Implementation and Experiments of an Extensible Parallel Processing System Supporting User Defined Database Operations

Yasushi KIYOKI, Takahiro KUROSAWA, Peng LIU
Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba, Ibaraki 305, JAPAN
E-MAIL: kiyoki%is.tsukuba.junet%japan@RELAY.CS.NET

Kazuhiko KATO, Takashi MASUDA
Department of Information Science, Faculty of Science
University of Tokyo, 7-3-1 Hongo, Bunkyo-ku 113, Tokyo, JAPAN

Abstract

This paper presents an implementation method and experimental results of an extensible parallel processing system for databases. We have already proposed a stream-oriented parallel processing scheme (stream-oriented scheme) of basic operations for databases and knowledge bases. This scheme is based on the demand-driven evaluation incorporating stream processing.

We have designed basic primitives as a set of basic facilities for implementing the stream-oriented scheme. By using these basic primitives, arbitrary basic operations for a wide variety of database applications can be described and executed in parallel. In this paper, we present an implementation method of these basic primitives. This method is used to implement the stream-oriented scheme in parallel processing environments in which message passing is used for interprocessor communication. This paper also shows several experimental results of actual query processing in a parallel processing environment in which multiple conventional processors are loosely connected to a high speed network.

1 Introduction

Relational database systems usually support a fixed set of basic operations called relational database operations. For supporting database applications, such as knowledge bases and engineering applications, it is necessary to allow the database administrator or user to specify new basic operations and data types, and to integrate them into the system. The key issue is to provide the administrator or user with the facilities to implement specific operations and data types and to enable them to be integrated into the system.

Furthermore, those operations should be executed efficiently. To enhance the processing performance of relational database operations, many algorithms and database machine architectures have been proposed (see e.g.,[2,3]). In general, these algorithms and architectures have been designed to execute relational database operations efficiently. That is, these algorithms and architectures have been oriented for a fixed set of operations.

For supporting advanced database applications, database systems should have facilities to execute arbitrary basic operations efficiently. We have proposed a stream-oriented parallel processing scheme (stream-oriented scheme) for basic operations of relational databases [4] and inference operations for deductive databases[5]. This scheme is based on the demand-driven evaluation[11,12] incorporating stream processing. We have designed basic primitives as a set of basic facilities for implementing the stream-oriented scheme[4]. By using these basic primitives, arbitrary basic operations for a wide variety of advanced database applications can be described and executed in parallel.

Two parallel processing environments are considered for implementing the stream-oriented parallel processing scheme as follows:

(1) Multiple conventional processors are connected to a high speed network. In this environment, message passing is performed to communicate stream elements and demands among multiple processors.

(2) Multiple processors are tightly connected through shared memory. In this environment, stream elements and demands are sent and received through shared memory that is allocated among multiple processors.

We have implemented the basic primitives of the extensible parallel processing system SMASH in the environment (1). In this paper, we present an implementation method of the basic primitives. In this implementation method, the communication between processors is performed by using the message passing mechanism. We have also performed several experiments of parallel query processing on the implemented system. We present several experimental results and discuss the efficiency of our implementation method and system.

2 Stream-oriented parallel processing

2.1 The Stream-oriented scheme

In this section, the stream-oriented scheme described in [4] is briefly reviewed. In our parallel processing scheme, the demand-driven evaluation strategy is used to exploit parallelism in processing queries for databases. In this scheme, independent function nodes, which are allocated to different processors, are executed in parallel. Furthermore, the *stream-oriented parallelism* is also exploited between function nodes (e.g. relational operation nodes) which are served as producers and consumers of intermediate streams. A function node, which consumes intermediate stream, is referred to as a "consumer node", and a function node, which produces intermediate stream, is referred to as a "producer node".

When a consumer node completes the processing of a grain (or a page) of stream elements which corresponds to its input buffer size, the node issues a demand to the producer node in order to have the input buffer refilled with the next grain of stream elements.

When a producer node receives a demand from the consumer node, the consumer node accesses a grain of stream elements from its input buffer and it executes the computation until the node has completed the production of one grain of resulting stream elements in the output buffer. The output buffer is then treated as the input buffer for the consumer node. Once a grain is accessed by a consumer node, the grain can be removed from the buffer.

Not every function node creates a whole intermediate stream for a single demand. Each node creates only one grain of stream elements for a single demand. Therefore, each buffer does not need to have a capacity for storing the whole intermediate stream, that is, the buffer size is not related to the size of the intermediate stream. The size of the buffer corresponds to grain size. The grain size differs among function nodes, that means, each grain size is fixed according to the size of the each buffer allocated to each function nodes.

If both the producer node and the consumer node of an intermediate stream are allocated to different processors, the double buffering mechanism is supported in a buffer between them, in order to exploit the stream-oriented parallelism.

In terms of references to the same stream, two reference methods named "re-computation method" and "caching method" are supported and used alternatively. In the re-computation method, if a stream is referenced more than once, the corresponding stream is reproduced each time it is referenced. In this case, each stream element can be deleted just after a reference to it is completed. If this method is employed, the computation to the stream can be performed within limited memory resources. However, when the same stream is referenced more than once, it must be reproduced, that is, the computation which produces the stream must be re-executed.

