overtake each other (see (1): worker 2 has sent a result before worker 1 although worker 1 received the next request first). A close request can only be sent if all worker processes sent a NULL result (no more results available, see (3) in Figure 4). Sending a close request before the last working process returned NULL could lead to a re-initialisation of the input stream (at the tuple server) which could deliver the first tuple to a still working process again.

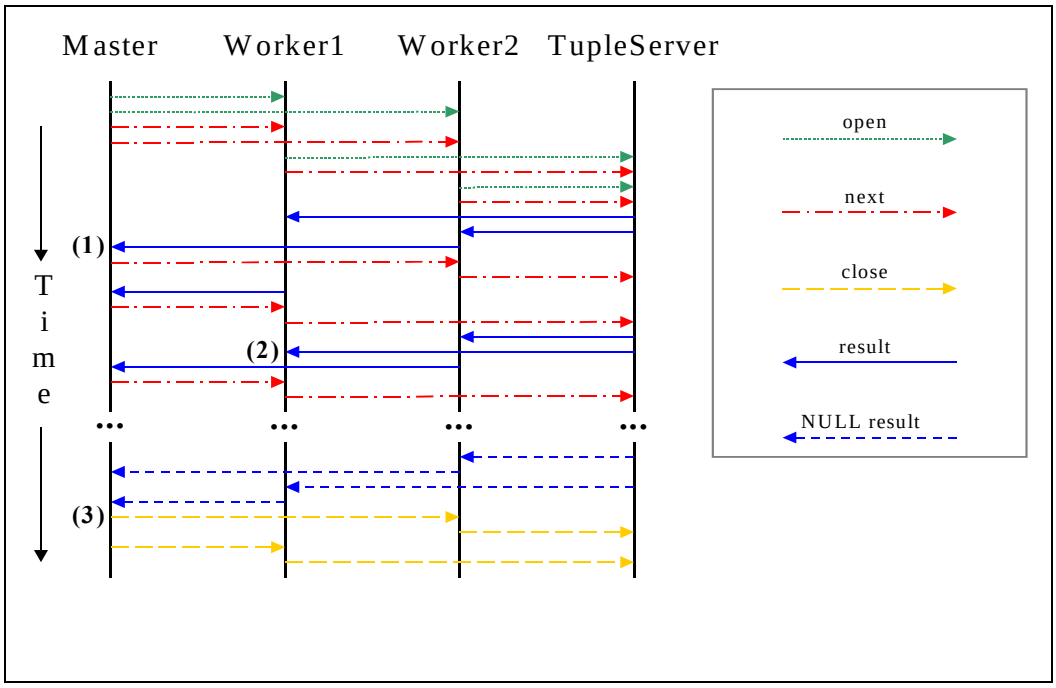

Figure 4: Internal communication during query processing

[4] As the query tree is a highly dynamic structure in main memory, it is more performant to send the query as a string and parse it on all processing nodes than to analyse the tree structure, send it as a stream, and create a query tree from that stream

3 Inter-Process Communication

In chapter 2 we explained the different process classes of the parallel RasDaMan server and their intercommunication. In this chapter we will describe some more technical aspects of the communication protocol used (MPI), the integration of an MPI implementation in the RasDaMan kernel and the communication primitives of MPI used. Further, we will explain the content of the messages, especially the transmission of intermediate results. In order to transmit intermediate results highly dynamic structures of the programming language had to be analysed and assembled to a string which can be sent with MPI.

3.1 Message Passing Interface

The parallel RasDaMan server utilises the Message Passing Interface standard (information about MPI can be found e.g. at [MPI] or in [GLS99] and [GLT99]) for the inter-process communication. We decided to use the MPI implementation LAM (Local Area Multicomputer) version 6.5.3 of the University of Notre Dame, USA, as this is one of the most stable, fastest and free available implementations of the MPI standard. LAM not only covers the original MPI standard but also most extensions of the new MPI-2 standard. Further information about the LAM can be found at [LAM].

LAM must be configured and compiled with the same C++ compiler as that used for the RasDaMan server. Don not forget to set include path and libraries of LAM (compiler and linker options) in the RasDaMan makefiles.

