# Designing Highly Parallel Applications Using Database Programming Concepts

Maarten R. van Steen
TNO Institute of Applied Computer Science
P.O. Box 6032
NL-2600 JA, Delft, The Netherlands

Edwin de Jong
Dept. of Computer Science
University of Leiden
P.O. Box 9512
NL-2300 RA, Leiden, The Netherlands

## Abstract

*Current parallel programming models suffer from the serious drawback that they have evolved from models intended to describe collections of concurrent processes competing for common resources and services. The intention of concurrent systems is in conflict with the one of parallel programs, where processes cooperate to achieve a common goal rather than compete. It is therefore required to devise a sufficiently different programming model, that provides adequate abstraction over the problems related to parallel programming.*

*We propose a new parallel programming model which adopts common, content-addressable storage structures. The model advocates nonnavigational manipulation of data, an aspect which has been widely recognized in the database community. We draw a comparison with more conventional models based on shared variables and message passing. As we will show, a model based on common, content-addressable storage provides a substantial improvement over more conventional models.*

## 1 Introduction

Over the last twenty-five years concurrency has become one of the most active areas of research in computer science. Originally starting with the notion of coroutines and concurrent processes [9,14], concurrent programming models have been widely applied in the design of operating systems [13,18,20] and databases [11,23,24]. Moreover, as efficient implementations of high-level concurrent languages became available, software that was originally coded in an assembly language could now be developed using high-level language constructs yielding well-structured, efficient, and portable implementations. The main focus of these high-level concurrent models was to provide the proper means for specifying intricate systems of processes *competing* for common resources and services.

With the introduction of commercially available multiprocessor computers, concurrency has become an even more prolific area of research. As insight in the behaviour of concurrent models grew, focus has gradually shifted from the problem of developing programs that behave in a well-defined manner, to that of developing programs that exploit parallelism to improve overall efficiency. An important aspect is that, contrary to the original concurrent programming models, exploitation of parallelism is achieved by developing programs in which processes *cooperate* to achieve a common goal. This shift of focus has brought us, somewhat surprisingly, to a stage comparable to the first stages of research in concurrent models. At the moment, parallel applications are generally written in a highly machine-dependent manner and often violate basic rules of well-structured software in order to retain efficiency [16].

We feel that these problems originate from the attempts to extend concurrent programming models to solve problems in a domain for which these models are not suited. Rather than extending these models, new ones should be devised that are tailored to the specific problems related to parallel application development. In this paper, we present the specification language Vista, which incorporates a new parallel programming model strongly influenced by concepts originating in the field of database programming. Before introducing Vista, we first give a brief overview of the problems related to existing parallel programming models, and show that support for a higher level of abstraction is necessary. Following the presentation of Vista, we present a comparison to related research, including a discussion on the Linda parallel programming model that has also successfully incorporated concepts from the field of database programming. We conclude with some indications of future research.

## 2 Parallel programming models

Current parallel programming models can be generally organized into a scheme of two global classes [2,22]:

0-8186-2295-4/91 $01.00 © 1991 IEEE

- models in which communication is based on *shared variables*, and
- models in which communication is based on *message-passing*.

A finer classification can be made by considering the way communication and synchronization is supported. In this way, Bal et al. alone distinguish ten different programming paradigms for distributed systems [2]. Moreover, nearly all of the approximately 100 languages they discuss are based on direct-addressable memory, the most notable exception being Linda [6].

Concurrent programming models have originally been devised for developing resource management systems such as operating systems and database systems. As such, programs based on these models reflect the physical architecture of a system in which a collection of concurrent processes compete for shared resources and services. The competition between processes has always been formulated in terms of the two classes of communication mentioned above.

Current models that exploit parallelism have evolved from these concurrent programming models. However, parallel solutions to problems are by nature not at all related to the physical architecture of a (parallel) machine. Instead, they merely describe how problems can be solved by using a collection of cooperating processes aimed at achieving a common goal. The explicit existence of an underlying machine architecture in concurrent programming models enforces the particularities of a specific machine in parallel program development to be taken into account. What is required, instead, is a means to exclusively capture the solution to a problem and its inherent parallelism, which is completely independent of machine architectural features.

## 2.1 Models based on common content-addressable storage

Recent developments in the area of concurrency have led to a third and totally different programming model, referred to as *tuple space* [15], *shared dataspace* [10], or *blackboard* [4]. In essence, the model is based on concurrent processes that exchange data through common content-addressable storage structures. By absence of explicit communication patterns between concurrent processes, the latter act in a highly decoupled manner, which makes the parallel programming task considerably less complex [5].

Although the model has only been introduced recently in the area of concurrent programming, it has in fact, already received much attention within the database community. The concept of content-addressable storage structures has been studied and applied extensively in the form of relational data models ever since their introduction by Codd in the early seventies [8].

