Benchmarking Intra-Query Parallelization in a Multidimensional Array Database System

Stojanco Stamkov*

* *School of Engineering and Science*
Jacobs University Bremen
28759 Bremen, Germany
s.stamkov@jacobs-university.de

Spring Semester 2007
Supervisor: Prof. Dr. Peter Baumann

Abstract

In query evaluation on a multidimensional array database, most of the time consumed is spent on CPU[1]-related tasks. Therefore, in order to increase the computational power of a DBMS[2], kind of a distributed processing mechanism is expected to be developed. As it is explained in the Overview section of this document, intra-query parallelism might be employed as one of the methods of efficiently increasing the processing power dedicated to a query. It is an established mechanism for attaining better performance in relational database systems, but intra-query parallelization has not yet not been widely applied to the upcoming field of multidimensional array databases. The first multidimensional array database employing this method is the *RasDaMan* DBMS, which constitutes the basic foundation for this paper. The aim of the paper is to test and analyze the work of such a DBMS deployed on a standard Linux cluster[3]. Building clusters of personal computers communicating over MPI (message passing interface) on a LAN[4] is a cheap and effective way of constructing systems with large amounts of CPU power. This paper discusses the benefits and analyzes the results of deploying a parallelized multidimensional-array database in such a distributed environment.

[1] Central Processing Unit
[2] Database Management System
[3] also called a Beowulf cluster
[4] Local Area Network

I. INTRODUCTION

I.A. Motivation

While relational databases are used for storing tabular data and its relations, multidimensional databases consist of arrays holding raw, binary data. Furthermore, as most of this data is automatically generated, and not user-contributed as the case with relational databases usually is, multidimensional databases frequently end up containing enormous amount of data. As a consequence, operations on such a database require great amounts of computer resources, including CPU power, disk space and high-speed communication. Considering that most queries on multidimensional data are computational queries [3], that is, the executional time of the query depends mostly on the processing power available to the DBMS, and are not bounded by I/O[5] operations. As a consequence, most of the efforts towards improving the performance of multidimensional databases should be focused on allocating more CPU power for the DBMS engine. Two common ways of increasing CPU resources is are: using Symmetric Multiprocessing (SMP) machines, which means having more than one processor in a single machine using shared memory, or, using workstation clusters - separate machines connected into a single computational network.

In this paper we will describe the implementation of parallel query processing in *RasDaMan* using MPI as a communicational interface. We will specifically concentrate on its performance on clustered Linux machines (called "Beowulf clusters"). However, the interface used (MPI) allows for implementation on a variety of hardware, including SMP machines and parallel supercomputers.

I.B. Overview on parallelism

There are several criteria according to which queries can be parallelized in relational databases. We can distinguish the types of query parallelism according to 2 main factors: the query execution process that is parallelized, and the structural part of the query that is parallelized. According to the first factor, we have:

- data parallelism: the same operation is done by several processes, each process using a subpartition 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.

According to the second factor, we have:

- 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 [4].

In the *RasDaMan* implementation, parallelization is achieved through intra-operator parallelism, separating the query into segments, and deploying the execution the most computationally intensive segment over the network of processors. As it will be shown further in the paper, due to the local nature of most of the CPU intensive operations, pipeline parallelism would not lead to major performance improvements. Instead, the concept of data parallelism is implemented, by assigning separate data fragments to each node for parallel processing. Explanation in greater details of these topics will follow in the subsequent sections.

[5] Input-Output

II. PARALLELISM IN *RasDaMan*

The parallel processing capabilities of *RasDaMan* were first implemented as a part of the ESTEDI project[6], and were presented in the paper "Parallel Query Support for Multidimensional Data: Inter-object Parallelism" by Karl Hahn et al. [1] Before analyzing the parallel query execution capabilities of *RasDaMan*, a query execution on the standard *RasDaMan* is explained.

II.A. Query processing

The standard structure of a simplified *RasDaMan* query is:

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

RasDaMan uses RasQL, a query language derived from the SQL[7] standard.

Before processing, the query is decomposed to a query tree, shown in figure 1. In the evaluation of the query, *RasDaMan* uses a method known as open-next-close. As inferred from the name, open-next-close uses the iterator concept for evaluating the results, and is composed of starting (opening) a query connection, going through the resulting structures (tuples) one by one, and finally closing the connection. [5]

