

The European Declarative System, Database, and Languages

Article

Published Version

Haworth, G. M. ORCID: <https://orcid.org/0000-0001-9896-1448>, Leunig, S., Hammer, C. and Reeve, M. (1990) The European Declarative System, Database, and Languages. IEEE Micro, 10 (6). 20-23, 83. ISSN 0272-1732 doi: 10.1109/40.62726 Available at <https://centaur.reading.ac.uk/4554/>

It is advisable to refer to the publisher's version if you intend to cite from the work. See [Guidance on citing](#).

Published version at: <http://www.computer.org/portal/web/csdl/doi/10.1109/40.62726>

To link to this article DOI: <http://dx.doi.org/10.1109/40.62726>

Publisher: IEEE

Publisher statement: ©1990 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE

All outputs in CentAUR are protected by Intellectual Property Rights law, including copyright law. Copyright and IPR is retained by the creators or other copyright holders. Terms and conditions for use of this material are defined in the [End User Agreement](#).

www.reading.ac.uk/centaur

CentAUR

Central Archive at the University of Reading

Reading's research outputs online

F E A T U R E S

8 Guest Editors' Introduction: Parallel Computing in Europe
Jean-Francois Omnes, Thierry Van der Pyl, and Philip Treleaven

12 Parallel Computers for Advanced Information Processing
Pierre H.M. America, Ben J.A. Hulshof, Eddy A.M. Odijk, Frans Sijstermans, Rob A.H. van Twist, and Rogier H.H. Wester
 Six major companies combined talents with several research organizations to investigate and compare design approaches for parallel computer systems.

16 Transputers—Past, Present, and Future
Colin Whitby-Strevens
 Enhancing performance of existing transputers, the ESPRIT projects continue to develop the all-around capabilities of this chip—with a vision for its future in general-purpose computing.

20 The European Declarative System, Database, and Languages
Guy Haworth, Steve Leunig, Carsten Hammer, and Mike Reeve
 To address future demands of immense, complex databases, this intelligent information server exploits large-scale parallelism and supports current interfaces such as Unix and SQL.

24 Architectures Within the ESPRIT SPAN Project
Peter Rounce and Jose Delgado
 To integrate symbolic and numeric computing on parallel systems, project participants developed a target architecture that resulted in a number of significant advancements in programming languages and architecture.

28 Pygmalion: ESPRIT II Project 2059, Neurocomputing
Bernard Angeniol
 Neuron processors, dedicated ASICs, and standard computational tools for programming and simulation could lead to a general-purpose parallel neurocomputer.

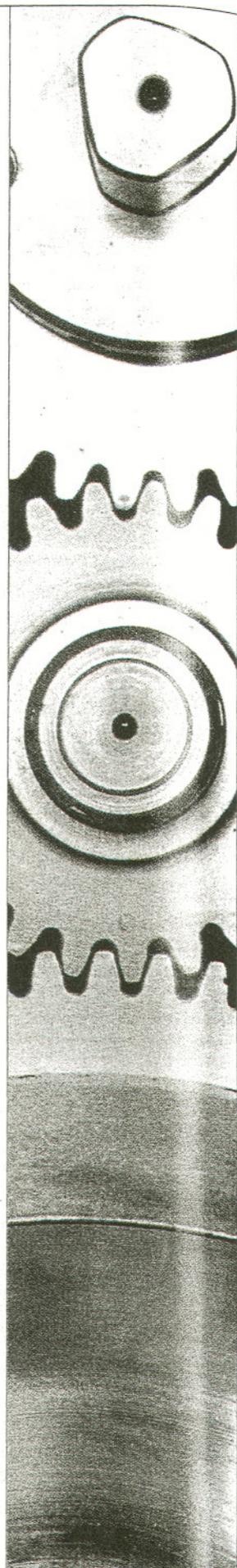
32 Annual Index, Volume 10

Cover by TIB/West © Michael Melford, Design & Direction

D E P A R T M E N T S

3 From the Editor-in-Chief
5 Micro World
 Reunifying East and West
11 1991 Editorial Calendar
39 Micro Law
 The *Paperback* case, Part 2
42 Micro Standards
 A standard to consider
46 Micro Review
 Desk planners plus

48 Software Report
 Quality improvement
51 Micro News
 Analyzing 32-bit computers; MCMs
53 New Products
 DSP, design tools, hand-held computers
59 Product Summary
104 Micro View
AMD v. Intel



On
de
en
ar
sci
Er
N
Te
A
cr
in
m
fe
in
a
C
L
b



The European Declarative System, Database, and Languages

The EP2025 EDS project develops a highly parallel information server that supports established high-value interfaces. We describe the motivation for the project, the architecture of the system, and the design and application of its database and language subsystems.

Guy Haworth

Steve Leunig

ICL

Carsten Hammer

Siemens

Mike Reeve

ECRC

In 1988 Bull, ICL, Siemens, and their jointly owned European Computer Research Centre (ECRC) identified a common interest in supporting the processing of future intensive applications. The four partners defined a European Declarative System proposal, which the European Commission supported as project EP2025, a part of the European Economic Community's ESPRIT program. The EDS project began in 1989 and extends until 1992 with phases of definition, component development, and system integration.