On the other hand, in the caching method, a stream is produced only once when the first reference to it is performed. The produced stream is then used in the other references to the stream. In this method, the stream must be retained until every reference to it is completed. If the stream is huge, like a stream of tuples in a relation, it seems that the memory could be swamped. However, reproduction of the same stream is unnecessary.

2.2 Basic primitives

We have designed the basic primitives, as basic facilities to implement the stream-oriented scheme. A set of our basic primitives has been presented in [4] in detail. Each basic primitive is independent of hardware architectures, on which the basic primitives are implemented. Furthermore, each basic primitive is also independent of algorithms of basic operations for databases and knowledge bases. This enables the basic primitives to be used in describing arbitrary basic operations for databases and knowledge bases. Basic primitives exploit parallelism inherent in arbitrary basic operations by incorporating the demand-driven evaluation with stream processing. In our system, an arbitrary basic operation is defined as a function, and that function is transformed into the procedural object codes which include basic primitives. A function activated at execution time is referred to as a function instance. Each function instance is allocated to one of the multiple sites, and parallel processing is then performed among function instances which are allocated to different sites.

Basic primitives are classified as follows:

(1) The creation of a function instance and a channel. (primitives: new , channel)

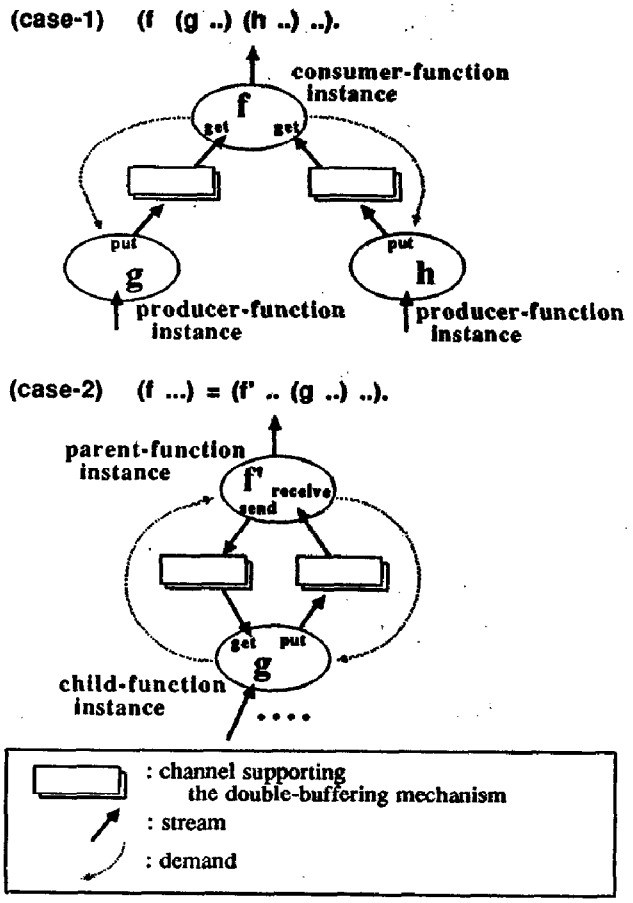


Figure 1: Sending and Receiving of stream elements and demands

The primitive "new" creates a specified function instance on the specified site. The primitive "channel" creates a channel, which connects to function instances to facilitate the passing and receiving of stream elements and demands between function instances. Every channel includes a buffer to store a grain of stream elements whose size is specified as "granularity".

(2) The sending and receiving of stream elements and demands. (primitives: get, put, send, receive, pre-demand)

There are two cases for transferring a stream between function instances ("f" and "g").

In the case that the output stream of a function instance ("g" or "h") becomes the input stream of another function instance ("f") as an actual argument, the function applications are represented in case-1 of Fig. 1.

In this case, the primitive "get" is used in the consumer function ("f") to fetch a stream element from the buffer of the channel allocated between two function instances ("f" and "g"). When the buffer is vacant, this primitive issues a demand to the producer function ("g"). Then, it waits until the buffer is refilled with the next grain of stream elements. At this time, if the double-buffering mechanism is supported in the buffer, as discussed in section 2.1, and the next grain has already been produced due to the pre-issued demand, this primitive be-

gins to fetch a stream element from the other area of the buffer. In this case, stream-oriented parallelism is exploited. When the producer function ("g") receives a demand, it stores a stream element in the buffer by using the primitive "put". Until the buffer is filled this primitive stores stream elements. Then, the primitive waits for its next demand.

In the case that a function instance ("f") invokes another function instance ("g") in the function body by using the primitives "new" and "channel", the function applications are represented in case-2 of Fig. 1

In this case, primitives "send" and "receive" are used in the function instance "f". In the function instance "f", to pass a stream element to the child function instance "g", the primitive "send" stores the stream element in the buffer. This primitive does not wait for the next demand when the buffer is filled with stream elements. Until the next demand arrives, this primitive does not store a stream element, that is, this primitive is ignored at the execution of the function instance "f". In the function instance "f", the primitive "receive" receives a stream element from the buffer as a part of a return value of the child function instance "g". When the buffer becomes vacant, this primitive issues a demand to the child function instance "g". However, at this time, this primitive does not wait for the next grain of stream elements. This primitive is ignored at the execution of function instance "f", until the next grain of stream elements is stored into the buffer by the child function instance.