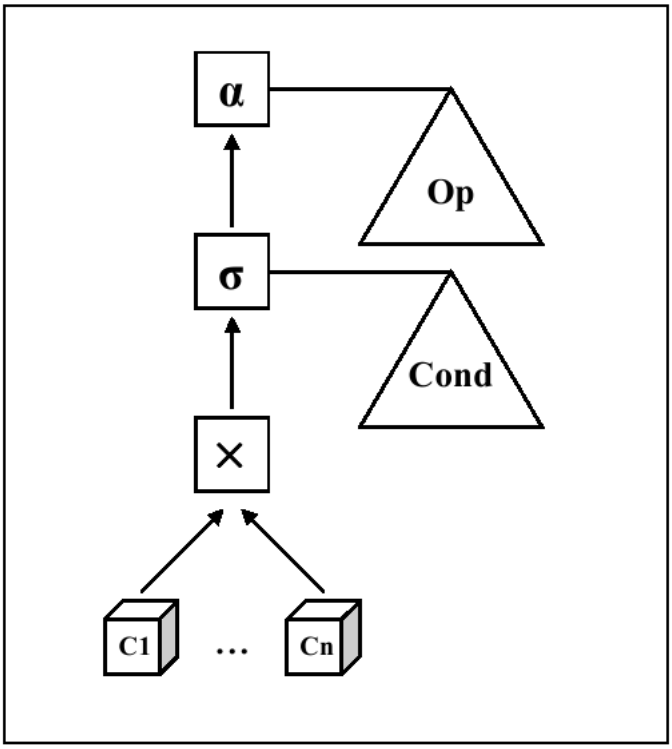

Fig. 1. A standard *RasDaMan* query

Let's dissect the query from the figure and analyze the internal steps of evaluation that *RasDaMan* undertakes. First, the method open() is called on the root node of the query (the operation). Further, the method invocation traverses the query tree from the root to the nodes (post-order propagation),

[6]European Spatio-Temporal Data Infrastructure For High-Performance Computing, http://www.estedi.org

[7]Structured Query Language

initializing resources necessary for the current and subsequent steps of the query execution, and further processing is left to the method next(). next() traverses the query tree in the same fashion, but evaluating in a bottom-up manner, returning each time a tuple from the result is obtained and starting again. The termination of the result retrieval stage is signalled by an exception, and finally, close() is called to release allocated resources and do the necessary clean-up. [2]

The example query can be split into 3 logical segments. These segments are actually used as a model for splitting the query at execution time as well, so the operations corresponding to each segment are explained, starting in a bottom-up fashion:

- cross product ($\times$): delivers the cross product of all the referenced objects from the referenced collections as N-tuples, where N is the number of referenced objects. In total, $|C_1| \times |C_2| \times ... \times |C_N|$ tuples are returned. Although this operation deals with great amounts of data, bulky binary data is not returned as a result and instead references to the data objects are returned. Segment represented by the FROM clause of the query.
- selection (σ): performs multidimensional selection operations on the cross-product results, including geometric operations (e.g. intersections, joins) and induced operations (e.g. for all, there exists). Calling next() on this segment returns the next data object which fulfills the condition. Segment represented by the WHERE clause of the query.
- application (α): final operation done on the data set returned from the previous stages. It may include trimming, sectioning or aggregation of the results. Segment represented by the SELECT operation of the query.

As mentioned above, the cross product operation does not transfer whole data sets, but instead just references to the cross-product results. In addition to not being communicationally expensive, the cross-product operation is not very computationally expensive either. Thus, the selection and application operations are the most time-consuming query operations, mainly because of their requirement for computational resources, and are the obvious choice for a distributed execution.

II.B. Work distribution and parallelization among RasDaMan processes

RasDaMan uses 2 administrative processes, and an arbitrary number of worker processes. All processes, although not needed immediately, are spawned on start-up of the *RasDaMan* server for increased responsiveness and efficiency when query execution is actually requested. Hence, a total number of n+2 processes is recommended, where n is the number of the processors we want to utilize for the parallel setup. Follows the division of *RasDaMan* process types:

- master process: responsible for server-client communication (via RPC[8], HTTP[9] or, in newer versions, RNC[10] protocol). Distributes the queries and the work to the internal (worked) processes, and communicates with them using MPI. Tasks include:
  - starting the *RasDaMan* server, by executing MPI commands which include the name of the server executable and its command-line options (protocol, port, server hostname etc.). This command is sent to all the MPI nodes, assuming that the server executable can be found in the system path on each node.
  - stopping the *RasDaMan* server, functions similar to starting it. A message is sent to all nodes to terminate the worker processes that have been started by the server.
  - opening and closing the database, by sending the appropriate database parameters to the processes

[8]Remote Procedure Call
[9]Hypertext Transfer Protocol
[10]Remote Network Processor

  - query execution, by assigning tasks to the internal processes and distributing the workload. Also collects results from the working processes, requests the rest of the results (by calling next() on the processes), and finally transmits these results to the client program which requested the query.
- internal tuple server process: responsible for retrieving, managing and distributing the multidimensional tuples created from the cross-product operation. Tasks include:
  - providing administration of the multidimensional tuples for the rest of the processes.
  - sending result identifiers to worker processes. Identifiers are sent on request, instead of sending all tuples at once, which helps balancing the load on the worker processes uniformly.
- internal worker processes: one or more processes responsible for the computationally intensive operations like selection and application. They receive results from the tuple server and process them. Tasks include:
  - using the method next(), tuples from the tuple server are obtained and the conditions and operations are evaluated
  - when the evaluation of the tuple matches the constraints imposed by the query, the suitable tuple is sent to the master process

  This is the type of process that actually does the computational work, it is the main performance bottleneck in the query execution and hence it's the most interesting process type for our parallelization task. In the parallel version of *RasDaMan*, there are only 2 administrative processes, which both run on the processor where *RasDaMan* was started, but there are normally n worker processes, one for each of the n processors on which the server is deployed. Each worker process performs both the application and selection operations, so that cached data from the selection operations can be reused by the application operation without the performance hit of freeing, transferring and re-allocating intermediate memory.

