Proceedings of the Fourth Symposium on Programming Languages and Software Tools
Visegrád, Hungary, June 9-10, 1995.

Parallel Execution of Object Functional Queries*

Lajos Schrettner[1], Tibor Gyimóthy[2], Zoltán Alexin[3], János Toczki[4]

[1] Department of Computer Science, JATE Szeged
e-mail: schrettner@inf.u-szeged.hu

[2] Research Group on the Theory of Automata, HAS
e-mail: gyimi@sol.cc.u-szeged.hu

[3] Department of Applied Informatics
e-mail: alexin@inf.u-szeged.hu

[4] Department of Computer Science
e-mail: toczki@inf.u-szeged.hu

Abstract:

In this paper a model is proposed for the parallel execution of OFL programs on MIMD architecture. OFL can be considered as a target language for object-oriented query compilers or a convenient source language for object-oriented query interpreters. The proposed OFL executor model uses the pipelined evaluation strategy to process queries.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and techniques; D1.1 [Programming Techniques]: Concurrent Programming; D1.3: Applicative Programming; I: Computing Methodologies.

Key Words and Phrases: Database Query Processing.

1. Introduction

1.1. Conventional Database Management

An important feature of databases that they ensure a persistent storage which means that the stored data can be permanently accessed by different applications and ad hoc queries. Usually, the business applications require only simply structured data records each stored just in a few bytes of memory. The transactions are often short and the number of different relationships between data items is not too big. Relational database systems are very suitable to handle these requirements. The table-oriented data model involved in relational systems provides good solution for the business applications. In Figure 1 the main elements of a conventional database management system (DBMS) is presented.

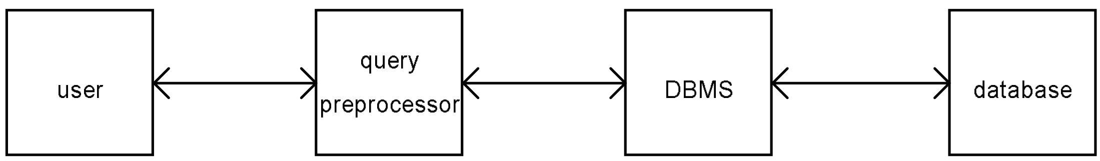

Figure 1: Database system

* Supported by the LPD COPERNICUS Project, CP 93:6638 and by MKM Grant No. 435/94

1.2. Object Oriented Model

There are some applications where the conventional DBMS concepts are not efficient. For example in the Geographic Information Systems (GIS) [1] the objects are often structured in a hierarchical manner. A conventional DBMS does not provide sufficient solution to handle these complex objects.

The object-oriented databases [2] which can be considered as DBMS with object-oriented data model play an important role in the efficient management of structured objects. The object-oriented data model supports the construction of complex objects, the definition of user defined types and methods. The concept of class is also involved in the object-oriented data model. Similar objects are grouped together in classes and methods and attributes are attached to these classes. In the object-oriented systems the objects are encapsulated which means that an object can be accessed through the methods defined on its class.

In this paper we present a method for the parallel execution of object oriented queries. The top-level schema of the object-oriented query process used in our approach is presented in Figure 2. We suppose that user's queries after a pre-processing phase (top-level optimisation) are given in OQL form [3].

From an OQL query an OFL (Object Functional Language) program is generated and the parallel executor operates on this program. OFL can be considered as a target language for object-oriented query compilers or a convenient source language for object-oriented query interpreters.