The basic primitive "pre-demand" is used when the double buffering mechanism is supported in the buffer of a channel. In this case, stream-oriented parallelism is exploited between function instances which are allocated to different sites. For the producer function instance of a stream, this primitive is issued only once at the beginning of execution of the consumer function so as to have the producer function create the first grain of stream elements eagerly.

(3) The implementation of "re-computation method" and "caching method". (primitive: rewind)

As reference methods to the same stream, "re-computation method" and "caching method" are alternatively used in our system. In supporting the re-computation method, the primitive "rewind" is used to request the re-production of the stream to the producer function instance. This primitive initializes the producer function instance and the channel allocated between these function instances. If the caching method is specified, the re-computation of the stream is not necessary, but the intermediate stream elements must be stored in the caching directory to cope with the next references to the same stream.

(4) Nondeterministic selection of a single stream element from multiple streams. (primitives: select, disable, enable)

By using the concept of "nondeterminism", when a function instance sends and receives several streams and demands through channels, higher parallelism is exploited. That is, when the function instance sends and receives several streams and demands, a higher degree of parallelism can be exploited by choosing one of the demands or one of the streams in the nondeterministic way.

The primitive "select" chooses one of channels, which are connected to the function instance. In the case of this primitive not being supported, some primitives in the function instance may wait for the arrival of stream elements or a demand at a channel, which is not activated due to the delay of the arrival of stream elements. In this case, the execution of the function instance is suspended and, as a result, the parallelism may decrease.

The primitive "select" is used incorporating with the primitives "disable" and "enable". The primitive "disable" removes the specified channel from candidates for the nondeterministic selection. The primitive "enable" adds the specified channel to candidates for the nondeterministic selection.

3 An implementation method

In this section, we present an implementation method of the basic primitives. This implementation method is thought of as being novel, for implementing parallel processing environments for databases by using multiple conventional processors connected to a high speed network. Using only conventional processors and networks, arbitrary basic operations for a wide variety of database applications can be executed in parallel.

In this method, communication of stream elements and demands is performed by using message-passing between function instances allocated to different sites. On the other hand, stream elements and demands are sent or received through the shared segment of a memory, between function instances which are allocated to the same site.

When several function instances are allocated to the same site, the stream-oriented scheme enables those function instances to be executed in pseudo parallel within limited buffer resources. Two approaches to the implementation of a single site have been considered. First approach is based on the method that the function instances allocated to a single site are activated and executed within a single process. In this approach, stream elements and demands are transferred within the single process. The kernel in the process manages the transition of control to one of executable function instances. We adopt this approach to the implementation of a single site. The other approach is based on the method that a single process is assigned to each function instance. In general, communication and switching among processes may cause overhead, because the transition of control is managed by the operating system. In this case, since stream elements and demands are transferred by communication and switching among processes, the overhead of communication and switching becomes unacceptable. Therefore, we do not employ this approach.

Each site consists of *a communication process (CP)* and *a functional computation process (FCP)* as shown in Fig. 2. At each site, only two processes are required to execute the basic operations and to communicate the stream elements and demands among the sites. Function instances and channels are created within the FCP. The message passing mechanism is implemented at CP to transfer stream elements and demands between the sites.

3.1 Kernel

A kernel is implemented at each site in order to manage communication among function instances, and to schedule the execution order of function instances that are allocated to a site. One kernel is provided for each site, and its facilities are separately implemented in FCP and CP as follows:

(1) creation of function instances.(FCP)

(2) creation of channels.(FCP)

(3) scheduling for pseudo parallel execution of function instances allocated to the single site.(FCP)

(4) control of interprocessor communication.(CP)

The interpretation of basic primitives, "new" and "channel", corresponds to (1) and (2), respectively. As (3), the kernel controls the execution of function instances which are allocated to

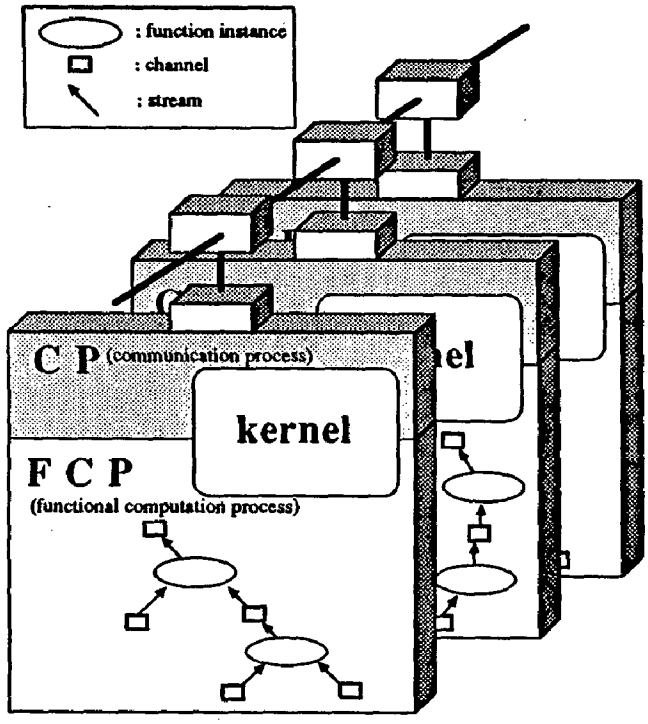


Figure 2: An overview of the system

the same site. In our implementation method, several function instances can be allocated to the same site. This allocation enables a query to be executed within the limited processor-resources, allowing the function instances in the same site to be executed in pseudo parallel as coroutines. The kernel obtains control when a producer function instance, which has completed producing a grain of stream elements, has suspended and is waiting for a next demand, or when a consumer function instance has suspended and is waiting for the next grain of stream elements. When the scheduler is activated by the kernel, it selects one of the executable function instances, and transfers control to it.