As it is evident which parts of the query execution are most computationally expensive and thus time-innefficient, and furthermore the modularity of *RasDaMan* clearly separates those processes from the rest, one obvious and straightforward method for optimization is intra-query parallelization. This method is implemented by distributing the worker processes over separate processors, significantly increasing the processing capacity of the system. Figure 2 shows the query tree for such a distributed environment. Compared to figure 1, direct inter-process communication is replaced with MPI calls, which provide flexibility and modularity to the system.

II.C. Inter-Process Communication

In this sub-section a brief overview on the types and content of the used MPI messages will be provided. Special emphasis will be put on intermediate results and their transfer, as it will contribute to the better understanding of the possibilities and limitations of the intra-query parallelization technique used. Following is a short list of the MPI-1 compatible MPI methods used in the implementation.

- init: initialization of the MPI communication, required for every MPI program
- finalize: shutting down the MPI communication channels and processes, again required for MPI programs
- send: standard blocking message sending. Blocking means that all communication is blocked until send returns, either by successfully sending the data or otherwise
- recv: standard blocking message receiving. Similarly to send, recv will block any other MPI communication attempt before all input expected arrives, or an error signal is received
- probe: probing for incoming messages, returns a list of all messages waiting to be received. This additional function is required for getting information about intermediate results being sent, and appropriately allocating a memory buffer with the sufficient size beforehand

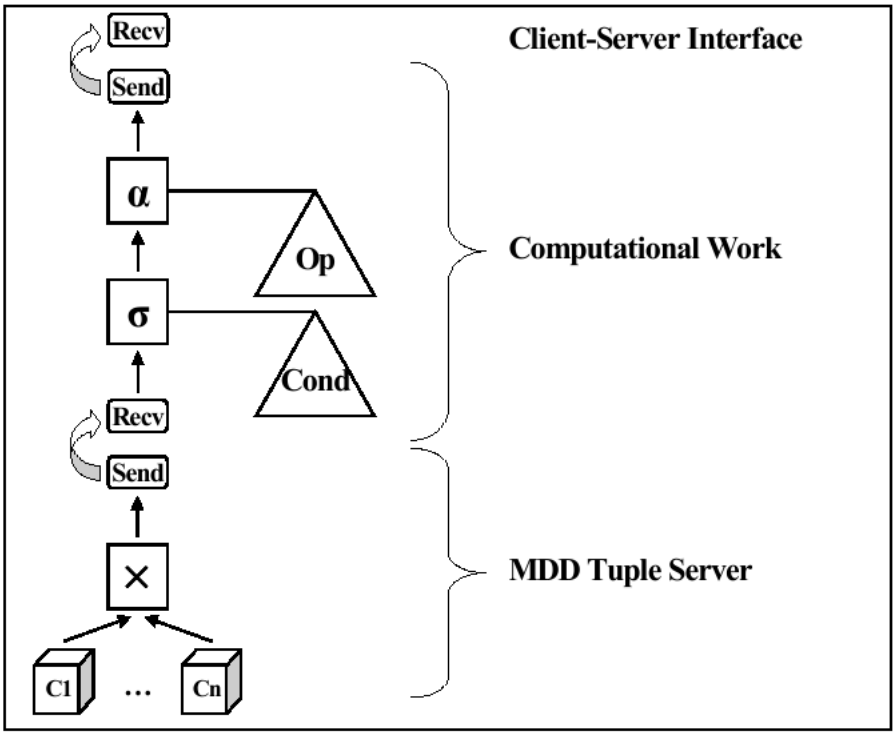

Fig. 2. A parallelized *RasDaMan* query, using MPI methods for inter-process communication. The stage where the computational work is done is distributed over multiple processing units

Now, knowing the methods used to transmit messages between processes running on different CPUs, we can analyze the two types of messages the parallel *RasDaMan* uses:

- controlling messages
  Controlling messages only serve for communicational, instead of transportational purposes. They are used to transmit requests between the nodes, such as: open(), next(), close(), reset(), and as mentioned in section II-A, are sent from the root to the leaves of query tree, i.e. from the master process to the worker processes and from the worker process to the tuple server process. Conversely, the results travel in the opposite direction of the query tree. These messages don't need to contain any data, and are in fact empty MPI messages identified only by the tags which can be set on MPI messages. This proves to be sufficient for the purpose, as processes can recognize the tags, associate them with the corresponding request and take action accordingly.
- messages for transferring dynamic structures, i.e. intermediate results
  As expected, transferring data structures is much more complex than sending controlling messages. One of the hurdles in transferring dynamic structures is that there is a variety of data types that need to be transfered transparently. The basic data types are octet, short, long, char, boolean, unsigned short, unsigned long, float and double. In addition, we might have more advanced data types, consisting of basic data types (comparable to structs in C/C++), or consisting of other advanced data types and a number of basic data types (comparable to classes in object-oriented programming languages). Moreover, we can have multidimensional, persistent and transient data objects. Worth noting is that the transient data objects, being preloaded from the relational database backend, reside in memory and can contain pointers pointing to various other data types, which makes them the most challenging case for extracting and sending all the required data in an MPI transfer.