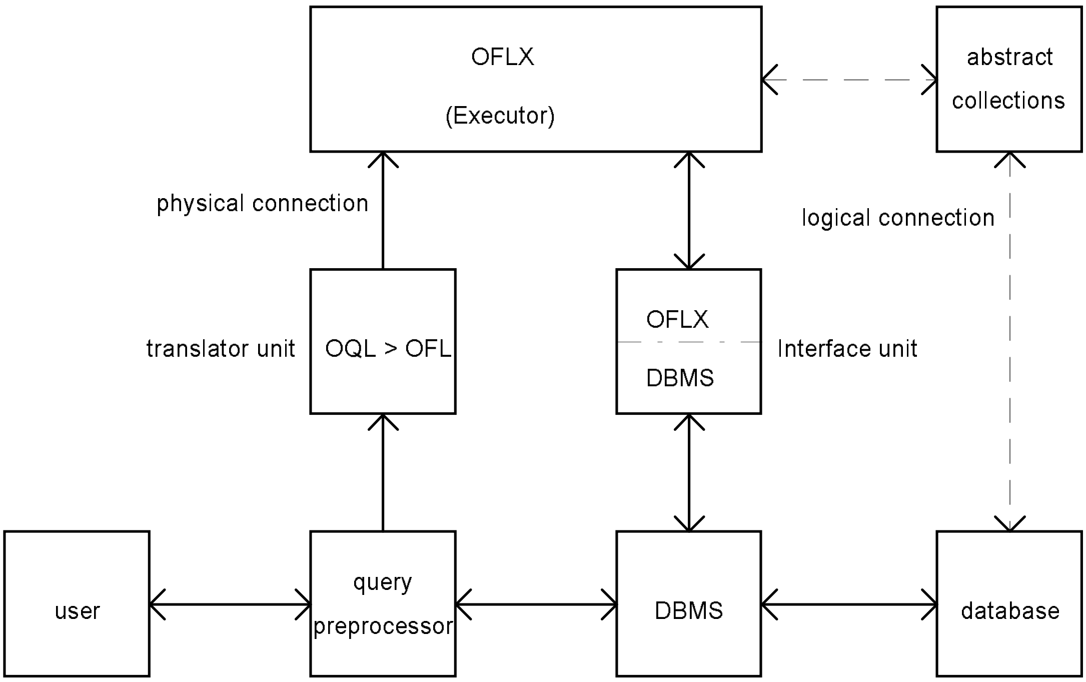

Figure 2: Database system with OFL

1.3. OFL - Object Functional Language

Based on first drafts [4, 5], the features of OFL can be summarised as follows: OFL programs are functional expressions and manipulate **objects** via **function** calls.

1.3.1. Objects

Typically some objects contain other objects as their members, these are called **collections**. Every collection must support three functions for the purpose of enumerating their members (see below).

Apart from three **special objects**, the OFL executor does not recognise the identity of the objects. These special objects are:

Error! Reference source not found. `false` boolean *false*

Error! Reference source not found. `true` boolean *true*

Error! Reference source not found. `nil` the object *nil*, used by the built-in functions

1.3.2. Object references

In OFL programs an **object reference** is used to access an object. There are two kinds of references:

Error! Reference source not found. indirect reference by **object variables**, these are
- either `false`, `true`, `nil` referencing the special objects mentioned above,
- or a **user variable**, defined by using the `assign` built-in function (see below).

Error! Reference source not found. direct reference to objects, these are string literals to the OFL executor. It is the responsibility of the function to which these direct references are passed to determine the identity of the object described by the string literal.

1.3.3. Functions

Every function accepts objects as its parameters and passes back a single object as its result, i.e. functions are single-valued and 'object' is the only data type in OFL programs. The same function name can be used to denote several functions with different arities. A parameter (argument) to a function can be an object reference or a function call.

There are two kinds of functions: **built-in** and **external**. External functions are not carried out by the OFL executor itself, their arguments are determined and passed along with the name of the function to the **environment** in which the executor is run. That environment is responsible for carrying out the function and returning the result to the executor.

There are **mandatory (traversal)** and **optional (behavioural)** external functions. Each collection maintains a pointer to its **current member** internally, this pointer can be set and moved by the traversal functions, this way the collection members can be enumerated.

Traversal functions

`Current( argument)`

Returns the current member of the collection

`First( argument)`

Sets the internal pointer to the first member and returns *true* if the collection is not empty. Returns *false* if the collection is empty.

`Next( argument)`

Returns *false* if all members have been enumerated. Sets the pointer to the next member of the collection and returns *true* otherwise.