In terms of (4), when function instances are allocated to different sites, the interprocessor communication is required to transfer demands and stream elements. At each site, for each interprocessor communication request, the kernel facility in the CP sends a grain of stream elements or a demand to the destination site through the network.

3.2 Function instance

Each function instance is sequentially executed, but parallelisms can be exploited among function instances. Each function instance receives one or several streams, and creates a stream as a return value. Stream elements and demands are transferred through a "channel" which is allocated between function instances.

Each function instance transits states as follows:

1. Executing State:
   This is the state that the function instance is executing. From this state, it transits to the waiting state when one of the following conditions occurs.
   (1) When the function instance completes producing a grain of stream elements, it transits to the waiting state to wait for the next demand. In this case, the waiting state is called "Waiting for the next demand".
   (2) When it completes consuming a grain of stream elements, it transits to the waiting state to wait for the next grain of stream elements. In this case, the waiting state is called "Waiting for the next grain of stream elements".
2. Waiting for the next demand:
   When the function instance transits to the waiting state due to the condition (1), it waits for the next demand issued by its consumer function instance. When this function instance receives a demand, it transits to the Ready state.
3. Waiting for the next grain of stream elements:
   When the function instance transits to the waiting state due to the condition (2), it waits for a grain of stream elements which has been produced by the producer function instance, after issuing the demand. In this state, when the function instance receives the grain, it transits to the Ready state.
4. Ready state:
   If several function instances are allocated to the same site, the function instance transited from the waiting state cannot be always executed immediately. First, the function instance transits to the Ready state, but not to the Executing state. From the Ready state, the function instance, which has obtained control from the scheduler, transits to the Executing state. Each kernel includes a scheduler. Then, the scheduler searches for the function instances which are ready to be activated, and gives control to one of these function instances. Currently, round-robin scheduling is employed.

3.3 Channel

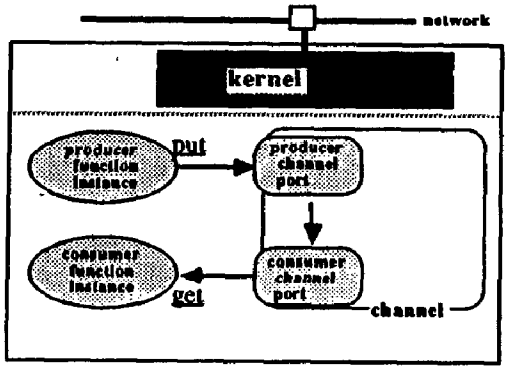
Function instances use a "channel" to transfer demands and stream elements between them, as shown in Fig. 3. The channel has a communication buffer for transferring grains of stream elements and demands. By using the channel, communication between function instances can be performed *transparently*. A channel has two main roles; one is to be the interface to a function instance as a "channel port", the other is to perform communication between channel ports in a single channel.

(1) channel port:
A channel port is the interface to a function instance. We call the channel port connected to the producer function instance *the producer channel port*. We call the channel port connected to the consumer function instance *the consumer channel port*. Each channel port is allocated to the site where the function instance, which is connected to it, is allocated. Communication between function instances is performed through the producer and consumer channel ports in a single channel according to requests of demand or data transfers.

(2) communication between channel ports:
When function instances connected to a channel are allocated to single site, or to different sites, they are executed in pseudo-parallel, or in parallel, respectively. The communication between producer and consumer channel ports is performed as follows:

(a) channel port communication within a single site:
When the producer and consumer channel ports are allocated to the same site, a shared segment of memory is used to communicate stream elements and demands as shown in Fig. 3-(a). In our method, since the function instances allocated to the same site behave as coroutines,

(a) channel port communication within a single site.



(b) channel port communication between different sites.

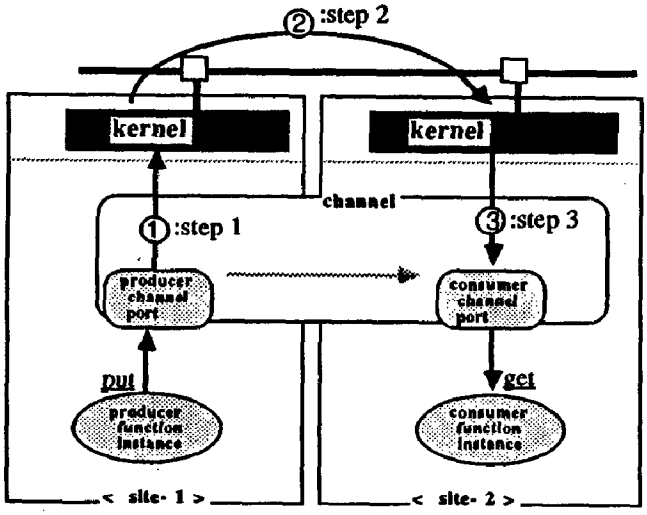


Figure 3: Communication between function instances

the execution and suspension of each function instance is managed under the control of a scheduler, thus the mutual exclusion can be performed. Accordingly, the accesses to the shared segment of memory by producer and consumer function instances are serialized and performed without conflict.