The source code of the parallel RasDaMan server should be compatible with other implementations of MPI (not tested) as only the following communication primitives from the MPI-1 standard are used:

- Init: initialisation of the MPI processes at start-up of the RasDaMan server
- Finalize: cleanup of MPI processes before server shutdown
- Send: standard blocking message send
- Recv: standard blocking message receive
- Probe: probing of an incoming message. This is required when receiving intermediate results. The sender and the length of the dynamic string has to be tested to allocate a buffer with adequate size.

As the RasDaMan server is programmed in C++ we used the C++ binding of LAM instead of standard C functions.

3.2 Messages

We can find two different classes of messages for the inter-process communication of the parallel RasDaMan server during a query execution[5]:

- controlling messages which only transmit a request
- messages for the shipping of dynamic structures, e.g. intermediate results or queries

The controlling messages are open, next, close, reset, and are always sent top-down within the query tree, i.e. from the master process to the worker processes and from a worker process to the tuple server. These messages contain no data, the distinction of the signal is done by message tags which can be used in an MPI message. The advantage of using a tag instead of, e.g., using request identifiers as data is, that processes waiting for a request only respond to messages with the appropriate tag.

Much more complex is the shipping of intermediate results between the processes. The following intermediate results can appear during evaluation of the query tree:

- simple data types, i.e. octet, short, long, char, boolean, unsigned short, unsigned long, float, double
- complex data types which are types that consist of simple types and/or other complex types (e.g., complex data type Pixel = {red, green, blue}, with red, green, blue being a octet)
- multidimensional data types, i.e. point, interval and multidimensional interval (e.g. multidimensional interval [10:30, 20:50, 100:200])
- persistent data objects, i.e. multidimensional objects which are not loaded from the relational database yet (identified by an object identifier)
- transient data objects, i.e. data objects which were already loaded from the relational database and resist in memory. These objects are a highly dynamic structure and are the most difficult case.

We will explain the packing and transmission of intermediate results with transient data objects as an example. In the RasDaMan source code a transient data object is a object of the class QT_MDD which is a dynamic structure as it contains member variables pointing to other classes (e.g. class TransMDDObject). All information of this object has to be extracted and stored in a stream (see Figure 5).

[5] We will not consider messages which pass down the client-server communication from the master to the internal MPI processes, e.g. open database, close database, sending of the query string, as this is straightforward

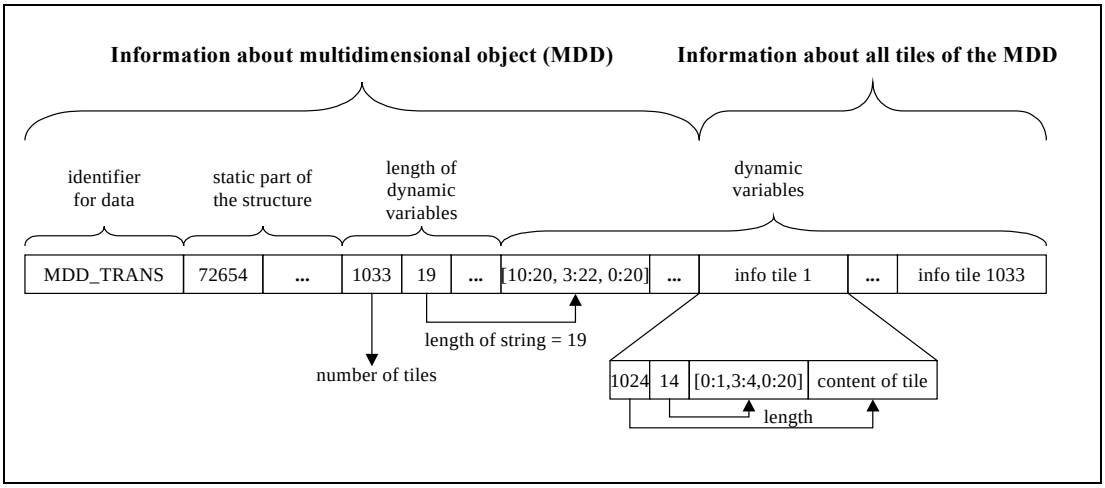

Figure 5: Transient data object, packed in a stream

The stream that will be sent with the MPI send command consists of

- a identifier for the data, in this case MDD_TRANS
- a static part of the structure (e.g. a unsigned long variable with value 72654 for the base type of the MDD cells, etc.)
- variables that specify the length of dynamic variables (e.g. the domain used here has length of 19 characters)
- dynamic variables. The length of this variables are not fixed (e.g. the spatial domain of the multidimensional object [10:20, 3:22, 0:20] which is a pointer to a character)

Transient multidimensional objects include pointers to tiles (storage format of the RasDaMan server) which again have static and dynamic parts, e.g. the content is dynamic as the tile size can vary.