Behavioural functions

A behavioural function is any external function that appears in an OFL program and is not a traversal function.

Built-in functions

A **predicate** is `false`, `true` or a function that returns *false* or *true*. All built-in functions return *nil*.

`assign( object_reference, argument)`

`object_reference` is treated as an indirect object reference throughout the program. The object it identifies is determined by the `argument`.

`sequence( argument, argument, ..., argument)`

Calls the arguments in the given order. The arguments are supposed to be functions. If an argument is an object reference, it has no effect. The return values of the functions are discarded.

`while( predicate, function)`

Calls the function until the predicate becomes *false*. Each iteration begins with the predicate evaluation.

`if( predicate, function, function)`

If the predicate returns *true*, the first function is called. If the predicate returns *false*, the second function is called. If the first parameter is not a predicate, the behaviour is undefined.

`forany( object_reference, predicate, function)`

Enumerates the members of the collection identified by `object_reference` until the call of `predicate` returns *true*. Then the `function` is called. If the collection is empty or `predicate` is never *true*, the function is not called.

`forall( object_reference, predicate, function)`

As `forany`, but enumerates all members of the collection and `function` is called each time `predicate` is *true*.

1.4. OQL - OFL Transformation

The OQL to OFL transformation is done by the OQL compiler. The compiler reads in a query and returns its OFL equivalent. The generated code contains a wide variety of library functions like the ones in the example below. The OQL compiler is available for UNIX and IBM PC machines as well.

An OQL query consists of several *defines* and a query. Each *define* corresponds to a view i.e. a collection expression that evaluates when used. A query can use the previously defined views. For the compilation, the schema of the data is necessary too. The OQL compiler can handle the schema definitions written in ODL language.

The query itself is an expression whose value (it may be either a large collection or a simpler object) is the result of the query. Therefore the compiler works as follows: it parses the input, replaces the *defines*, then builds up an expression-tree, finally from the expression-tree it generates the OFL code.

The following example shows how the OQL compilers transforms a SELECT query. Selecting from multiple collections means traversing all of the collections and picking up the suitable objects from them; so the compiler generates encapsulated

```
sequence(assign(x, ...), forall(x, ...))
```

statements for each item in the FROM part of the SELECT. For the SELECTed columns it generates **`Display`** library procedure calls in the body of the innermost **`forall`**. The **Display** simply prints the result to the terminal screen. The conditions in the WHERE part will be used in the headings of the **`forall`** statements for filtering the traversed collection elements.

```
SELECT p.LastName, v.Color, c.PartLabel
FROM p in Person, v in p.Owner, c in v.Composed
WHERE p.Age=16
```

```
sequence(
  assign( _person, 'People'),
  forall( _person, IntEq( Field( 'age', Current( _person)), '16'),
    sequence(
      assign( _vehicle, Field( 'owner', Current( _person))),
      forall( _vehicle, true,
        sequence(
          assign( _part, Field( 'composed', Current( _vehicle))),
          forall( _part, true,
            sequence(
              Display( Field( 'lastname', Current( _person))),
              Display( Field( 'color', Current( _vehicle))),
              Display( Field( 'label', Current( _part)))
            )
          )
        )
      )
    )
  )
)
```