(b) channel port communication between different sites: When the producer and consumer channel ports are allocated to different sites, message-passing is performed to communicate stream elements and demands. Stream elements and demands are transferred as message packets. The interprocessor communication is performed as shown in Fig. 3-(b).

step 1:
By the function instance in the FCP, stream elements and demands are stored in the shared segment of memory that has been allocated between the FCP and the CP. A large amount of data may be transferred between the FCP and the CP. Therefore, we use communication with shared segments. Because the FCP and the CP are in the same site, they can communicate with shared segments. This method does not need to call the operating system for each communication request, so the overhead of communication is relatively small. (Currently, we are also implementing the parallel processing environment with the tightly-coupled multiprocessors with shared memory. In this environment, we utilize the communication method using shared segments for transferring stream elements and demands among processors.) Accesses to shared segments are serialized by ordering the access requests incorporating a semaphore mechanism.

step 2:
The kernel facility in the CP creates a packet including a grain of stream elements or a demand, which are stored in the shared segment, it then transfers the packet to the CP of the destination site.

step 3:
In the destination site, the CP stores the received stream elements or a received demand into the shared segment of memory which is allocated between the FCP and the CP. Then, the function instance receives the grain of stream elements or the demand.

3.4 Interprocessor communication

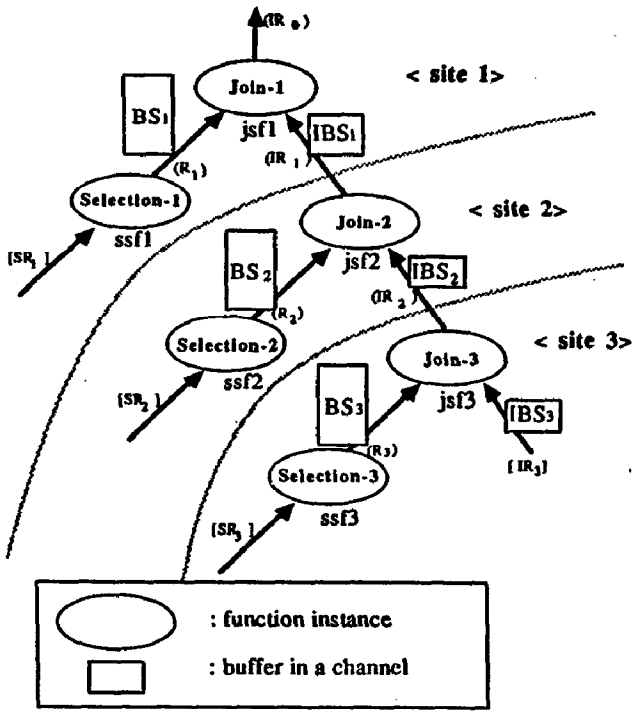
We have implemented a parallel processing system SMASH on the basis of the implementation method discussed above. The current hardware environment consists of a number of Sun-3 workstations(Unix 4.2BSD [10]) connected to the Ethernet [10]. We use facilities of interprocessor communication which are supported in Unix.

We have chosen the UDP [10] as the communication protocol among the sites. In the communicating model of UDP, interprocessor communication is performed through a shared bus. In comparing to the communication protocol TCP [10], we explain the reasons why we have adopted the UDP.

1. Since the UDP is a lower level communication protocol than the TCP, its communication can be performed more efficiently than that of the TCP. That is, since the UDP is simple protocol, its overhead for network communication is small, and it is advantageous for transferring a large amount of data, such as databases.
2. In the UDP, a processor can dynamically select the communication partner when it receives a request for communication. That is , unlike the TCP, it is not necessary to statically fix a one-to-one virtual circuit between the processors. On the other hand, the TCP is based on the communication model of *Client and Server*. This forces the processors to communicate with each other as a master and a servant. However, in our implementation method, there is no special processor such as a supervisor, and parallel processing is performed by communicating with processors of the same ability. Therefore, the UDP is suited to our method.
3. Communication in the UDP is less reliable than that in the TCP. Therefore, the reliability of communication must be assured within the framework of our implementation. In our system, the CP supports reliable communication. On the other hand, the TCP has a higher reliability than the UDP. However, this higher reliability in the TCP often causes heavy overhead, especially, for applications of database processing which are required to manipulate large amounts of data. In comparing the TCP with the UDP, we found the UDP to be more efficient.

4 Experiments and results

We have performed several experiments on the experimental system for examining the effectiveness of our implementation



BS_1, BS_2, BS_3 : *outer−relation buffer size (tuples)*
IBS_1, IBS_2, IBS_3 : *inner−relation buffer size (tuples)*
SR_1, SR_2, SR_3, IR_3 : *base−relation size (tuples)*
R_1, R_2, R_3 : *outer−relation size (tuples)*
IR_1, IR_2, IR_3 : *inner−relation size (tuples)*
IR_0 : *result relation size (tuples)*
jsf_1, jsf_2, jsf_3 : *join selectivity factor*
ssf_1, ssf_2, ssf_3 : *selection selectivity factor*

(intermediate relation size =
jsf * (inner-relation size * outer-relation size))
(intermediate relation size = ssf * base-relation size)

Figure 4: The type of query for experiments

method presented in this paper. In addition, we have measured the execution time of several queries. The execution time includes the actual CPU time, system call time for disk I/O processing, and communication time among sites.