EDS machines primarily function as information servers to manage all varieties of information intelligently. They will support languages and interfaces of value that are already established and in use: Unix, extended SQL, Lisp, C++, and the ECRC Elipsys parallel logic programming language.

The following analysis of the requirement for information servers motivated the EDS project and justifies this role for EDS machines.

Information server requirement

Enterprises in the industrial, service, government, administrative, and defense sectors use information technology today. They depend increasingly on their information resource to:

- reduce operational costs,
- improve effectiveness from stock holding to customer service,

- support business development in new markets, and
- create a lasting competitive advantage in a rapidly changing world.

These enterprises often regard their information as more important than their next product or service. As a result, they pursue a systems architecture that delivers highly reliable information technology support and comprises a:

- complete, coherent, and robust information base;
- portfolio of applications interworking through data; and
- framework of long-life interfaces protecting their investments.

Large corporate systems increasingly play the role of database or information servers; servicing the SQL interface today requires some 75-80 percent of the processor cycles. Many factors increase the load on information servers, a fact likely to require the development of systems with highly parallel architecture to meet the future demand.¹

Information resource. We've deliberately chosen the word *information* to be an umbrella term for the complete knowledge spectrum. We see this spectrum ranging from conventional formatted data to less structured text, representations of sound and image, and higher order knowledge in the forms, for example, of constraints, integrity

rules, business rules, and processes. Knowledge in its broadest sense, of course, includes facts and analysis, certainty, rumor, and speculation.

The volume of such information reportedly grows at some 25-30 percent yearly, a rate that we expect to be sustained by increased interest in text and image. Since this information is so valuable, we hope to store it with security levels that would be the envy of any bank. These levels suggest a large, central facility rather than a set of all too portable personal computer disks.

Information access. Only on-line systems support the effectiveness needed in our budget-conscious, competitive world. Literate information workers, desktop technology, CASE (computer-aided software engineering) tools, and business-to-business systems increase the volume of on-line transactions at some 20-30 percent a year. In addition, responses must come in a suitably short time, regardless of information volumes, transaction rates, or the incompleteness of the data input to specify the query.

We can characterize transactions in terms of frequency and complexity, as judged by the load they place on the computer system. Classic transaction processing occurs at a high rate with low complexity, while knowledge-based systems are highly complex and process at a low rate. Any computer system has a finite throughput capacity, shown by the performance frontier (see Figure 1).

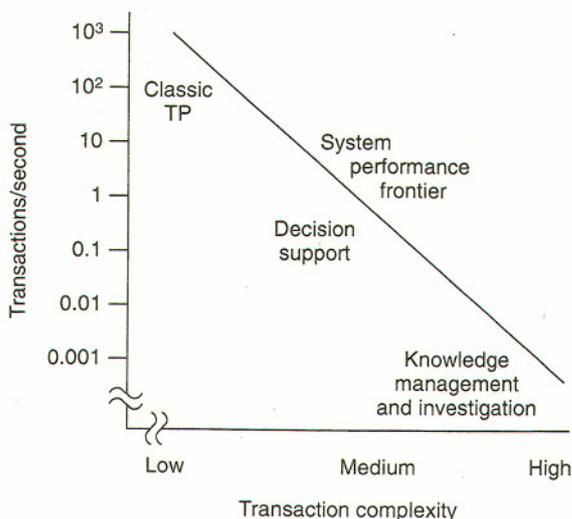


Figure 1. The range of transaction types.

Some evidence shows that performance problems impede the exploitation of "fifth-generation" knowledge-manipulation techniques. However, complex queries over knowledge bases do exist in several areas. These areas include govern-

ment administration, CAD (computer-aided design) systems including software engineering, the storage and scheduling system of distribution industries, and the remote maintenance systems of large utilities.

EDS technology intercept

The EDS project aims to advance the information server performance frontier in the fastest way. We plan to do so by intercepting key hardware and software technologies and integrating them behind established interfaces of high value to prospective customers.

The ANSI/ISO SQL standard² is the key interface today between the application and the database manager. This query language always allows the user to ask for a set of records, most likely chosen from a large database. In principle, a million processors could simultaneously assess one each of a million records to service an SQL query with obvious benefits to the response time of the query.

Opinions differ as to whether SQL will evolve sufficiently to meet the new requirements for managing more complex data types and manipulating knowledge. We believe the current investment in SQL will guarantee SQL a long life. We therefore proposed an extension of SQL, ESQL, to meet future requirements for more comprehensive databases.³

On the hardware side, the most rapid change is occurring in microprocessors, which continue to increase in raw power at some 50 percent each year. Today, microprocessors promise 25 MIPS (million instructions per second); tomorrow, 40, 60, and 100 MIPS. The challenge for the computer system architect is to achieve a similar increase in total systems throughput.

In addition, storage technologies are diversifying; large-scale RAM storage is often the most cost-effective way to improve total systems performance and the price/performance ratio. We expect to see today's commodity, 4-Mbit dynamic RAM chip succeeded by the 16-Mbit chip in 1994 and the 64-Mbit chip in 1998. DRAM cost per byte now drops at 60 percent a year, and in 1995 we expect DRAM storage to be only 20 times the cost of magnetic-disk storage.

The EDS machine therefore exploits microprocessors and large DRAM storage, supported by a communications infrastructure of suitable responsiveness and bandwidth. It avoids the bottleneck of a single path from processing power to storage by adopting a distributed "share-nothing" architecture. This architecture offers linear performance returns when the number of processors increases into the hundreds.