The overall process structure of the parallel *RasDaMan* array DBMS is shown in figure 3.

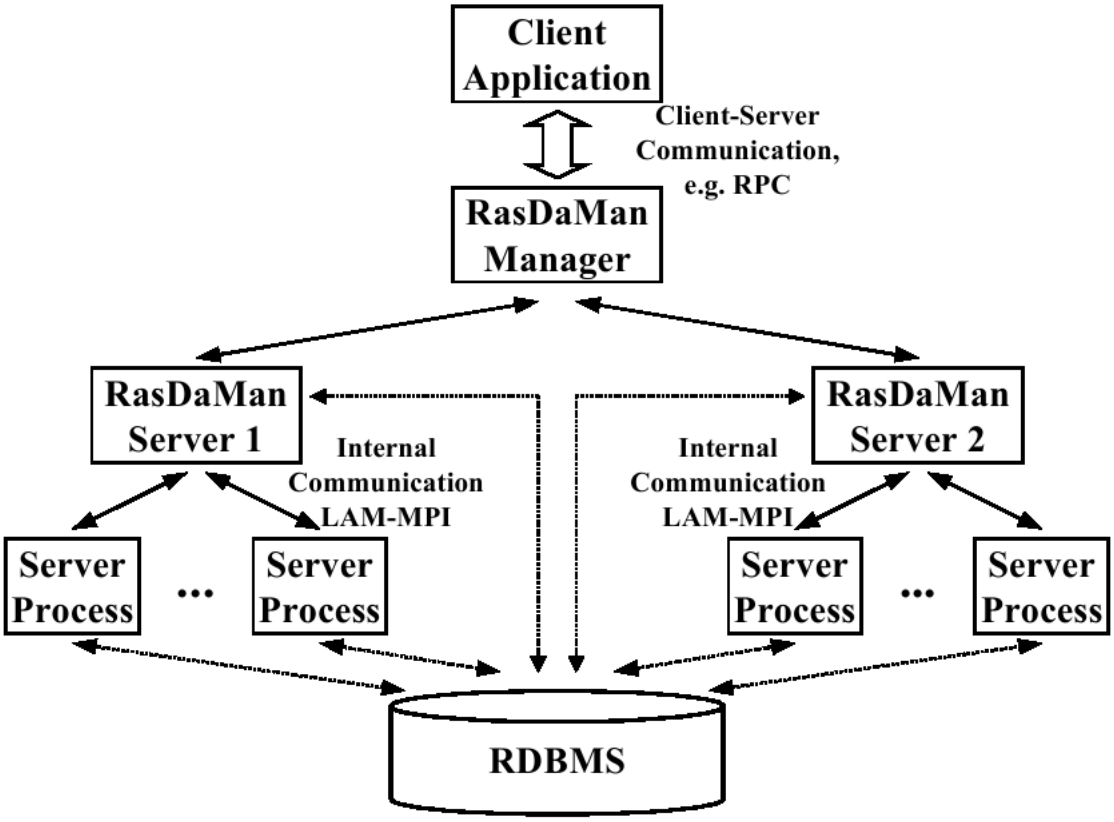

Fig. 3. Architecture of the parallel *RasDaMan* server

III. Comparison - SMP vs. LAN process distribution

III.A. Setup

In the original setup [1], a Symmetric Multiprocessing Machine (SMP) was used (named SunMachine in the comparison table). As the aim of the paper is to test and analyze the work of *RasDaMan* on a standard cluster of Linux machines, connected via Message Passing Interface (MPI), our setup is somewhat different. It consists of two x86[11] machines, named Mojito and Maitai, connected into a Ethernet LAN, both holding installations of the parallel *RasDaMan*. Following we have the full specifications of both setups, including hardware, software and networking configuration.

[11] 32-bit Intel architecture

	SunMachine	Mojito	Maitai
OS	SunOS	GNU/Linux	GNU/Linux
Kernel version	N/A	2.6.8-2	2.6.13-15
Distribution	N/A	Suse	Debian
Number of processors	2	1	1
Processor type	UltraSPARC II	Intel Pentium III	Intel Pentium 4
Processor clock	400MHz	1GHz	2GHz
Main memory	1.1 GB	256 MB	512 MB
HDD capacity	100 GB	40 GB	40 GB
Relational DBMS	Oracle 8.1.6	Oracle 10g	N/A
Network interface	N/A	Ethernet 100baseTx-FD	Ethernet 100baseTx-FD[a]
LAM-MPI version	7.1.1	7.1.3	7.1.3
SSH[b] version	N/A	OpenSSH 4.1	OpenSSH 4.1

[a] 100 Mbps Full Duplex
[b] Secure Shell. Used by LAM-MPI for internal communication

III-A.1) LAN setup - detailed procedure and challenges: The parallel version of *RasDaMan* developed by Karl Hahn et al. [1] successfully employed intra-query parallelization methods and managed to achieve a significant level of optimization with some common query types. It was designed with portability and extensibility in mind, using the highly flexible and well-defined MPI standard, and developed and tested on a POSIX-compliant operating system[12].