4.1 Queries for experiments

Our system can support a wide variety of database operations. Queries consisting of relational database operations are used, as examples. We utilize four simple queries, with the same structure, consisting of three join and three selection operations, as shown in Fig. 4. Selection and join nodes are allocated to different sites, and executed in parallel by the stream-oriented scheme. In each join node, the tuples of the grain stored in the outer-relation buffer are sorted on joining attribute(s), and each tuple in the inner-relation buffer is compared with the sorted tuples by using the *binary-search* algorithm. The operand base-relation of each selection node is stored in the site where the node is allocated.

Table 1: Base-relation sizes and selectivity factors in the evaluated queries

		Query1	*Query2*	*Query3*	*Query4*
Selection−3	ssf_3	1/10	1/10	1/10	1/10
	SR_3	10240	10240	10240	10240
Join−3	jsf_3	1/1024	1/512	1/512	1/2048
	R_3	1024	1024	1024	1024
	IR_3	1024	1024	1024	1024
Selection−2	ssf_2	1/10	1/10	1/10	1/10
	SR_2	10240	10240	10240	10240
Join−2	jsf_2	1/1024	1/1024	1/512	1/2048
	R_2	1024	1024	1024	1024
	IR_2	1024	2048	2048	512
Selection−1	ssf_1	1/10	1/10	1/10	1/10
	SR_1	10240	10240	10240	10240
Join−1	jsf_1	1/1024	1/1024	1/512	1/2048
	R_1	1024	1024	1024	1024
	IR_1	1024	2048	4096	256
output	IR_0	1024	2048	8192	128

Table 1 shows the parameter settings of relation sizes and selectivity factors [7,9,13] for each query.

In *Query 1*, the size of the intermediate relation produced by each join node is the same as the size of the operand inner-relation. In *Query 2*, *Join-3* produces an intermediate relation with twice the size of the operand inner-relation. *Join-2* and *Join-1* produce relations of the same size as that of the intermediate relation produced by *Join-3*. In *Query 3*, each node (*Join-1*, *Join-2*, *Join-3*) produces a relation with twice the size of the operand inner-relation. In *Query 4*, each node produces a relation with half the size of the operand inner-relation.

In the stream-oriented scheme, if the selectivity factors and the buffer sizes are kept constant, the frequency of communications, the amount of transfer data and the comparison times between pages will remain constant. Although the total execution time becomes longer with the increase of base-relation size, neither parallelism nor effect on communication traffic changes. Therefore, we did not perform experiments on various settings of base-relation sizes.

We considered the queries with the same base-relation sizes and different join selectivity factors. Tuples in all the relations are 64 bytes long, each including 4 byte integer attributes as selection and joining attributes. All of the integer attributes have uniformly distributed values, but the range of their distributions varies to generate different join selectivity factors.

4.2 Experimental results

In these experiments, the re-computation method has been used as the parameter passing method in the stream-oriented scheme. In employing re-computation method, the total number of computations is dependent only on the size of the outer-relation buffer which has been allocated to each operation node. The outer-relation buffer size has an effect not only on the number of computations, but on the parallelism between the operations which are being executed at different sites. Therefore, it is very important to allocate available buffer resources optimally to each outer-relation buffer. In parallel processing cases, the size of the grain being transferred between sites also affects parallelism. In the experiments, this grain size corresponds to the inner-relation buffer size. Furthermore, buffer

sizes for base relations affect the number of disk I/O operations. Therefore, it is important to optimally allocate buffer resources to inner-relation buffers and buffers for base relations. That is, the limited buffer resources must optimally be allocated to each buffer for each operation node. In [6,7,8], for implementing the stream-oriented scheme in sequential, parallel and distributed processing environments, we have presented several algorithms to optimally allocate the buffer resources to each buffer. These algorithms are used to minimize the number of computations and disk I/O operations, and to exploit parallelism in query processing. The experiments discussed in this section have been focused on the investigation of the effects of the outer-relation buffer size and the inner-relation buffer size on the number of computations and parallelisms.

4.2.1 Effects of buffer resources

Three processors(sites) were used to execute each query. To store outer-relation tuples of each join operation, the same amount of buffer resources was allocated to each site, that is, $BS_1 = BS_2 = BS_3$. Fig. 5 shows the execution time in parallel processing of *Query 1, 2, 3* and *4* in varying the outer-relation buffer size.

The execution time is shortened with the increase of the outer-relation buffer size (BS_1, BS_2, BS_3). This is because, the increase of the outer-relation buffer size decreases the number of computations in query processing. Execution time is shortened discontinuously with the increase of the buffer size. The number of re-computations of the join operation does not continuously change with the increase of the buffer size. In the range of the buffer sizes with which the number of re-computations does not change, the execution time of a query is not always shortened with the increase of the buffer size.