The EDS system

The static and simplified view of the EDS system seen in Figure 2 on the next page identifies the main interfaces and components. We designed the system to comprise a parallel

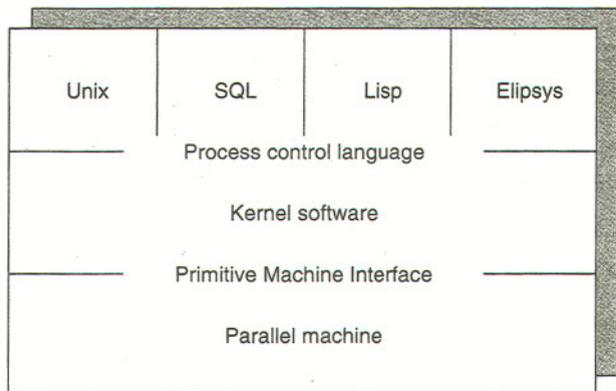


Figure 2. EDS system architecture.

processing machine and Emex kernel supporting Unix, extended SQL, Lisp, and Elipsys subsystems. The system will attach as an accelerator to a variety of Unix and proprietary hosts and be configurable up to 256 processors, each with up to 64 Mbytes of storage. We predict the following performance figures:

- *Database processing.* Meets the simple line-of-business Transaction Processing Council A benchmark performing 12,000 transactions per second at 30 percent utilization.
- *Lisp.* Meets the Boyer benchmark performing 140 Boyer runs per second.
- *Elipsys.* 32 MLIPS (million logical inferences per second) on average.

The EDS hardware. The EDS parallel machine⁴ consists of a message-passing network, which provides a number of identical connection ports for attaching various functional elements. We envisage four types of elements: processing, diagnostic, input and output, and host connection (Figure 3).

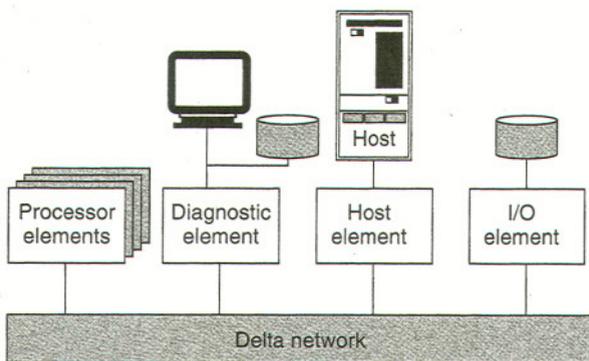


Figure 3. EDS hardware.

The processing element to be implemented for the prototypes consists of (see Figure 4):

- a main processing unit, a high-performance Sparc RISC (reduced instruction-set computer) with matching cache and memory management unit;
- a system support unit to offload the most critical parallelism primitives from the main processor;
- a network interface unit providing buffering and data transfer; and
- a local storage unit holding a maximum of 64 Mbytes of data.

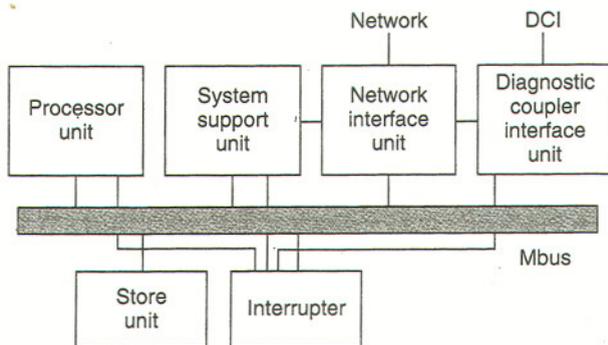


Figure 4. EDS processing element.

We expect later production versions of EDS to exploit the 16-Mbit chip and support 4 Gbytes of nonvolatile memory per processing element. We plan to simulate the effect of the well-understood nonvolatile storage during the project.

We designed the processing element to support efficiently the parallel operations of the execution models of the kernel, the database system, and the language systems. In addition to executing instructions within a normal sequential thread of computation, the processing element must support basic kernel operations such as passing a message. The primitive machine interface, or PMI, shown earlier in Figure 2, provides specific operations to support kernel functionality. PMI also introduces the required independence between the kernel software and the parallel machine hardware.

The EDS kernel and PCL. The EDS Process Control Language is the common interface through which all subsystems exploit and provide guidance to the parallelism features of the machine.

The concepts upon which PCL is based closely relate to those in Unix and in existing kernel interfaces such as Chorus Systeme's Chorus and Carnegie Mellon's Mach, both of which are designed to manage distributed systems. These concepts include virtual memory, processes, and interprocess communication. PCL develops these concepts to provide the

functionality and performance levels that a large-scale parallel system like EDS offers.

We particularly developed certain features of PCL in EDS:

- a multilevel process-context model with very lightweight threads,
- a storage model providing considerable flexibility in the sharing and management of virtual memory in a distributed system,
- efficient and reliable message passing,
- an exception-handling mechanism based on the message-passing scheme, and
- flexible scheduling and load balancing for a highly parallel system.

The inclusion within the EDS architecture of a common kernel and PCL interface brings a number of benefits: standard control of the machine, the exploitation of parallelism, and the use of system resources.