However, the actual implementation, realized mainly as a "proof of concept", was very limited to the research environment where it was developed and tested. It was deployed on a SUN architecture, using a SUN version of UNIX, and using a SMP multi-processor environment. Also, being a research project, external documentation regarding the requirements, environment and deployment of the parallel *RasDaMan* were extremely scarce. In addition, as the paper was written in 2002, many tools and methods used in the development and employment have changed, either by introducing new versions or by becoming obsolete and replaced by newer packages. Each new version of a single development tool can bring subtle (or obvious) bugs or functionality impairments to a program, and 5 years[13] of unmaintained program might require major maintenance work in order to preserve compatibility. *RasDaMan* is especially exposed to such factors, as it consists of a large code base, with modules and APIs[14] in a number of different programming languages (C, C++, Java, shell scripts etc.), utilizing many CASE[15] tools (such as Make, Flex, Bison etc.). Furthermore, *RasDaMan* has to maintain connectivity modules with the supported relational DBMS backends, such as Oracle, O2, DB2 etc., which can also change over time.

Having the given situation and circumstances, updating and configuring *RasDaMan* to work on our setup was a time consuming and non-trivial process. For compatibility issues, we used an MPI implementation which is no longer maintained (LAM-MPI 7.1.3) and is now merged with the newer OpenMPI implementation. The machines were assigned SSH key pairs to enable automatic, non-interactive MPI initialization. Next step of the environment configuration was installing and configuring Oracle version 10g, which, given the hardware resources of the host machine (Mojito), was quite time consuming. Once the environment was ready, the compilation stage was started. There were numerous glitches and incongruities, especially notable being the ones related with Flex

[12] SunOS, also known as Solaris, is based on UNIX System V, which is POSIX-compliant
[13] The amount of time passed since 2002, as of 2007
[14] Application Programming Interface
[15] Computer-Aided Software Engineering

incompatibilities which needed to be remedied. Another incompatibility was the timing mechanism used in the original implementation, which was SunOS specific and had to be replaced with standard GNU/Linux C/C++ time utilities[16]. There were numerous issues arising from the utilities managing the debugging output as well, which apparently, being debugging utilities, weren't tested sufficiently and thus lacked robustness and functionality. The debugging mechanism had to be improved and some major classes re-implemented, which finally resulted in a stable and reliable debugging utility. Having access to the exhaustive program logs, further debugging was much faster and more straightforward. At this stage, having a compiled *RasDaMan* and a suitable environment, last and final step for completing the setup was to create the internal *RasDaMan* table structures in the backend relational DBMS, namely, Oracle. As the original parallel *RasDaMan* was tested on Oracle 8.1.6, substantial modifications needed to be done to the creation scripts (which are shipped with *RasDaMan*) in order to achieve compliance with the new Oracle 10g DBMS. As each recreation of the table structures of *RasDaMan* took around 90 minutes to complete on Mojito, this stage ended being much more time-consuming then expected.

III.B. Performance and result evaluation - SMP setup [1]

Typical RasQL queries on 2-dimensional and 3-dimensional data were chosen. The example the benchmarking was run on is the following:

A collection containing 60 3D MDDs[17], with spatial domain of [0:63,0:127,0:119] and base type float, each with size of $64 \times 128 \times 120 \times 8$ B=3932160B$\approx$4 MB [18], and thus the size of the entire collection is $\approx$240 MB. Each 3D cube represents temperature values in a specific height above sea level in degrees Kelvin for 10 years (120 months) and was calculated in meteorological simulations.

The query executed on this collection (named mpim3d) was:

```
SELECT all_cells(a > 200.0)
FROM mpim3d as a
```

This simple RasQL statement analyzes each MDD of the collection and returns a boolean for each MDD, indicating if all cell values of the MDD are greater than 200.0 degrees Kelvin. As it is argued in the forementioned paper, this query is particularly well suited for parallel execution because:

- in the application stage of the query an induced operation ($>$) and an aggregation operation "all_cells" is performed, which are both strongly CPU-bound
- the collection includes 60 sufficiently large MDD. The data volume is sufficient, there aren't too many tuples to lookup from database, and computational costs can dominate over communication costs, and as the data was randomly generated in simulations, data skew does not not appear
- the master server needs to receive minimal results from the worker processes, in our case just 60 boolean values, i.e. the intermediate result transferred is almost at the lowest possible amount

As reported in the paper, this query achieved a relative improvement of about 1.91 (t_{old}/t_{new}). Otherwise, the typical speed-ups of CPU-bound queries lied within a range of 1.60 and 1.95. According to the paper, the time distribution over the internal process classes was:

- about 2% for the tuple server, as the process does not do computational work but handles central administration of the tuples. The time consumption of this process class is low in most cases.
- about 4% for the master process, as in this case the result to be transmitted to the client is very small, namely, boolean values

[16] the sys/time.h library was used

[17] Multidimensional Discrete Data

[18] Mega Bytes, 1 Byte=8 bits, a standard float type consists of 32 bits, 1 MB = 2^{20} B

- 94% for the two worker processes, as most of the query evaluation relies on computationally intensive operations done in the worker processes

From the above distribution, it becomes apparent once again that this query is extremely suitable for benchmarking, as more than 90% of the time spent on its execution is a part of the worker processes and can be parallelized.