4.2.2 Effects of the outer-relation buffer size on parallelism

Resource allocation to the outer-relation buffer of each join node gives a significant effect on parallelism inherent in a query. In [6], we have introduced a criterion for resource allocation to exploit the stream-oriented parallelism inherent in the operation nodes. Here, we briefly review the criterion. We consider parallelism inherent between two nodes (*Join-2,Join-3*) in Fig. 4. It is assumed that two operation nodes are being executed by different sites.

If *Join-2* performs the comparisons between two operand grains currently stored in the input buffers (BS_2, IBS_2), and *Join-3* completes producing the next inner-relation grain during the comparisons in *Join-2*, the pipeline delay is eliminated. (That is, the stream-oriented parallelism is exploited.) In other words, a suspension of operation execution causes a pipeline delay, which occurs in the absence of the next inner-relation grain.

The number of comparisons between an inner-relation grain IBS_2 and a sorted outer-relation grain BS_2 in *Join-2* is

$$IBS_2 * (\log_2 BS_2 + jsf_2 * BS_2). \quad (1)$$

The number of comparisons in *Join-3*, to produce an inner-relation grain (IBS_2) of *Join-2* by using the *binary-search* algorithm ($1/jsf_2$ is the average number of comparisons required to generate one output tuple by the *nested-loop*) is

$$IBS_2 * (\log_2 BS_3 + jsf_3 * BS_3)/(jsf_3 * BS_3). \quad (2)$$

The criterion to execute *Join-2* continuously without pipeline delay is as follows:

(1) $\geq$ (2).

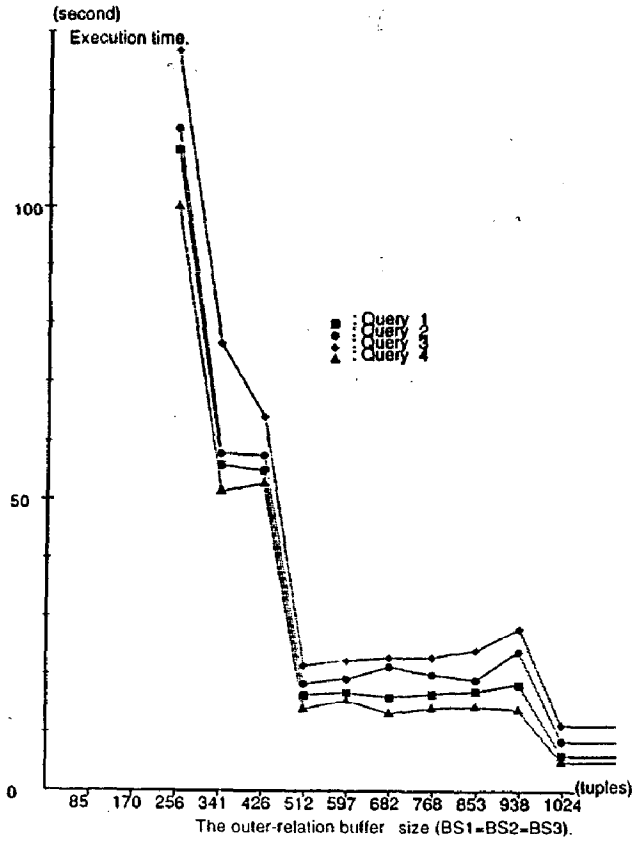


Figure 5: Relationship between the outer-relation buffer size and execution time.

If this criterion is satisfied in any adjacent nodes served as producer and consumer nodes at different sites, the root-node of a query tree (e.g. *Join-1*) can be executed without the suspension.

Fig. 5 shows the execution time of the queries *Query 1, 2, 3* and *4*. In Fig. 5, with the size (1024 tuples) of outer-relation buffer resources available at each site, the criterion for exploiting parallelism mentioned above is satisfactory in executing *Query 1,2,* and *3*. In this size, there is no need to perform the re-computation for the inner-relations in executing *Query 1,2,3* and *4* because the buffer size is large enough to contain all the tuples of the outer-relations. As a result, the best execution efficiency is shown.

The increase of the outer-relation buffer size contributes both to reducing the number of computations and to satisfying the criterion for exploiting parallelism.

4.2.3 Effects of communication granularity on parallelism

The size of the inner-relation buffer does not influence the number of computations in query processing. However, it influences the number of times it takes to transfer stream elements and the number of times it takes to issue demands. We varied the size $IBS_i (i = 1, 2, 3)$ of the inner-relation buffer in *Query 1, 2, 3,* and *4* to examine its effects on the number of times it takes to transfer packets of stream elements, and the number of times it takes to issue demands. The change of IBS_i corre-

sponds to that of granularity for message-passing. Fig. 6 shows the relationship between the execution time and the alternation of the inner-relation buffer size, when all the tuples of the outer-relations are contained in the outer-relation buffers. As shown in this figure, with the smaller size of the inner-relation buffer, the execution time becomes longer, because the overhead has become heavier due to the increase of the number of times it takes to transfer packets of stream elements or demands. With the larger size of the inner-relation buffer, the execution time also becomes longer because the increase of the buffer size decreases the parallelism in stream-oriented processing. In particular, the setting of granularity is important in the small granularity range.