The EDS database system

The main exploitation focus of the EDS project is the development of an advanced database server. The server provide an order-of-magnitude performance improvement over mainframes and advanced functionality to extend the range of applications it supports.

The improved functionality will include facilities for:

- support of user-defined data types and methods,
- support of complex objects and large objects,
- deductive database capabilities,
- general integrity constraints, and
- triggers (actions to be carried out when a given event occurs).

These features will not only extend the range of applications that can be supported efficiently and naturally but will also increase programmer productivity currently supported by standard relational database systems.

To achieve these objectives, we use a number of design strategies:

- exploitation of the parallelism available in the base EDS system;
- exploitation of large, stable RAMs to hold the persistent data over time and across system breaks;
- a database system based on standard relational database technology that is extended to provide object-oriented database and deductive database facilities;
- an interface, ESQL,³ which is an extension of SQL. (The language provides a rich and extensible type system based on ADTs, or abstract data types, in which the

methods can be defined in various programming languages. It also provides complex objects with object sharing by combining the ADTs with object identity, and a Datalog-like deductive capability);

- database queries compiled into native machine instructions wherever possible; and
- an optimizer designed to be extensible to allow the system to evolve.

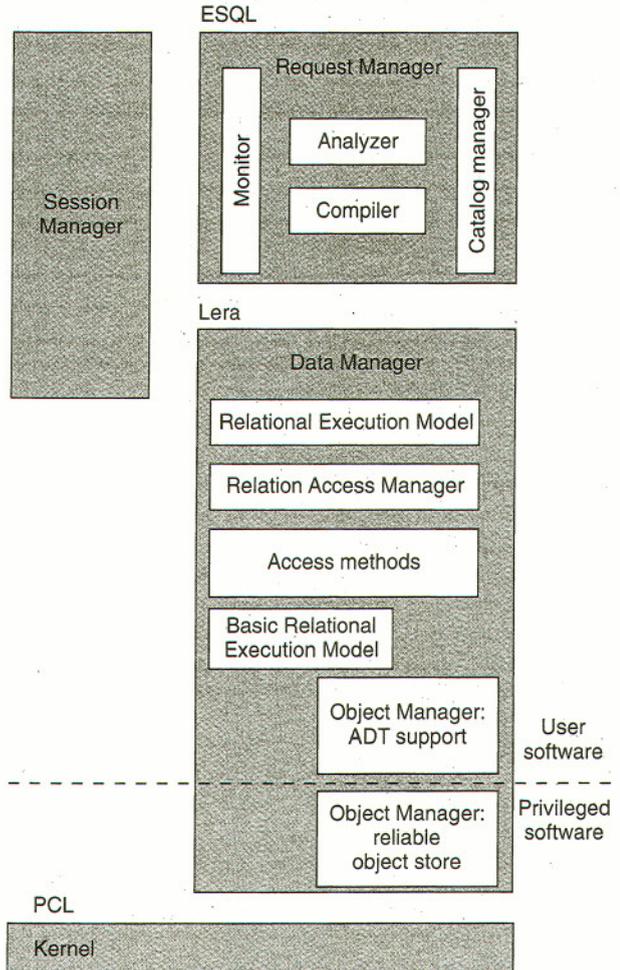


Figure 5. The logical structure of the database system.

Database system architecture. The database system splits into three main components, as shown in Figure 5. The Request Manager compiles database commands into a native machine code, the Data Manager provides the runtime facili-

continued on p. 83

EDS

continued from p. 23

ties required to execute those commands, and the Object Manager provides the shared object storage.

The Session Manager component provides the mechanism by which an application starts a database session. It creates an instance of each of the Request Manager and the Data Manager for each database session.

Figure 5 also shows the main interfaces between the components of the system. The first is ESQL, which is used by an application to access the database system. Lera is an extended relational algebra that is used between the Request Manager and the Data Manager. Last is the Process Control Language interface provided by the kernel.

A set of ESQL commands forms the input to the Request Manager, which compiles these commands in five stages:

- *Syntax analysis.* This stage parses the input and converts it to an internal structure. It also performs type checks.
- *Logical optimization.* The logical optimizer reorganizes the query by applying transformation rules to the query. These transformations perform functions such as predicate migration to minimize the size of intermediate results, the elimination of common subexpressions to remove redundant work, operator transformation to combine operators to simplify the task of the physical optimizer, application of constraints, and optimization of recursive queries.
- *Physical optimization.* The physical optimizer determines the order of the basic operations to minimize intermediate results, selects the best access path, chooses the algorithms, and determines the optimal degree of parallelism in the query. The choice of these options is based on the minimization of a cost function.
- *Parallelization.* The parallelizer translates the intermediate form generated by the physical optimizer into the parallel program representing the query.
- *Code generation.* This stage performs the final generation of the object module containing machine code and calls to the runtime facilities of the Data Manager.

As Figure 5 shows, the Request Manager consists of four main components: the monitor, analyzer, compiler, and catalog manager. The monitor provides the operational interface between the application and its instance of the Request Manager. The analyzer performs the first stage of the compilation of a query, and the compiler performs stages 2 to 5.

To support the management of the relations in a database and the compilation and optimization of queries, the Request Manager maintains a catalog, sometimes called a metabase, of information about the relations and schema in the data-

base. The catalog manager provides the Request Manager with a simple interface for accessing this data.