In figure 4 we can see the performance chart for 10 CPU-bound queries with ascending responding times. It is important to note the threshold region at around 3 seconds, where the parallel version of *RasDaMan* becomes more efficient than the standard one. This can be explained by the fact that for short query executions, the overhead caused by initialization time and communication latency of the MPI processes exceeds the speed-up obtained due to the parallelization. However, after the 3 sec threshold, the time efficiency of the parallel *RasDaMan* is almost double the one of the standard *RasDaMan*.

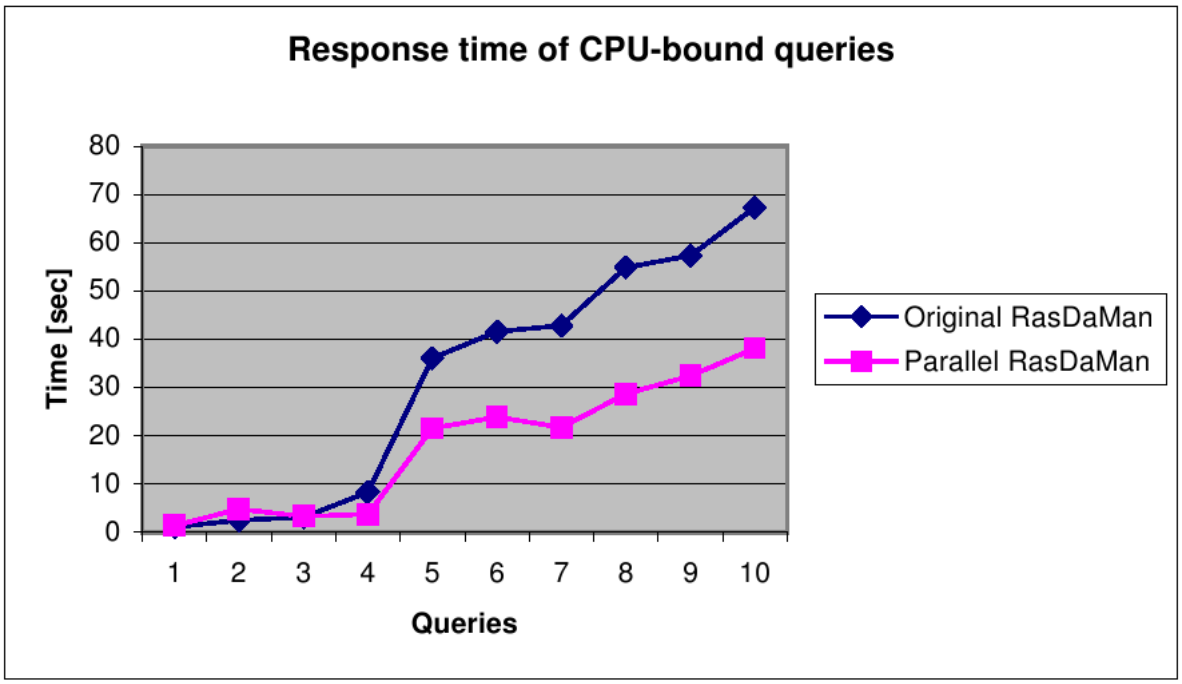

Fig. 4. Performance of the original *RasDaMan* server compared to the parallel implementation on a 2-CPU SMP machine

III.C. Performance and result evaluation - LAN setup

For our benchmarking, a collection with variable number of GreySet MDDs was used. GreySet is a 2 dimensional MDD type, with base type of char. The dimensions of each MDD was [0:1222, 0:855], which results in approximately 1 MB of data per MDD. The image used for generating the MDDs was randomly chosen and contains approximately even distribution of pixel values, i.e. the average value of the pixels is around the expected value of 128, which implies that the data imported will be free of significant data skew. This image is shown on figure 5.

The image was first scaled up to the right size, and then converted to pgm type and imported using the insertppm utility that comes with *RasDaMan*. The query executed, similar to the SMP setup, is:

```
select all_cells(x>200)
from bench_coll as x
```

Fig. 5. A scaled-down version of the image used as an MDD, taken from the Wikipedia article on “Grayscale” - http://en.wikipedia.org/wiki/Grayscale

where “bench_coll” is the name of the collection.

We will analyze 3 different benchmarking tests, specifically chosen to emphasize specific aspect of the intra-query parallelization.