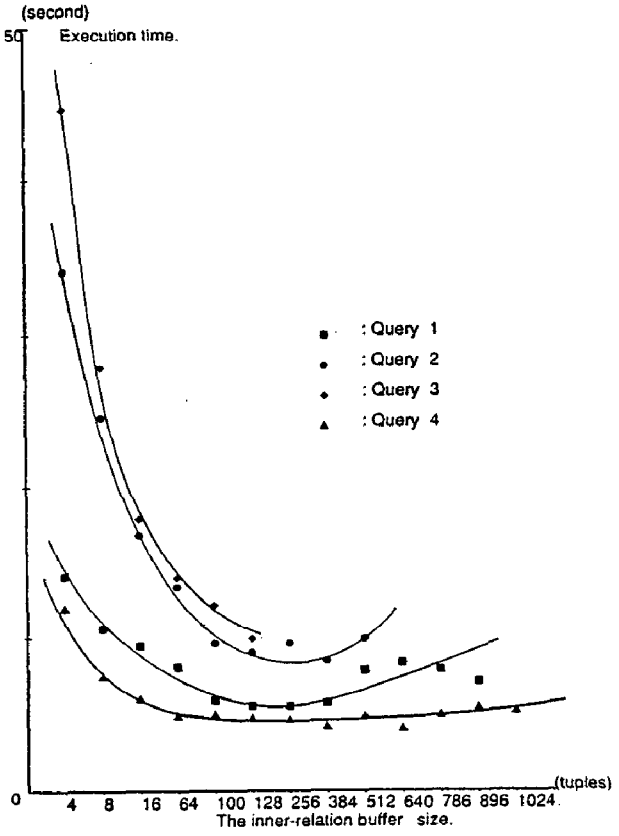


Figure 6: Execution time in various settings of communication granularity.

5 Conclusion

In this paper, we have presented an implementation method of an extensible parallel processing system SMASH for supporting a wide variety of basic operations. Our experimental results have shown that this implementation method is effective in performing parallel processing of actual queries. This implementation method can be applied to a wide variety of parallel processing environments in which the message passing mechanism is used for the interprocessor communication. Currently, we are designing another implementation method of the basic primitives for implementing the stream-oriented parallel processing scheme on tightly-coupled multiple processors with shared memory. In utilizing this method, the communication schemes between function instances are different from those presented in this paper. Furthermore, resource allocation strategies are also different.

We are extending our basic primitives so as to support a wide variety of advanced database applications. In particular, in extending our basic primitives, the flexibility and extensibility for descriptions and manipulations of complex objects[1] have been very important.

References

[1] M. P. Atkinson and O. P. Buneman, "Types and persistence in database programming languages," *ACM Comput. Surv.*, Vol. 19, No. 2, pp. 105-190, 1987.

[2] P. B. Hawthoron and D. J. DeWitt, "Performance analysis of alternative database machine architectures," *IEEE Trans. Softw. Eng.*, Vol. SE-8 No. 1, pp. 61-76, 1982.

[3] Y. Kiyoki, K. Tanaka, N. Kamibayashi and H. Aiso, "Design and evaluation of a relational database machine employing advanced data structures and algorithms," *in Proc. 8th Int. Symp. on Computer Architecture*, pp. 407-423, 1981.

[4] Y. Kiyoki, K. Kato and T. Masuda,"A relational database machine based on functional programming concepts," *in Proc. 1986 ACM-IEEE Computer Society Fall Joint Computer Conf.*, pp. 969-978, Nov. 1986.

[5] Y. Kiyoki, K. Kato, N. Yamaguchi and T. Masuda, "A stream-oriented approach to parallel processing for deductive databases," *in Proc. 5th Int. Workshop on Database Machines*, pp. 102-115, 1987.

[6] Y. Kiyoki, P. Liu and T. Masuda, "A resource allocation strategy in the stream-oriented parallel processing scheme for relational database operations," *Transactions of Information Processing Society of Japan*, Vol. 28, No.11, pp. 1177-1192, 1987.

[7] P. Liu, Y. Kiyoki and T. Masuda, "A computation method for buffer resource allocation in the stream-oriented processing scheme for relational database operations," *Transactions of Information Processing Society of Japan*, Vol. 29, No. 8, pp. 770-781, 1988.

[8] P. Liu, Y. Kiyoki and T. Masuda, "Efficient algorithms for resource allocation in parallel and distributed query processing environments," *Proceeding of the IEEE Distributed Computing Systems,1989* (to appear).

[9] P.G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access path selection in a relational database system," *Proc. of the ACM-SIGMOD Conf.*, pp.23-34, 1979 .

[10] "Programming reference manual for the Sun workstation," *Sun Micro Systems, Inc.*, 1986.

[11] P. C. Treleaven, D. R. Brownbridge and R. P. Hopkins, "Data-driven and demand-driven computer architecture, " *ACM Comput. Surv.*, Vol. 14, No. 1, Mar. 1982.

[12] S. R. Vegdahl, "A survey of proposed architectures for the execution of functional languages," *IEEE Trans. Comput.*, Vol. C-33, No. 12, pp. 1050-1071, Dec. 1984.

[13] C. T. Yu and C. C. Chang, "Distributed query processing," *ACM Computing Surveys*, Vol. 16, No.4, pp. 399-433, Dec. 1984.