The parallel programs generated by the Request Manager execute in the runtime environment provided by the Data Manager, which consists of four main components:

- *Relational Execution Model.* This runtime library includes relational operations; operations supporting the ADT, objects, and rules; and controls operators.
- *Relation Access Manager.* This manager provides a global abstraction of the relations in the database. That is, it hides the distributed nature of the relations from the operations in the Relational Execution Model. The manager also provides the mechanism for calling the appropriate access methods for the indexes associated with a relation.
- *A set of access methods.* These methods provide the mechanisms for accessing the tuples of a relation. One access method will implement each index associated with a relation. The indexes offer fast methods for accessing the tuples of a relation.
- *Basic Relational Execution Model.* This parallel program environment provides abstractions tailored for the efficient execution of Request Manager programs.

The Object Manager provides basic object storage and manipulation facilities required to support the database system. This manager stores persistent objects, controls concurrent use of shared objects, and provides logging and recovery facilities for transaction support. We based the Object Manager on the Arjuna system⁵ with ideas incorporated from the CHOICES⁶ and Camelot⁷ projects.

Parallel execution of queries. The major influences on the Relational Execution Model design were the DDC project⁸ and the Bubba project.⁹ DDC was a project in the ESPRIT I program that had the objective of building a multiprocessor database machine. The Bubba project was a multiprocessor database system. We based the parallel execution of the queries on the following principles:

- 1) The relations are horizontally partitioned into fragments that are distributed across the set of available processing elements of the machine. One of the advances in the physical optimizer is the development of methods for determining the degree of parallelism in an operation. These methods allow the system to determine the optimal number of processing elements to be used during evaluation. For base relations the assignment of fragments to physical processing elements is relatively static. For the intermediate relations however, the assignment occurs at execution time. We refer to the set of processing elements across which a relation is partitioned as the *home* of the relation.

- 2) Where possible, processing takes place at the location of the data so the data is not moved. Naturally, this is not possible when an operation involves more than one relation. In this case the optimizer must choose the local operations in such a way as to minimize the movement of data.
- 3) The separation of data flow and control flow allows optimizations that significantly reduce the number of control messages.

A standard exemplar that is being used within the project forms the basis of the description of the parallel evaluation of queries. This exemplar is a share management system. The schema in Figure 6 defines two relations from the exemplar.

```
CREATE TABLE scost (
  share-id c4,
  cost integer2;
  currency c2);
CREATE TABLE exchange (
  currency c2,
  rate integer2);
```

and the query is:

```
SELECT share-id, cost, rate FROM scost, exchange
WHERE cost < 100 AND scost.currency = exchange.currency
```

Figure 6. Share management system exemplar.

We plan to extend the Create table command to allow users to specify the distribution algorithm, the attribute to be used, and the size of the home. In the absence of user-supplied information the physical optimizer chooses these parameters for the relations. The Data Manager determines the mapping of the relations based on the sizes of the relations' homes and the loading of the processing elements.

In this example we assume that the Data Manager chooses processing elements 1, 3, 8, and 9 for Scost and 4 and 5 for Exchange. We also assume that a hash function on Share-id distributes Scost, and a hash function on exchange distributes Currency.

When the Request Manager compiles the query, the physical optimizer decomposes the Join operation implied by the query into two suboperations Sel and Join. Sel prefilters the local fragment of Scost for those tuples in which cost is less than 100. Sel then distributes the tuples using the hash function for Exchange. The Join suboperation joins a tuple from Scost with the local fragment of Exchange. A trigger message sent to the Sel operations starts the processing of the query. Figure 7 illustrates the execution of this query.

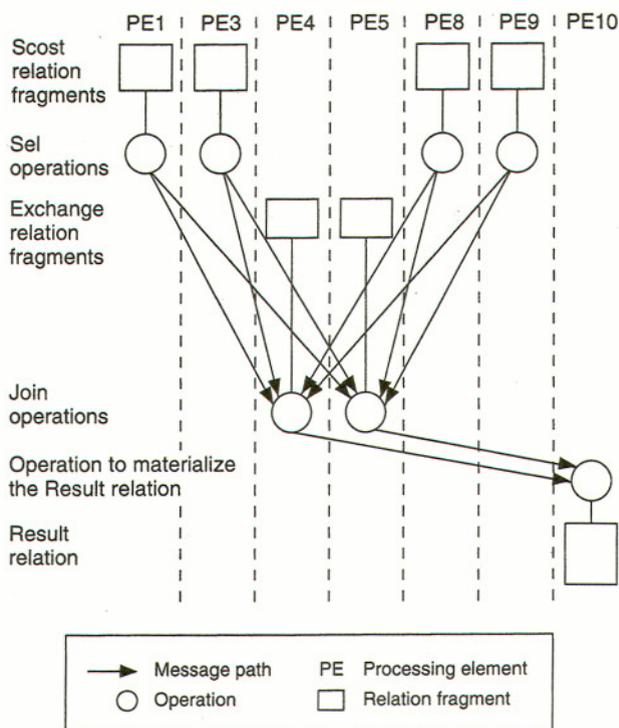


Figure 7. An example of a query execution.