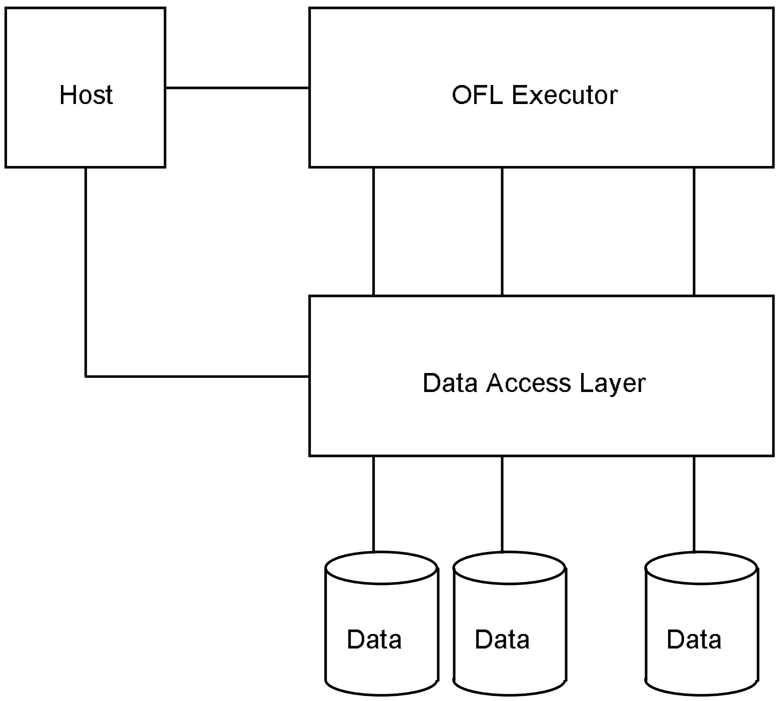

Figure 3 Parallel database machine

2. Parallel Executor

2.1. Parallel Environment

For the implementation of the parallel OFL executor we first have to decide what kind of environment we assume in which the executor runs. We propose a general parallel database machine architecture and show how the executor can be implemented on it. The architecture is based on the MIMD model of parallelism, i.e. processors with local memory communicate on message channels. The transputers [6] belong to this model and will be used to first simulate and eventually implement the system.

2.2. Parallel Database Machine

The database machine is separated into two functional units, the Executor and the Data Access Layer (Figure 3). Both of these are assumed to consist of several processors. The task of the Data Access Layer is to serve the Executor. It encompasses both the OFLX-DBMS interface and DBMS modules (see Figure 2). We do not deal with the internal structure of the access layer here, it is assumed that it carries out its assigned tasks. With this assumption we can concentrate on the execution of OFL programs, while realising that the internal organisation of the access layer is not trivial. The system is connected to a host machine where query input and query pre-processing takes place.

The OFL Executor uses the pipelined evaluation strategy to process queries. For this reason, the processors in the Executor unit are connected to each other in a pipeline (Figure 4). The roles of the functional units will be discussed later.

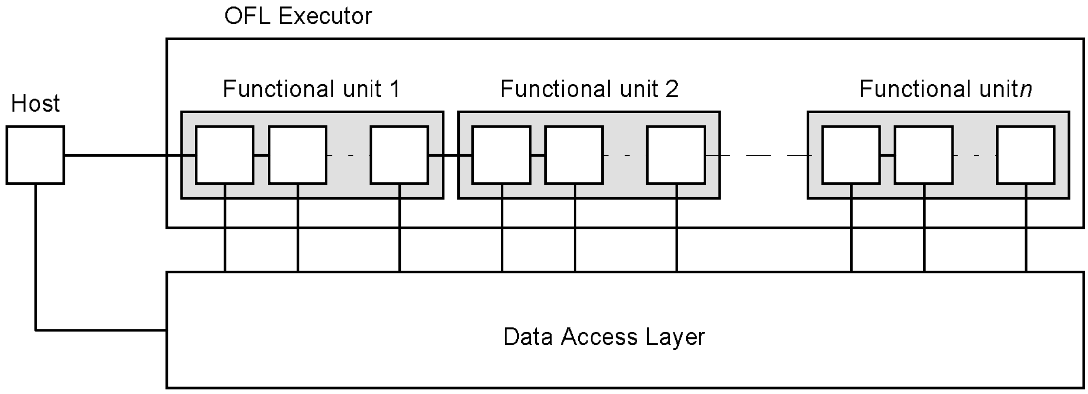

Figure 4: Executor internal structure

2.2.1. Query Pre-processing

Before the execution of an OFL program it has to be analysed and pre-processed so that the executor would be able to evaluate it efficiently. The papers which define and describe OFL suggest that OFL programs generated to be evaluated using the pipelined strategy have a common structure. This structure is best shown by the so-called abstract collection traversal graph, which is the last step in the generation sequence before an OFL program is produced. The `forall` function has the most important role in this respect, it determines the overall structure of a query.