The first one, shown on figure 6, shows the execution of different number of queries over a constant number of 10 MDD objects. The improvement ration is relatively constant, averaging to 1.5.

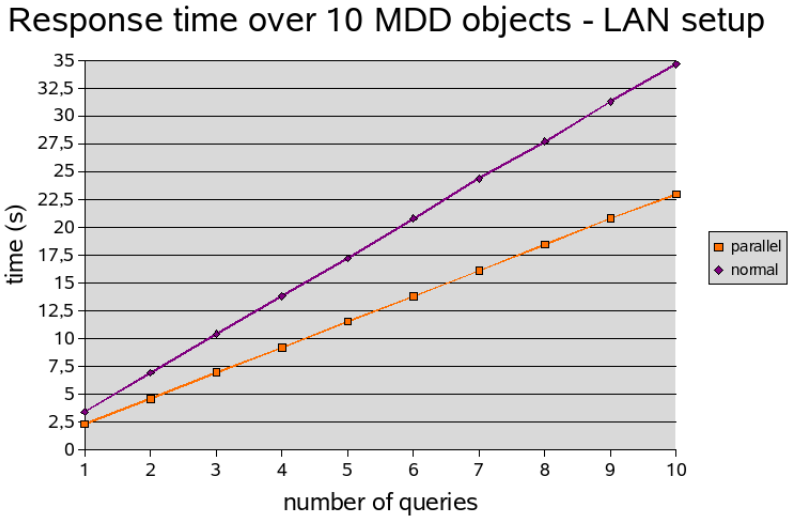

Fig. 6. Response time over 10 MDD objects

The second plot, (figure 7), shows the execution of a single query over an increasing number of MDD objects. The improvement ratio is fluctuating, mainly because of variable instantiation times of the communicational channels, as we are observing an execution of a single query at a time. There is an increasement tendency, because of the increasing possibility for load distribution among the worker processes. As it was explained in the previous sections, the number of MDD objects is crucial for the ability to parallelization a query. the The average improvement has increased to is 1.568.

Finally, in the third plot (figure 8), we are analyzing a constant number of queries (5) over an increasing number of MDD objects. Our hypothesis about the fluctuations from the previous plot can be confirmed, as it can be seen that the slopes are much smoother when we have multiple query executions at a time. There is no significant increasing or decreasing of the optimization ratio, and as expected, the average has rised slightly to 1.573. The reason for the increase of the ratio is the multiple number of queries executed at once, which helps save on initialization time. However, the

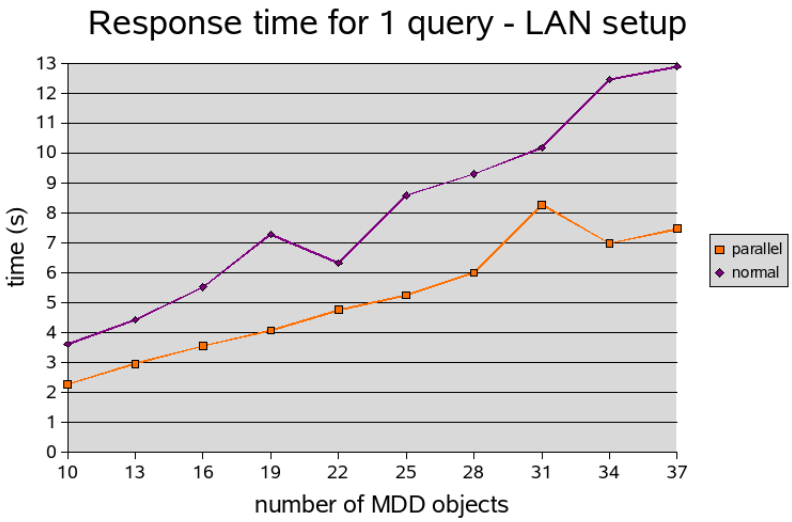

Fig. 7. Response time for 1 query

small amount of increaseement indicates that we're near a threshold where the ratio approaches a constant, i.e. adding more queries or MDD objects will not increase it.