This simple example illustrates the main principles of the computational model:

- relations are partitioned into fragments, which are distributed across their homes;
- relational operations decompose into operations that execute at the home of the relations on which they operate and so use purely local data; and
- their inputs are either a stream of messages or a fragment of a stored relation.

However, this very simplified account of the execution of the query does not discuss many important issues. One important benefit of processing the relations locally is that it allows the lock management to also be processed locally, thus providing an important performance improvement.

The Elipsys language

Elipsys¹⁰ is a parallel logic programming system for complex applications. The system integrates Or-parallelism, constraint satisfaction through finite domains, and an interface to the EDS database server.

The particular combination of Or-parallelism and constraint-satisfaction problem-solving techniques, which prune the

search space in an a-priori manner, provides an efficient platform for executing search-intensive programs. Elipsys solves a typical combinatorial search problem—for example, graph coloring, scheduling, and some other related operations research problems—in polynomial time.

The syntax of the Elipsys programming language is derived from DEC-10 Prolog. Elipsys makes available to the programmer the following features:

- *Data-driven computation.* This feature gives the programmer a flexible way of instructing the logic programming system in the way the paths of the search space can be computed.
- *Built-in constraints.* We build in simple equalities and inequalities, linear equations, and optimized branch-and-bound techniques, which range over the domain of finite discrete sets.
- *User-definable parallel constructs.* Predicates can be annotated as candidates for parallel evaluation and interface to the EDS database server through ESQl.

The Elipsys execution model in Figure 7 combines a message-passing architecture for the control and scheduling of parallel work and a distributed, shared virtual address space for the implementation of the binding environment. The above combination permits Elipsys to execute efficiently under the Emex kernel by taking advantage of the facilities provided for task and thread management. It also uses the Emex-provided, distributed, and shared virtual memory scheme, which is kept coherent by a "lazy, strong" method. This coherence scheme does not perform any coherence maintenance operations by default; explicit synchronization points in the application code trigger the operations.

The Elipsys binding environment is both read-only and shared. A control Or-tree and a shared environment represent the search space. A descendent Or-node inherits the binding environment of its ancestor Or-nodes. This inherited environment is read-only. Thus all the descendent Or-nodes hold the same view of the inherited environment. Modifications to the shared environment occur through auxiliary structures, which are local descendent Or-nodes. These structures in turn become shared whenever a control Or-node gives rise to any descendent Or-nodes.

The message-based Elipsys control and scheduling mechanisms make use of the control Or-tree data structure, which is distributed over a set of workers. Each worker is allocated to one EDS processing element; a worker consists of a distributed scheduler, performing scheduling and control functions, and a set of engines. An engine performs sequential resolution steps, extended linear resolution with a selection function applied to definite clauses over finite domains. It also manages the interface to the EDS database server. A scheduler-engine interface describes the schedul-

ing policy, pruning, and input/output interactions between the scheduler and the engine.

Advanced applications using Elipsys

Elipsys is oriented toward complex applications. We are developing a suite of programs to demonstrate the practicality of Elipsys for a wide range of applications domains. These programs will highlight different design features of Elipsys:

- *Compatibility with existing applications.* A civil engineering program analyzes possible faults in concrete piles from acoustic data. The UK University of Bristol is parallelizing and porting this sequential Prolog program to the Elipsys subsystem. This activity will identify the potential problems that may be encountered when converting existing Prolog applications to Elipsys.
- *Capability for handling large data sets.* The University of Athens is developing a tourist advisory system for Greece. The system provides customized holiday packages for individual tourists as well as general information for potential visitors to Greece. This application makes extensive use of the Elipsys connection to the EDS database subsystem. The raw tourist data can be stored in the EDS database rather than within Elipsys itself.
- *Deductive capability.* System and Management S.p.A. in Italy currently implements a Treasury Management System, an expert system for banking. Its role is to suggest profitable investment plans to bankers. Such an application is well suited for Elipsys and fully uses Elipsys' inherent deductive power and capability for interworking with the EDS database server.
- *Capability for managing complex data structures.* ECRC in Munich is developing several applications in the domain of molecular biology. This domain requires the management of vast amounts of complex data, again using the Elipsys/database connection. Moreover, the system requires the complex symbolic processing, for example, in structures matching different DNA molecules, and the built-in constraint facility of Elipsys.

The application language

Lisp is a powerful general-purpose programming language whose programs are written on a much higher level of abstraction than the Pascal, Ada, and C procedural languages. Being so powerful, Lisp has been widely used for complex applications such as artificial intelligence. To this expressive power, we added the power of parallel processing in EDS Lisp.¹¹

EDS Lisp extends the Common Lisp language. Because Common Lisp constitutes a de facto standard, users can easily port most existing Lisp applications to the EDS machine.

We selected the extensions of EDS Lisp after an intensive

study of other parallel Lisp systems. The extensions allow access to the EDS database system, and they provide language constructs for explicit parallelism. Explicit parallelism enables the programmer to specify large-grain parallelism that fits well to distributed-memory machines like the EDS.

