Interprocessor Communication Support in the Omega Parallel Database System

Leonid B. Sokolinsky
Chelyabinsk State University
Chelyabinsk, Russia
sokolinsky@acm.org

Abstract[1]

Interprocessor communication is one of the ultimate factors affecting the performance of parallel DBMSs for massively parallel computer systems. This paper presents a hierarchical architecture of parallel DBMS Omega which allows to manage the problem of interprocessor communication. The paper describes a message system designed for DBMS Omega. This message system consists of two self-contained subsystems: Ω-Conductor and Ω-Router. The Omega system is a prototype of the parallel database machine elaborated for MVS-100 massively parallel computer system. Omega project is developed in Chelyabinsk State University (Russia).

1. Introduction

One of the problems bound up with parallel DBMSs performance is the problem of interprocessor communication overhead reducing. It's especially important for massively parallel computer systems where the processors can't share memory to communicate each other.

In order to manage this problem we can try to reduce interprocessor communication to ultimate minimum. This strategy leads to *shared-nothing* architecture [1]. On a conceptual level, in a shared-nothing system, each processor has its own private memory and its own disk subsystem and it acts as a server for the data on its disks; processors communicate by passing messages through an interconnect (see Figure 1).

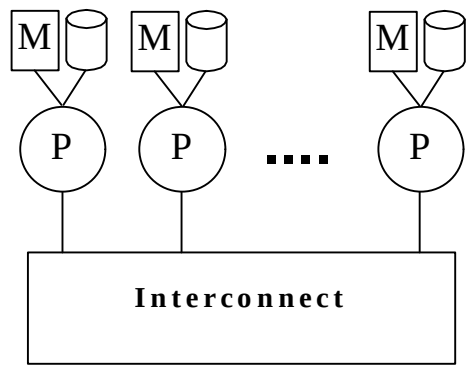

Figure 1. Shared-nothing architecture

In such systems, tuples of each relation in the database are partitioned (declustered) across disk storage units, attached to each processor. A relational operator can be split (replicated) into many independent operators (replicas) each executing on separate processor and scanning its own part of the relation. Then the results of the replicas are merged into a total relation.

Shared-nothing architecture was investigated in various research projects including Gamma [2], Bubba [3] and Arbre [4] and used in several commercial DBMSs, e.g., NCR's Teradata [6] and IBM's DB2/PE [5]. This architecture allows achieving high-level parallelization without huge data chipping through interconnect. The most serious problem for shared-nothing systems is load balancing [7]. To manage this problem we have to dynamically reorganize the data in the database in proper way. Such reorganization usually requires huge interconnect traffic. An interesting decision is proposed in [8, 9]. It's based on a hierarchical parallel system whose nodes are shared memory multiprocessor.

In this paper we consider another kind of hierarchical architecture used for parallel DBMS Omega [10]. Omega system consists of shared-disks multiprocessors interconnected by high-speed network. This hierarchical architecture allows managing the problems of interprocessor communication and load balancing. Omega parallel DBMS is

[1] This work was supported by Russian Foundation for Basic Research under Grant 97-07-90148.

Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999

implemented on MVS-100 massively parallel multiprocessor system [11].

MVS-100 is a Russian super-computer family with MIMD architecture. This system was designed by Scientific Research Institute "Kvant" (Moscow), Applied Mathematics Institute of Russian Academy of Sciences, Mathematics and Mechanics Institute of Ural's Branch of Russian Academy of Sciences and some other research institutes. MVS consists of multiple *processor units*. Each processor unit has uniform structure (see Figure 2).

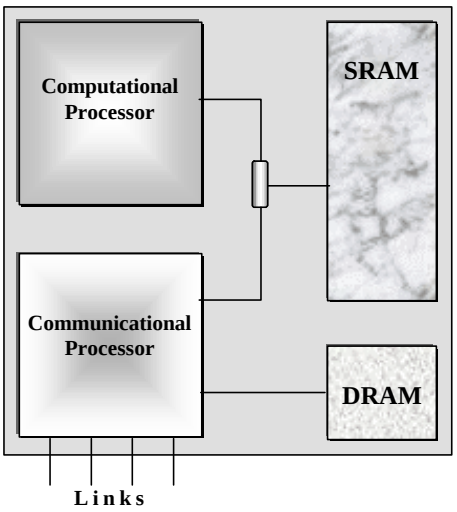

Figure 2. MVS processor unit.