The pre-processor has to look for the first `forall` function occurrence in the OFL program and split up the OFL tree (program) so that the subtree corresponding to the third argument of the `forall` is separated. This is performed iteratively until the resulting subtree contains no `forall`. The subtrees found this way will be evaluated by the functional units introduced in Figure 4. The `forall` functions selected for code separation purposes all lay on a path starting from the root of the parse tree. In principle any path of the parse tree could be selected for this purpose but not all of them can be expected to be equally preferable. More work is needed to find out what factors should be considered when the path is determined.

The example program given earlier is sliced as follows:

❶
```
sequence(
  assign( _person, 'People'),
  forall(
    _person,
    IntEq( Field( 'age', Current( _person)), '16'),
    ?
) )
```

❷
```
sequence(
  assign( _vehicle, Field( 'owner', Current( _person))),
  forall(
    _vehicle,
    true,
    ?
) )
```

❸
```
sequence(
  assign( _part, Field( 'composed', Current( _vehicle))),
  forall(
    _part,
    true,
    ?
) )
```

❹
```
sequence(
  Display( Field( 'lastname', Current( _person))),
  Display( Field( 'color', Current( _vehicle))),
  Display( Field( 'label', Current( _part)))
)
```

The program is separated into four functional units. Invocations of the `Current` function have been replaced with object variables local to the unit, because the objects of the corresponding collection are sent on the message channels between functional units and are processed in parallel. These transformations can easily be carried out by a pre-processor.

2.2.2. Query Evaluation

The evaluation proceeds in two steps. First the subtrees are distributed to the functional units. In the second step, evaluation of the subtrees begins in parallel. Functional units can proceed unless an object is available to work on. The finer structure of the executor is discussed after an introduction to the INMOS parallel C for transputers.

2.3. Software

2.3.1. Parallel C Features

The INMOS parallel C system consists of an ANSI compliant C compiler extended with features needed for process creation, scheduling and message passing [7]. These are implemented as C functions and are separated into libraries. We plan to base our implementation on this language.

The parallel computational model consists of variables, processes and channels. Processes are ordinary C functions and run independently of each other. Variables can be of any C type including user defined types. Channels are a new data type defined in the extension libraries. They serve as a communication medium between processes. If two processes reside on different processors, the only way for them to exchange data is to use channel communication. Every channel has at most two processes associated with it, one of them can send messages on the channel, the other can receive the messages. Communication takes place when both processes are ready for it, the first reaching the communication point must wait for the other to arrive too.

2.3.2. OFL Objects and Functions

An OFL program manipulates objects via functions. In order to be able to make an OFL implementation, we have to decide how to represent objects during execution.

Fortunately only a few objects are processed directly by the executor, most of them are results of external function calls and these resulting objects are stored and passed to other functions later. For these reasons the objects manipulated by executor are the boolean constants `true`, `false` and the special object `nil` that is the result of some second-order functions like `forany` and `sequence`. It is used when a function does not really have a meaningful result. It can only appear in contexts where the function result would not be used anyway. We assume also that there exist a standard collection

`Boolean` consisting of the two boolean constants and an other standard collection `Nil` which contains only the object `nil`.

The representation of objects is therefore quite simple, it is a C structure that contains the value of the object if they are one of the above three or contains an identifier otherwise. This identifier is used only outside the executor, its value is not inspected in any way. For efficiency reasons in a real implementation objects of small size (e.g. strings) can be stored directly in the object structure, but this does not affect our discussion here.

Only a small number of functions are built in the language. Other functions appearing in an OFL program are called external functions and indeed they are executed by the system to which the OFL executor is connected (in the interface unit in Figure 2). There are some external functions which are required to be present so that the executor can work. These are the following, the so-called traversal functions:

`Boolean First( Collection)`: This function selects an object from the collection as the first object. If successful, this object may be accessed later by the function `Current`.