An EDS Lisp program can have an indefinite number of parallel processes. The programmer creates processes to perform some action in parallel and to return a value. The EDS system schedules these processes. EDS Lisp contains a single construct to spawn parallel processes, the Future construct known from other parallel Lisp dialects.¹² Future constructs support transparent use of results of parallel processes. The main idea is that a Future immediately returns an (initially empty) placeholder for the result of the spawned process. The spawning process can then continue operation. When some process accesses this result, it waits until the result is available and then continues operation. Both the placeholder mechanism and the implicit waiting are invisible to the programmer. Consider, for example, the following piece of EDS Lisp code:

```
(setq x (future f p1 ... pn))
```

which corresponds to

```
x := future (f (p1, ..., pn));
```

in a procedural programming style. A parallel process is spawned using the Future construct to compute the function f with parameters p_1, \dots, p_n . The Future call immediately returns a placeholder for the result of f and assigns it to the variable x . The spawning process then continues in parallel to the process computing f . If a process reads the variable x , it tests implicitly whether the result is available and waits if necessary.

EDS Lisp also provides a Mailbox concept for communication between processes and a Critical Section mechanism, among other things, for synchronized access to shared variables.

Metal

The Metal machine translation system translates natural-language documents¹³ and currently requires a special-purpose Lisp machine for production use. The EDS Lisp application is complex enough to conquer both the CPU-power and storage limitations of today's workstations.

We expect a speedup of more than a factor of 300 for running Metal on the EDS machine; the translation of 250 pages that needs 10 hours today should be accomplished in two minutes on the EDS machine, as shown in Figure 8. This performance increase is highly relevant for the application, because the translation volume for technical documentation

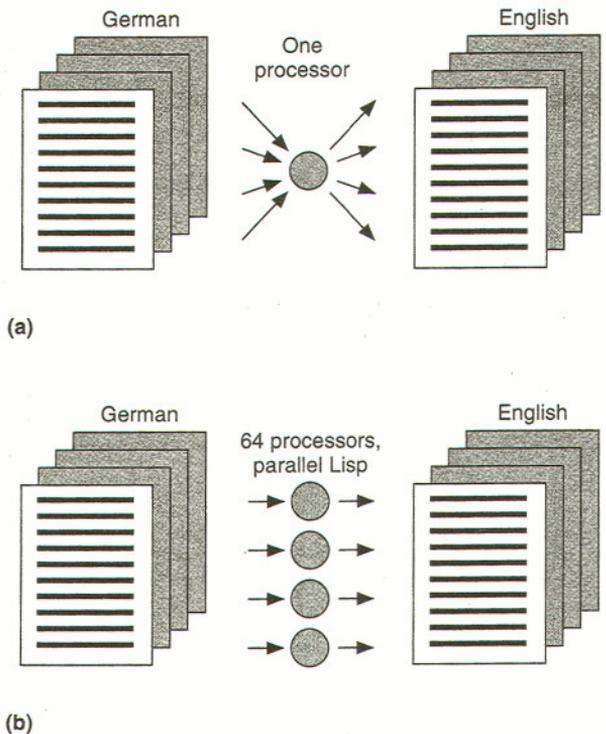


Figure 8. Sequential (a) and parallel (b) translation on Lisp systems.

is immense. The documentation for a complete technical product line often amounts to several hundred thousand pages that have to be made available in a multitude of languages.

Users can access the parallel processing power of EDS Lisp not only to translate such a huge mass of text but also to improve the quality of the translation by using more advanced and therefore more resource-intensive algorithms.

THE EDS PROJECT IS A MAJOR, PROMISING Commission of the European Community-sponsored ESPRIT II collaboration between Bull, ICL, Siemens, and their jointly owned ECRC research center. The EDS system primarily focuses on the large-scale information server, which must manage information efficiently and effectively across the spectrum from data to knowledge.

The EDS system enables programs calling the SQL, Lisp, and Elipsys interfaces to exploit large-scale parallelism essentially transparently. We've described the database and language aspects of the EDS system, looking at the SQL, Lisp, and Elipsys subsystems.

The EDS project combines the complementary skills of its partners and associate partners to achieve a clear and common goal. At the end of the second of four years, the project is on schedule to switch on an EDS machine in 1991 and demonstrate applications in 1992. [1]

Acknowledgments

We thank both the European Commission, DG XIII, and the senior management of the EDS partners and associate partners for their continuing support for this strategically important European initiative. We also thank all the members of the EDS team for their dedicated work on the project.

References

1. G. Haworth, "Information Servers, Present and Future," *Proc. Unicom Commercial Parallel Processing Seminar*, Unicom, Uxbridge, Middlesex, UK, to be published in 1991.
2. *ISO 9075, International Standard, Information Processing Systems, Database Language SQL*, 1987-06-15, International Standards Organization.
3. G. Gardarin and P. Valduriez, "ESQL: An Extended SQL with Object and Deductive Capabilities," *Proc. Int'l. Conf. Database and Expert System Applications*, A.M. Tjoa and R. Wagner, eds., Springer-Verlag, Berlin, Aug. 1990, pp. 299-307.
4. M. Ward, P. Townsend, and G. Watzlawik, "EDS Hardware Architecture," *Proc. Parle 90 Conf., Lecture Notes in Computer Science*, H. Burkhard, ed., Vol. 457, Springer-Verlag, pp. 816-827.
5. G.N. Dixon and S.K. Shrivastava, "Exploiting Type-Inheritance Facilities to Implement Recoverability in Object-Based Systems," *Proc. Sixth Symp. Reliability in Distributed Software and Database Systems*, Mar. 1987, pp. 107-114.
6. R. Campbell, G. Johnston, and V. Russo, "Choices (Class Hierarchic Open Interface for Custom Embedded Systems)," *ACM Operating Systems Review*, Vol. 21, No. 3, July 1987, pp. 9-17.
7. A.Z. Spector et al., "The Camelot Project," *Database Engineering*, Vol. 9, No. 4, Dec. 1986.
8. B. Bergsten and R. Gonzalez-Rubio, "A Database Accelerator and Its Languages," *Proc. Conpar 88, BCS Workshop Series*, C.R. Jesshope and K.D. Reinartz, eds., Cambridge University Press, 1989, pp. 63-71.
9. H. Boral et al., "Prototyping Bubba, a Highly Parallel Database System," *IEEE Trans. Knowledge and Data Engineering*, Vol. 2, No. 1, Mar. 1990, pp. 4-24.
10. S.A. Delgado-Rannauro et al., "A Shared Environment Parallel Logic Programming System on Distributed Memory Architectures," ECRC document submitted to 2nd European Distributed Memory Computing Conference, (available from author), 1991.