The unit consists of two processors: computational processor and communicational processor. These two processors share common SRAM (static random access memory). The communicational processor has additionally its own private DRAM (dynamic random access memory). Communication processor has four links. The link is bi-directional high-speed interconnect channel. Processor units are bounded by links in a network. Some link can connect the system with disk subsystem unit or with host-computer. One system can consist of hundreds processor units, several disk subsystem units and one or more host-computers. Topology of interprocessor connections is not fixed and can be varied from simple rule box to hypercube.

As an operation system, MVS-100 has OS Router designed in Applied Mathematics Institute of Russian Academy of Sciences. OS Router is toolset class distributed operating system. This is single-task operating system. It means that only one process can be executed on each processor. OS Router is running on the communicational processor; user task is running on the computational processor.

OS Router provides the following main functions:

- loading a user program from a host at a processor module;
- data exchanging between user task and host in a way of some subset of UNIX I/O system functions;
- message exchanging between tasks run on different processors.

Input/output operations are executed on communicational processor *in parallel* with user tasks run on computational processors. This is an important feature of MVS computer system.

The rest of the paper is organized as follows. Section 2 gives an overview of the hardware and software architecture of Omega system. In section 3, we describe the software architecture and main functions of Ω-Conductor subsystem. The software architecture and main functions of Ω-Router subsystem are described in section 4. Conclusion remarks and future research directions are given in section 5.

2. Architecture of Omega System

2.1 Hardware Architecture of Omega

In [10], an original hierarchical hardware architecture for Omega system was proposed. In accordance with this architecture the system is built up as a shared-nothing system whose nodes are shared-disk multiprocessors called Ω-clusters. All these Ω-clusters have uniform structure shown in Figure 3.

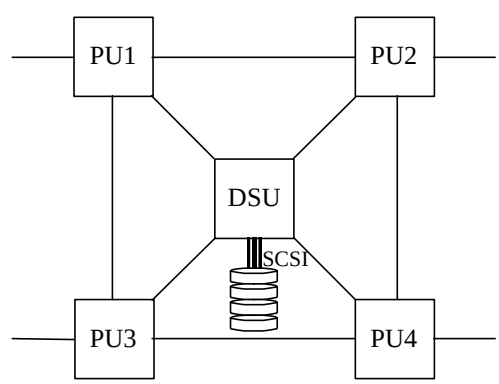

Figure 3. Ω-cluster structure (PU – processor unit; DSU – disk subsystem unit).

Each Ω-cluster consists of four processor units connected by links. Every processor unit is connected to disk subsystem unit (DSU). DSU has its own communication processor with private memory connected to four disks by SCSI adapter. One link of each processor unit remains free to connect a Ω-cluster to others Ω-clusters. In general, Ω-cluster may have structure differs from one shown in. However the following general principles remain constant:

- The count of processor units in Ω-cluster hasn't to exceed the count of disks. In every point of time a processor unit may have access only to one disk. In this sense the Ω-cluster architecture is similar to shared-nothing architecture. But in different points of time a

processor unit may have access to different disks. In this sense the Ω-cluster architecture is similar to shared-disks architecture.

- The Ω-cluster has at least two free links to connect another Ω-clusters. It is conditioned by fault-tolerance requirements.
- Every processor unit of the Ω-cluster has to be directly connected to disk subsystem unit. The minimum of the path length between two processor units hasn't exceeded 2. Thereby the Ω-cluster has closely-coupled structure.
- The count of processor units in Ω-cluster hasn't to exceed six. So the Ω-cluster is limited scalable.
- All Ω-clusters within the certain system has to have identical element structure and interconnect topology. It allows to use double-level addressing: the number of a Ω-cluster in the Ω-system and the number of a processor unit in the Ω-cluster.

The Ω-system is built up from uniform Ω-clusters. One Ω-system can be scaled up to several hundreds Ω-clusters. There is no restriction for the interconnect topology in the Ω-system.

A sample of the interconnect topology, called *simple rule box*, is shown in Figure 4.

The existence of two hosts in the configuration is conditioned by fault-tolerance requirements.

This hierarchical architecture assumes two levels of data partitioning. Relations are partitioned across Ω-clusters and within each Ω-cluster across disks.

This approach allows to manage load balancing more flexibly because it can be addressed at two levels, locally among the processors of each Ω-cluster and globally among all Ω-clusters.