Applying this concept the receiving process can identify the size of the dynamic structures and allocate buffers with adequate size. The receiver constructs a object of the class QT_MDD which is identical to the object that was packed and sent by the sender. This concept is the only way to transmit intermediate results. Using a fixed buffer size is not practical because the results can vary in size between one byte (e.g. NULL value) and objects with arbitrary size.

4 Summary

4.1 Achievements

As RasDaMan was not originally designed for more than one processor, query execution time of the RasDaMan server is often determined by CPU speed. Our goal was the utilisation of parallel hardware in order to speed-up CPU-bound queries, especially very expensive queries with execution times of several minutes or even hours. We designed a concept to split-up the computational work on multidimensional objects between different processes. This required an adaptation and segmentation of the query tree to allow different parts of the query tree to be processed by different processes. Further, an algorithm for the transmission of highly dynamic structures, i.e. intermediate results, has to be developed. In order to achieve good speed-up we minimised process initialisation time and inter-process communication.

The concept described of parallel query processing was fully implemented in the RasDaMan server kernel. Extensive test scenarios were performed regarding the structure of the resulting query tree and the intermediate results that have to be transmitted. The parallel server is stable for all tested queries and delivers the correct data to the client application.

Performance measurements prove the concept. On a two processor machine we observed an increase in speed by a factor up to 1.83 which is a very good result. Further performance measurements on computers with more processors will be done within the next weeks. We expect similar performance improvements on these machines as the concept implemented makes no assumptions regarding the number of processes.

The implemented data parallelism partitions the data with a granularity of complete multidimensional objects. This has the benefit that the concept is straightforward and avoids too much communication overhead which would lead to a loss of performance. On the other hand, if a collection only includes one multidimensional object and data is not referenced in more than one collection, no parallel speed-up is possible (the parallel RasDaMan server returns the correct result of course).

In order to summarise, the parallel RasDaMan server shows a very good performance speed-up for almost all queries, especially computational queries that are very expensive.

4.2 Future Work

The parallel query processing capabilities will be integrated in the new version 5 of RasDaMan as soon as possible. This will make possible the evaluation of the new parallel features by the ESTEDI partners using parallel hardware.

The first improvement of parallel RasDaMan server will be the implementation of an ”intelligent” paralleliser which analyses the query and decides dynamically if and how to parallelize the query tree. This could improve

performance, in particular queries only affecting one single multidimensional object. Another challenge will be the transfer of the concept used from multiprocessor machines to workstation clusters.

The most challenging future work will be the implementation of intra-object parallelism in the RasDaMan kernel. The estimation of costs produced by parallel processing and communication on the one hand, and speed-up on the other hand, is much more difficult than in the concept described here.

5 References

[AK01a] Active Knowledge: "RasDaMan Dokumentation Version 5.0", München, 2001

[AK01b] Active Knowledge: "RasDaMan C++ Developers Guide Version 5.0", München, 2001

[DG92] DeWitt D. J., Gray J.: ”Parallel Database Systems: The Future of High Performance Database Systems”, Communication of the ACM, 1992

[D1b00] Specification Report D1b of the ESTEDI project, 2000

[Furt99] Furtado P.A.: "Storage Management of Multidimensional Arrays in Database Management Systems", PhD Thesis, Technical University Munich, 1999

[GLS99] Gropp W., Ewing L., Skjellum A.: ”Using MPI”, The MIT Press, 2nd Edition, 1999

[GLT99] Gropp W., Ewing L., Thakur R.: ”Using MPI-2”, The MIT Press, 2nd Edition, 1999

[LAM] http://www.lam-mpi.org

[MPI] http://www.mpi-forum.org

[Nipp00] Nippl C.: ”Providing efficient, extensible and adaptive intra query parallelism for advanced applications”, PhD Thesis, Technical University Munich, 1999

[Rit99] Ritsch R.: “Optimization and Evaluation of Array Queries in Database Management Systems”, PhD Thesis, Technical University Munich, 1999