11. C. Hammer and T. Henties, "Parallel Lisp for a Distributed Memory Machine," *Proc. Lisp Europal Workshop on High Performance and Parallel Computing in Lisp*, Ashok Gupta (ed.), Europal, Surrey, UK, Nov. 1990.
12. R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, New York, Vol. 7, No. 4, Oct. 1985, pp. 501-538.
13. T. Schneider, "The METAL System, Status 1987," *Proc. Machine Translation Summit II*, C. Rohrer and T. Schneider (ed.), Deutsche Gesellschaft für Dokumentation, Frankfurt, 1989, pp. 128-136.



Guy Haworth is the marketing manager for Information Servers within ICL's Computer Products Division. He has worked with ICL's database and CASE products, particularly the CAFS (Content Addressable File Store) search engine and is now responsible for the market focus and exploitation of the EDS project. He is interested in the performance and reliability of large-scale database systems.

Haworth holds an MA degree in mathematics from Oxford University and the Diploma in computer science from Cambridge University.



Steve Leunig, ICL's lead designer on the EDS database system, has worked as a programmer and lead designer on its VME operating system. Most recently, he participated in the Alvey Programme's Flagship Project, which developed a multiprocessor graph-reduction machine.

Leunig holds a BSc degree in mathematics from the University of Western Australia and an MSc degree in computing science from the University of London.



Carsten Hammer works in Siemens Corporate Research and Development in Munich. He leads the Languages for Parallel Computers Group and holds project responsibilities for the parallel EDS Lisp system. Previously, he led the activities for a vectorizing Pascal compiler.

Hammer received the Diploma in computer science and his doctorate from the Technical University of Braunschweig in Germany.



Mike Reeve currently leads the Computer Architecture Group at the European Computer Research Centre GmbH (ECRC). The group researches parallel declarative programming, distributed operating systems, and sequential and parallel machine architectures. In 1989 he accepted the Sony sabbatical chair, working at Sony's Computer Science Laboratory in Tokyo. For the five preceding years he held a SERC Advanced Research Fellowship at Imperial College, University of London.

Reeve obtained his BSc and PhD degrees in computer science from Imperial College.

Direct questions regarding this article to Guy Haworth, Marketing Manager, Information Servers, ICL, Kings House, 33 Kings Road, Reading, Berkshire, United Kingdom RG1 3PX.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159

Medium 160

High 161

SPAN

continued from p. 27

in which the `||` operator in example i) specifies that statements s_1 to s_n are to be performed in parallel. On one processor, this operation may be mapped into a series of processes activated in some undefined order. In ii), the `;` operator specifies sequential execution of the statements. Replicators exist for simplifying repetitive parallel or sequential statements. Examples iii) and iv) are alternatives to i) and ii). Parle only provides for synchronous execution of processes initiated by the `||` operator, in that these processes must all terminate before the initiating program can continue to the next statement.

The simplest conditional statement contains a single statement, which is executed when a guard condition is true. The most complex conditional statement contains multiple, guarded statements; the guards execute in parallel. When the statement associated with the first true guard executes, partially executed guards are discarded.

Although intended as a compiler target language (CTL), Parle is probably a better parallel programming language. When it is used as a CTL, Parle's weak typing places a large runtime checking burden on architectures not specifically designed to support the architectural model. The process model also has limitations, especially the limited control over processes and their synchronous execution. Process creation is statically defined at compilation time, and processes once initiated run to completion. The model cannot suspend or delete a process once it is started because there is no process by which an executing process can be identified. Neither does the model support dynamic process creation and process migration.

The process model is fine for a programming language, but it limits a CTL. The applications projects produced substantial amounts of code in Parle, which was easy to use and effective.

Virtual Machine Code

The VMC fully realizes the Kernel System model at the level of machine code or assembler. It provides a model of the Kernel System that can be ported onto a variety of parallel architectures. Consequently, the VMC is a low-level language that reduces the work of performing the port. The philosophy behind the design of the VMC was again to provide the necessary components to support the Kernel System while limiting complexity. Thus, the VMC has

- a reduced instruction-set style computational model.
- a small, simple instruction set with which more complex operations can be implemented. (The exception is that list operations are members of the instruction set, as lists are primitive elements.)