Inter-cluster communication is done via *W-router* message-passing system described in section 4 while inter-processor communication within a Ω-cluster is done more efficiently via *W-conductor* described in section 3.

The advantage of Ω-conductor is based on closely-coupled predefined Ω-cluster structure. In this sense the Ω-conductor implementation is topology-dependent while the Ω-router implementation is topology-independent.

2.2 Software Architecture of Omega

As it was shown in [12], an operating system support provided by an universal operating system is not appropriate for specific DBMS needs in many ways.

For example, a message passing initiation in UNIX-similar distributed operating system Helios for MVS-100 includes about 3000 instructions.

It forces DBMS designers to create a special operating environment for the DBMS, which actually corresponds to particular micro operating system.

In Figure 5, the prototype software architecture of Omega system is shown. This software runs on each processor unit and splits by three levels.

Both OS Router and MVS-100 processor unit compose a virtual processor module. We can assume this module has a single (computational) processor and private memory.

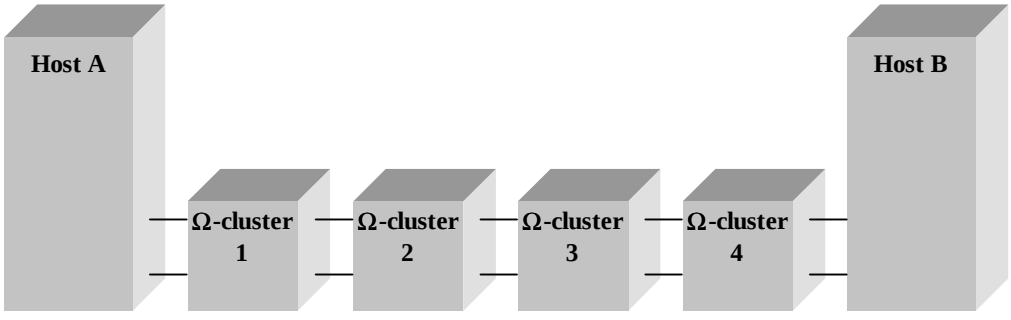

Figure 4. Possible Ω-system hardware architecture: Simple rule box.