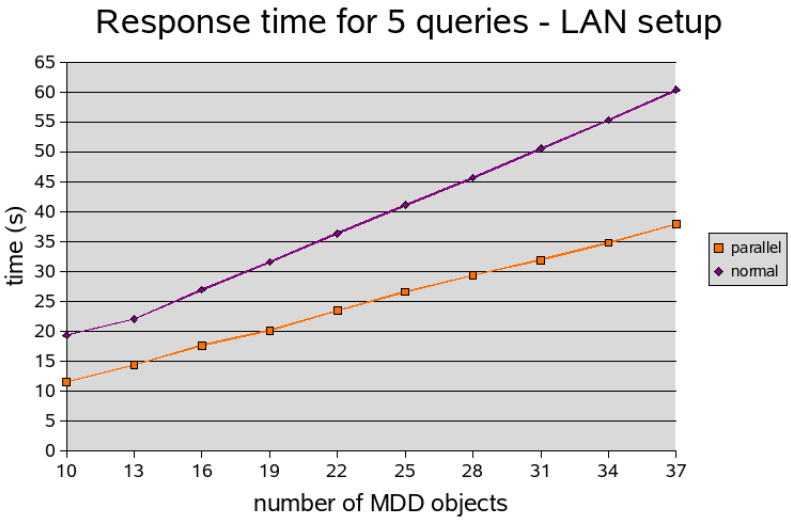

Fig. 8. Response time for 5 queries

IV. Summary

IV.A. Achievements

The field of multidimensional array DBMS is still young and evolving, but it is already evident that, compared to relational DBMS, multidimensional array operations are usually CPU-bound as opposed to I/O-bound. This fact stirred the motivation of the original effort [1] for investigating the parallelization possibilities of *RasDaMan* and eventually implementing them. The outcome of the research was very satisfying, as improvements of a factor up to 1.91 of the time consumption were obtained. However, as with many other research projects, the product was not meant for widespread use. It was not tested on multiple software platforms, multiple architectures or different parallel environments. Moreover, a good portion of the code is outdated now, 5 years after the first implementation. We menaged to bring the implementation to today's computer world and test it. By using standard and widely used GNU/Linux distributions, cheap and available Ethernet multiprocessor setup and current day programming and operating environment, we have secured the further development of the parallel *RasDaMan*. The parallel *RasDaMan* is currently functioning with up-to-date compilers, relational DBMS and programming tools, on the common Beowulf cluster setup, and it is one step closed to seeing the light of the day as an efficient, production-use multidimensional array database.

IV.B. To do

- integrating parallel features of *RasDaMan* version 5, which is the one we used in the benchmarking, with the current, much improved *RasDaMan* version 6.
- deploying on larger Beowulf clusters and investigating the scalability of the setup, i.e. until what level will the improvement increase proportionally and when it will stagnate
- extensive testing of parallel *RasDaMan*, so that the software quality reaches the one of the much older and more tested original code base
- implementing an algorithm for the transmission of highly dynamic structures, i.e. transient intermediate objects, still needs to be developed. In the current implementation, there is only limited support for advanced datatypes
- implementing additional parallel concepts on different query tree segments, or implementing inter-query parallelism
- finally, implementing a mechanism for automatically recognizing queries which can be optimized via parallelization (of the type discussed in subsection III-B) and can use the parallel module, and which can not and will be executed normally, on a single processor

V. Conclusion

A parallel *RasDaMan* setup is still very effective when the LAN is used as the MPI communication channel, instead of the loopback interface used in the SMP machine. The disadvantages of our approach lie mainly in the fact that LAN networks are not as efficient, in terms of data throughput and response time, as the internal communication in a single machine would be. Additionally, many external factors can influence, cause instability or congest a LAN network, as the network can be used for multiple purposes.

Still, performance measurements for suitable queries show a very satisfactory optimization ratio, averaging around 1.55. Albeit understandably slower than the SMP machine setup, the LAM setup can offer a much larger flexibility, allowing for physically remote computers to connect as nodes in a

processing network. Additionally, standard PC[19] configurations are much cheaper and available than multiprocessor machines. And finally, in favor of the networks, we can state that they are constantly improving, both in bandwidth and in latency, and the communicational efficiency of our LAN setup will just keep on increasing.

The currently implemented parallelism includes only a fraction of the procedures that can be parallelized via different methods, but nevertheless shows that the concept is viable and that the implementation chosen is portable, extensible and scalable. Also, the parallelization method employed is straightforward and efficient in the communicational aspect, and furthermore, seems to fit perfectly with the way the *RasDaMan* DBMS is structured. However, this method has it's shortcomings as well, which seem to origin from the same reasoning as the benefits. Because of the straightforward distribution of tuples among the servers, the method is efficient only when there is a larger number of objects in a collection; otherwise, the communicational overhead makes the efficiency at best as good as the efficiency of the standard server. As a final conclusion, we have confirmed yet again the efficiency of the parallel *RasDaMan* , and additionally, we have tested it's portability and scalability. We have also done more exhaustive benchmarking than in the original implementation, and all the indications show that the parallel *RasDaMan* is a promising concept which can be of interest of many applications which depend on the query execution time.

[19] Personal Computer

REFERENCES

[1] Karl Hahn, Bernd Reiner, Gabriele Höfling and Peter Baumann, *Parallel Query Support for Multidimensional Data: Inter-object Parallelism*, 2002.

[2] Karl Hahn, *Parallel Query Support for Multidimensional Data*, VLDB Workshop: Supercomputing databases, 2001.

[3] Roland Ritsch, *Optimization and Evaluation of Array Queries in Database Management Systems*, PhD thesis, 1999.

[4] DeWitt D. J., Gray J., *Parallel Database Systems: The Future of High Performance Database Systems*, 1992

[5] Joe Hellerstein, Anthony Joseph, *Advanced Topics in Computer Systems - Query Processing*, http://bnrg.cs.berkeley.edu/∼adj/cs262/Lec_11_5.html

[6] Active Knowledge GmbH, *RasDaMan Version 5.0 Documentation*, 2001, http://www.active-knowledge.de