`Boolean Next( Collection)`: This function selects the next object from the collection. Again the object may be accessed later by `Current`. If `Next` returns `false`, then all members of the collection have been enumerated.

`Object Current( Collection)`: Returns the current instance of the specified collection.

2.3.3. Processes and Configuration

A functional unit consists of the processes shown in Figure 5. The input and output processes are responsible for handling the communication with the neighbouring units. The buffer process stores objects which are results of the execution of the function assigned to this particular functional unit. If the neighbour on the right becomes idle, it sends a message to the output process, which retrieves an object from the buffer and forwards it to the idle neighbour for processing.

The most important part of this process structure is the evaluator which is an OFL interpreter and executes the same code repeatedly to every object received. Remember that the code assigned to the evaluator is determined in the pre-processing phase.

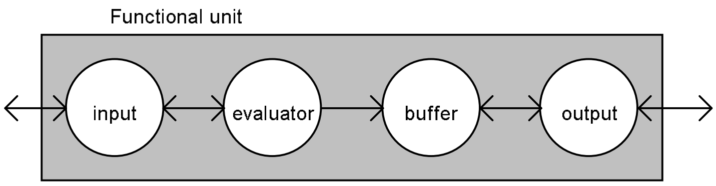

Figure 5: Processes in a functional unit

We now give the pseudocode of the evaluator based on a similar sequential code given in [4] and [5].

```
Object eval( Expression expr) {
  switch( expr_type( expr)) {
    case expr_Object:
      return( expr); break;
    case expr_Function:
      switch( func_type( expr)) {
        case func_forall:
          return eval_forall( func_params( expr)); break;
        case func_forany:
          return eval_forany( func_params( expr)); break;
        case func_while:
          return eval_while( func_params( expr)); break;
        case func_if:
          return eval_if( func_params( expr)); break;
        case func_seq:
          return eval_seq( func_params( expr)); break;
        otherwise:
          return eval_extern( expr); break;
      };
  };
};

Object eval_forall( Expression coll, Expression pred) {
  c = eval( coll);
  ok = eval_extern( First( c));
  while ( ok) {
    if ( eval( pred)) {
      obj = eval_extern( Current( c));
      store( obj);
    };
    ok = eval_extern( Next( c));
  };
  return nil;
};
```

One thing should be pointed out, the role of the function `store`. When it is invoked, the object parameter is stored in the buffer process together with a suitable environment information describing the state of the processing. This is required so that the next functional unit in the pipeline could continue the processing of the query.

The code of `eval_forany` is similar, clearly the `store` function is not called at all.

```
Object eval_while( Expression cond, Expression func) {
  while( eval( cond))
    eval( func);
  return nil;
};

Object eval_if( Expression cond, Expression t_f, Expression f_f) {
  if ( eval( cond))
    return eval( t_f);
  else
    return eval( f_f);
};

Object eval_seq( Expression f[], int fsize) {
  for ( i=1; i <= fsize; i++)
    eval( f[ i]);
  return nil;
};
```

As can be seen, all built-in functions except `if` return the object `nil` as a result. It is the responsibility of the compiler that produces the OFL program that this behaviour does not cause any problem during processing.

We have not taken into account so far how many processors we have for the executor. If we have at most as many processors as functional units, then the scheme above is appropriate. But as the number of functional units is determined by the number of `forall`s in a program and that number is limited, we should consider the case when there are more processors than functional units. In this case we assign several processors to a single functional unit, but this induces minor changes in the process structure as shown in Figure 6.

It can be seen that the input, output and evaluator processes are repeated as many times as necessary, but still there is only one buffer process. Further the input and output processes of a processor are connected to be able to communicate with each other directly. This is so because this way it is possible for all evaluators to access new objects via a chain of input and output processes.