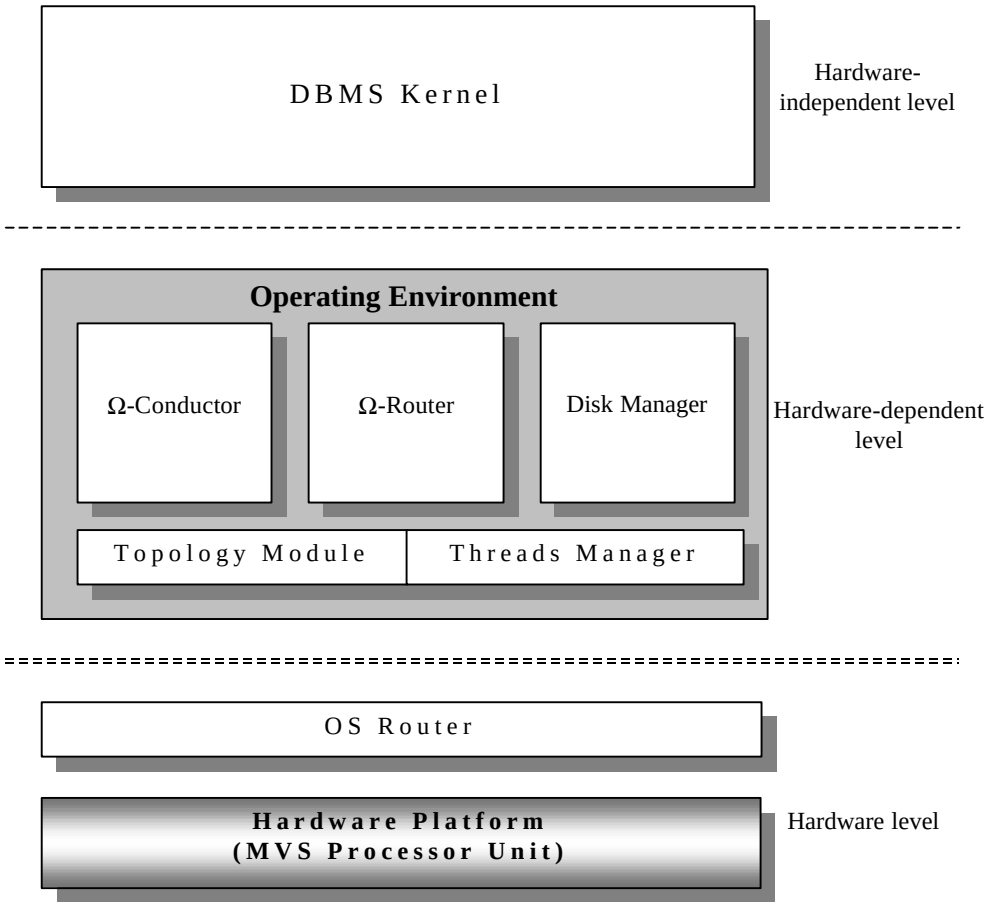

Figure 5. Major system components at one processor unit

These virtual modules can communicate each other by using the following OS Router functions.

- To run a message receipt:
  int **r_read**(int /*process*/, void* /*buffer*/, int /*length*/);
- To run a message delivery:
  int **r_write**(int /*process*/, void* /*buffer*/, int /*length*/);
- To test, is channel free for receipt:
  int **t_read**(int /*process*/);
- To test, is channel free for delivery:
  int **t_write**(int /*process*/);
- To wait for channel release for receipt:
  int **w_read**(int /*process*/);
- To wait for channel release for delivery:
  int **w_write**(int /*process*/).

We should point out that **r_read** and **r_write** operations are performed in asynchronous mode. Functions **t_read**, **t_write, w_read** and **w_write** are dedicated to synchronize these input/output operations. In Figure 5, this level is called "Hardware level".

The second level is presented by Omega *operating environment*. In, this level's called "Hardware-dependent level". The Omega operating environment includes five subsystems: topology module, threads manager, Ω-conductor, Ω-router and disk manager. The operating environment was designed especially for Omega DBMS to provide a suitable operating system support.

The *topology module* forms a view of the Omega hardware architecture as a set of Ω-clusters. It supports a hierarchical addressing for Omega system nodes (virtual processor modules). The address consists of a number of the Ω-cluster within the Omega system and a number of the node within the Ω-cluster. The information about a particular MVS-100 topological structure is read from a configuration file during initial system loading.

Threads manager implements a support for the light-weighted processes (threads). Two types of threads are supported: system threads and user threads. The *system threads* are generated in the Omega operating environment. There are at least three system threads in Omega system: Ω-conductor, Ω-router and disk manager. The *user threads* are generated in the DBMS kernel. An original supplier/consumer model described in [13] is used for the threads dispatching.

The Ω-conductor implements a message-passing system for intra-cluster communication. The Ω-router implements a message-passing system for inter-cluster communication.

The disk manager implements a page-level interface with the disk subsystem unit.

The operating environment abstracts the Omega DBMS kernel from the details of the hardware implementation. It's a base for Omega system portability to MVS-1000 and another hardware platform.

3. Ω-Conductor

Ω-conductor has to provide a facility for effective interprocessor communication within the Ω-cluster. In this case it has to utilize closely-coupled predefined Ω-cluster structure. Ω-conductor has to provide an asynchronous exchange mode which is suitable for multithread environment. According to these objectives we proposed the following model for the Ω-conductor.

Each two nodes inside the cluster can be connected by any number of one-directional channels. These channels can perform data transfer independently in asynchronous mode. The channel is unambiguously identified by port number (one channel has to have equal port numbers on each two nodes which it connects). To connect two nodes each of them has to perform appropriate set up operations.

The interface of the Ω-conductor includes the following major functions.

```
/* Conductor initialization */
void cn_init();
```

Function **cn_init()** performs an initialization of an internal system variables and runs the Ω-conductor system thread. Conductor initialization has to be performed before any call of another conductor function.

```
/* To run an exchange by the channel */
int /*channel ID*/ cn_runChnl(
   int,      /* recipient node number within the cluster */
   int,      /* port number */
   int,      /* channel type (0 – read; 1 – write) */
   void*,    /* data buffer */
   int       /* buffer length */ );
```

Function **cn_runChnl**() create one-directional channel between current node and recipient node, associated with specified port number. The data transfer can be done only if the recipient node runs an exchange by the channel has complementary type and the same port number. Waiting for response and data exchange are executed in asynchronous mode.

```
/* To test, if the data exchange is done */
int /* 1 – done, 0 – not done */
cn_testChnl(int/* channel ID */);
```

Function **cn_testChnl()** performs a check whether the data transfer by the channel is done.

In the Ω-conductor implementation we used the following message format. Every message consists of two parts: a message header and informational part "info". To pass the message we have primarily to pass the header and then to pass the info as separate part. The message header has the following fields: a sender node number; a port number and an info length. If info length equals zero the message hasn't info part. Info part has no a structure.

When the **cn_runChnl**() operation is performed a channel descriptor record is appended to the channel table. This table is organized as a queue and has the following fields: a channel identifier; a recipient node number; a port number; a channel type (read or write); a channel state; an info buffer pointer; an info length. In case of "write" type of the channel, after the descriptor record has been made a zero length message is sent to the recipient node. This message is interpreted by the recipient node as a request to set up a channel for passing a data. This request is registered in the recipient node channel table as a new channel descriptor record (without an input buffer and info length assignment).

When the corresponding **cn_runChnl**() operation has been processed in the recipient node, the input buffer and info length fields are determined for the corresponding channel descriptor record. After that, the recipient node sends a zero length message to the initiating node. The initiating node interprets this message as a readiness of the recipient node to accept a data. After that the initiating node locks the corresponding hardware link by writing for another channels on this node. Then one sends a message heading (with nonzero info length) to the recipient node. When heading transfer done the initiating node immediately passes the info part. Seeing the link is write-protected, none another channel can send by this link any information between those two transfers. When info transfer done, the initiating node unlocks the link by writing and removes the corresponding channel descriptor record from the channel table.

When the recipient node has received a message heading with nonzero info length, it locks the corresponding hardware link by reading for another channels on this node and waits for the info part to come. When the info part has come, it is directed to assigned buffer. After that the recipient node unlocks the link by reading and removes the corresponding channel descriptor record from the channel table.

In described protocol, one message transfer takes four elementary interchanges and none memory-to-memory data copying. In case of intra-cluster exchange this protocol appears as efficient enough because elementary interchange time is relatively low within the cluster. Another important property of Ω-conductor is that it doesn't involve a dynamic memory allocation for new channel generating.

4. Ω-Router

Ω-router has to provide a facility for effective inter-cluster communication within the Ω system. On the one hand, elementary interchange can take a long time in this case. On the other hand the inter-cluster interchanges are relatively rare.

The external model of the Ω-router is similar to one for Ω-conductor. But Ω-router uses quite different internal protocol.

The interface of the Ω-router includes the following major functions.

```
/* Router initialization */
void rt_init();
```

Function **rt_init()** performs an initialization of an internal system variables and runs the Ω-router system thread. Router initialization has to be performed before any call of another router function.

```
/* To generate new writing channel */
int /*channel ID*/ rt_newWrite(
  int,   /* recipient cluster number within the system*/
  int,   /* recipient node number within the cluster */
  int,   /* port number */
  int    /* message length */ );
```

Function **rt_newWrite**() create new writing channel associated with specified port number. The channel can be set up only if the recipient node has perform a corresponding **rt_newRead**() operation. By the channel we can write any number of messages with length equals *infoLen*.

```
/* To generate new reading channel */
int /*channel ID*/ rt_newRead(
  int,   /* transmitter cluster number within the system*/
  int,   /* transmitter node number within the cluster */
  int,   /* port number */
  int    /* message length */ );
```

Function **rt_newRead**() create new reading channel associated with specified port number. The channel can be set up only if the transmitter node has perform a corresponding **rt_newWrite**() operation. By the channel we can read any number of messages with length equals *infoLen*.

```
/* To write a message */
int /* 1 – done, 0 – not done */ rt_write(
  int      /* channel ID */
  void*,   /* data buffer */ );
```

Function **rt_write()** transfers the message from specified buffer by specified channel. If the functions returns zero, you have to repeat the last transfer.

```
/* To read a message */
int /* 1 – done, 0 – not done */ rt_read(
  int      /* channel ID */
  void*,   /* data buffer */ );
```

Function **rt_read()** reads the message from specified channel to specified buffer. If the function returns zero, you have to repeat the last reading.

In the Ω-router implementation we used the following message format. Every message consists of two parts: a message header and informational part "info". To pass the message we have primarily to pass the header and then to pass the info as separate part. The message header has the following fields: an absolute number of the sender node; a port number and a info length. Info part has no a structure.