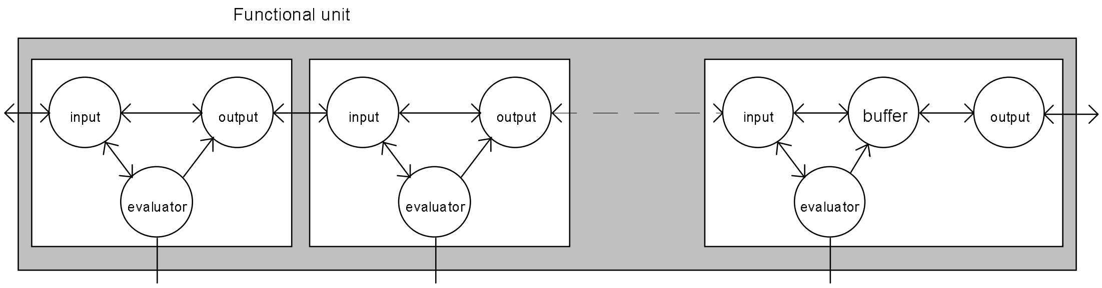

Figure 6: Multiprocessor functional unit

2.4. Performance Measurement - Simulation

We would like to measure the effectiveness of our system without having to build a parallel database machine and implement all the necessary software components. For this to be possible, we only implement the OQL-OFL translator and the OFL executor and substitute all the remaining parts with software modules that simulate the behaviour of the real ones. Some of our partners participating in the LPD project are developing cost models for databases with various physical organisation and logical structure. We hope to apply those results to test our system.

In order to collect the runtime performance data we include a profiling module into the executor. The task of the module is to accept calls from various parts of the program and store runtime data in the memory. When a query has been processed, the data collected are transmitted to the host machine and stored in files. These files are then subjected to data analysis, for which purpose a separate software tool is under development. One file is produced per processor, each file consisting of a series of records that contain a timestamp and the quantity of some agent at that instant. The data analyser tool produces data charts in an easily comprehensible graphical format. Runtime data can browsed, enhanced and compared on the display. With the assistance of this tool we hope to be able to pinpoint causes of possible inefficiencies and improve our design.

3. Discussion

We presented a possible solution for the parallel evaluation of database queries expressed in the language OFL. OFL was proposed as an intermediate low level language that is independent of data models and can efficiently be executed. Based on papers describing OFL and a sequential executor, we designed a parallel executor which tries to retain the semantics of the language as much as possible and at the same time tries to exploit the potential parallelism present in queries. We are planning to examine the possibility to embed the OFL executor in a more general parallel environment that has been developed for the solution of so-called decomposible problems [8]. Although this approach seems sensible, we will have to redefine parts of the semantics of the language with a parallel executor in mind.

4. References

[1] O. Gunther: Efficient Structures for Geometric Data Management. *LNCS 337*, Springer-Verlag, Berlin 1988.

[2] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, S. Zdonik: The object-oriented database manifesto, *In proc. of DOODS*, Kyoto, Japan 1989.

[3] R.G.G Cattell, ed.: *Object Databases: The ODMG-93 Standard*, Morgan&Kaufman, 1993

[4] G. Gardarin et al: OFL: An Object Functional Language to Map Extended SQL, *LPD Report LPD.PRiSM.003*

[5] G. Gardarin, F. Machuca, P. Pucheral: A Functional Execution Model to Evaluate Object-Oriented Queries, *PRiSM Technical Report*

[6] INMOS Ltd: *The Transputer Databook*, 1993

[7] INMOS Ltd: *IMS D7214 ANSI C Toolset*, 1993

[8] L. Schrettner, I. E. Jelly: A Test Environment for Investigation of Dynamic Load Balancing in Transputer Networks, *Proceedings of the World Transputer Congress*, Aachen, Germany, September 1993, IOS Press

5. Postal Addresses

Lajos Schrettner
Department of Computer Science
József Attila University
PO Box 652, 6701 Szeged
Hungary

Tibor Gyimóthy
Research Group on the Theory of Automata
Hungarian Academy of Sciences
1 Aradi vértanúk tere, 6720 Szeged
Hungary

Zoltán Alexin
Department of Applied Informatics
József Attila University
PO Box 652, 6701 Szeged
Hungary

János Toczki
Department of Computer Science
József Attila University
PO Box 652, 6701 Szeged
Hungary