When the **rt_newWrite**() or **rt_newRead**() operation is performed, a channel descriptor record is appended to the channel table. This table is organized as a queue and has the following fields: a channel identifier; a recipient node absolute number; a port number; a channel type (read or write); a channel state; a buffer pool pointer; a message length. Every channel is associated with internal buffer pool. This buffer pool is generated by a dynamic memory allocation.

When a **rt_write**() operation is performed, the specified message is simply copied to the buffer pool. The Ω-router system thread serves this buffer pool as a queue and transfers first message from the queue to recipient node. The recipient node puts received message to the input buffer pool. For every *N/2* interchanges (*N* equals the buffer pool length) the Ω-router sends a zero length message to the recipient node. The recipient node interprets this message, as a request to confirm that 50% of input buffer pool is empty. If it's true, the recipient node returns a message header with nonzero info length. Otherwise it returns a message header with zero info length. In the last case, the recipient node blocks the channel for writing until it has received a message header with nonzero info length.

When a **rt_read**() operation is performed, the message is simply read from the buffer pool. If the buffer pool is empty the **rt_read**() returns 0.

In described protocol, one message transfer takes only two interchanges. It's especially important in case when the interchange takes a long time. In contrast with Ω-conductor, Ω-router involves memory-to-memory data copying and dynamic memory allocation. Nevertheless the last two points are not critical because the inter-cluster exchange is relatively rare.

5. Conclusion

In this paper we described an original hierarchical hardware architecture for Omega parallel DBMS. This architecture is well accorded for MVS-100 massively-parallel computer system.

The software architecture of Omega prototype was briefly discussed. Three levels of abstraction were introduced.

Interface, implementation and transfer protocols of intra-cluster message exchange subsystem were described. This

subsystem was called Ω-conductor. Ω-conductor demonstrated well performance on intensive data exchanges between nodes within one cluster.

We described interface, implementation and transfer protocols of Ω-router. It's a subsystem carries out the inter-cluster message exchange. Ω-router demonstrated well performance on data exchanges between nodes from remote clusters.

Both subsystems were implemented in C programming language for MVS-100 parallel supercomputer.

References

1. Stonebraker M. "The case for shared nothing". *Database Engineering*. March 1986; 9 (1).
2. DeWitt D.J., et al. "The Gamma database machine project". *IEEE Trans. on Knowledge and Data Engineering*. March 1990; 2 (1): 44-62.
3. Boral H, et al. "Prototyping Bubba: a Highly Parallel Database System". *IEEE Trans. on Knowledge and Data Engineering*. March 1990; 2 (1): 4-24.
4. Lorie R., et al. "Adding Intra-Parallelism to an Existing DBMS: Early Experience". *IEEE Bull. on Database Engineering*. March 1989; 12 (1): 2-8.
5. Baru C. K., et al. "DB2 Parallel Edition". *IBM System Journal*. 1995; 34 (2): 292-322.
6. Teradata Corporation. DBC/1012 "Data Base Computer Concepts & Facilities". *Document No. C02-0001-00*. 1983.
7. DeWitt D., Gray J. "Parallel Database Systems: The Future of High-Performance Database Systems". *CACM*. June 1992; 35 (6): 85-98.
8. Valduries P. "Parallel Database Systems: the case for shared-something". In: *Proc. of 9th International Conference on Data Engineering*. Vienna, Austria. April 19-20, 1993, pp 460-465.
9. Bouganim L., Florescu D., Valduriez P. "Dynamic Load Balancing in Hierarchical Parallel Database Systems". In: *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, Mumbai (Bombay), India. September 1996, pp 436-447.
10. Sokolinsky L., Axenov O., Gutova S. Omega: "The Highly Parallel Database System Project". In: *Proceedings of the First East-European Symposium on Advances in Database and Information Systems (ADBIS'97)*, St.-Petersburg. September 2-5, 1997, vol. 2, pp 88-90.
11. Zabrodin A.V., Levin V.K., Korneev V.V. "The Massively Parallel Computer System MBC-100". In: *Proceedings of PaCT-95*, 1995, pp 342-356 *(Lecture Notes in Computer Science*, vol. 964)
12. Stonebraker M. "Operating System Support for Database Management" *CACM*. July 1981; 24 (7): pp 412-418.
13. Sokolinsky L. B. "A Light-weighted Process Support in the Omega Parallel DBMS for MVS-100 Multiprocessor Computer System". In: *Fundamental and applied aspects of the large distributed program systems design: Proc. of the All-Russian scient. conf*, Novorossiysk, September 21-26, 1998, pp 132-138 (In Russian).