An important feature of a programming model based on content-addressable storage is that data can be manipulated by powerful, yet simple declarative languages which concentrate on expressing *which* data is needed by a process, rather than *how* the data is to be retrieved. The importance of this nonnavigational manipulation of data has been widely recognized within the database community and has led to a vast amount of research, exemplified in the field of deductive and knowledge-based systems (see e.g. [23]). Illustrative is the fact that even in the area of object-oriented database systems to which navigational access of objects seems inherent, support for declarative object manipulation in the form of relational-like query languages is growing [17].

The communication scheme supported by common, content-addressable storage has two important consequences. In the first place, there is no need for a process to be aware of the existence of other processes. This avoids the need to adopt communication and synchronization constructs such as shared variables, message-passing operations, (remote) procedure calls, etc, which are all based on assumptions concerning machine architectural features. In the second place, as processes act completely independent, parallelism can be exploited in abundance by replicating processes.

## 2.2 An example

To illustrate the implications of the various parallel programming models, we consider a parallel solution to the well-known single-source shortest path problem. The problem is to find the lengths of the shortest paths in a weighted, directed graph from a given source node $s$ to all other nodes in the graph. Let $w(u,v)$ denote the (non-negative) weight of the link from node $u$ to node $v$; if no such link exists then $w(u,v) = \infty$.

An initial solution to this problem proceeds as follows. Let $v.len$ denote the length of a path from node $s$ to node $v$. Initially, we set $v.len = w(s,v)$ for each node $v$. The algorithm replaces, for each pair of nodes $u$ and $v$, $v.len$ by $\min\{v.len, u.len+w(u,v)\}$ until no further changes in the value of each $v.len$ can occur. At that point, $v.len$ is known to contain the length of a shortest path from $s$ to $v$.

The issue now becomes to improve the algorithm in such a way that it can be efficiently executed on any sequential or parallel machine. Let us first consider the problem of detecting when the shortest paths have been found. A standard solution originally proposed by Moore [19], is to maintain a set $S$ of nodes that need to be inspected. Initially, $S = \{s\}$. The algorithm proceeds by removing a node $u$ from $S$ and examining each outgoing edge $(u,v)$. If $v.len > u.len + w(u,v)$, $v$ is added to $S$

and $v.len$ is replaced by $u.len + w(u,v)$. The algorithm terminates as soon as $S$ is empty.

## 2.3 A parallel solution based on shared variables

Let us now consider a parallel solution to the single-source shortest path problem, starting with a solution based on shared variables. Parallelism is obtained by creating a number of $p > 1$ asynchronous processes $P_1,\ldots,P_p$ that act in parallel on the set $S$ and all nodes of the graph. Both the set $S$ and the nodes of the graph are implemented as shared variables, which requires that two additional concerns are addressed.

First, exclusive access to the shared variables must be arranged. Otherwise, the same node might be removed from $S$ or inserted into $S$ by more than one process, or any $v.len$ might be simultaneously updated by multiple processes, yielding erroneous values.

The second concern to be addressed consists of properly detecting termination of all processes $P_1,\ldots,P_p$. Clearly, it is not appropriate to stop when a process finds the set $S$ to be empty, since other processes may still be examining nodes. Instead, it must be globally recorded which processes are waiting to remove a node from $S$. Only if all processes $P_1,\ldots,P_p$ are waiting, the computation is known to be terminated.

In designing a parallel solution based on shared variables, we experience that attention concentrates on preventing simultaneous updates on shared variables. By introducing some sort of mutual exclusion, focus subsequently shifts to the problem of how to prevent the shared variables from becoming a bottleneck in communication. Note that we do not address the latter issue in the solution outlined above. In addition, we see that termination detection must be explicitly hardwired into the parallel solution. It is clear that neither the latter nor the problem of mutual exclusion have anything in common with the originally proposed solution.

## 2.4 A parallel solution based on message-passing

A parallel solution that employs message-passing instead of shared variables, can be devised as follows. Again a number of $p$ identical processes are created. This time, the set of nodes is partitioned into $p$ subsets $N_1,\ldots,N_p$ such that process $P_i$ is responsible for maintaining the current information on all nodes in subset $N_i$. Furthermore, a separate process $P_S$ is created for managing the set $S$. Again two additional concerns must be explicitly addressed in this parallel solution.

The first concern is to set up an appropriate communication pattern between the processes. This might be done as follows. Process $P_S$ receives messages from each process $P_i$ that either contain a request for a node from $S$, or that contain a node $v$ to be added to $S$. Conversely, upon receipt of a node $u$ from $P_S$, process $P_i$ inspects each outgoing edge $(u,v)$. If $v \in N_i$, $P_i$ directly updates $v.len$ if necessary and sends $v$ to process $P_S$ for insertion into $S$. Otherwise, if $v \in N_j$, $i \neq j$, $P_i$ sends the value of $u.len + w(u,v)$ to process $P_j$ so that the latter can in turn update $v.len$ and send $v$ to process $P_S$ if necessary.

The second concern is again to detect termination of all processes. To that end, process $P_S$ records which processes are waiting for a node to be sent from $S$. If all processes $P_i$ are waiting, process $P_S$ broadcasts termination.

In designing a parallel solution based on message passing, we notice that attention shifts to the problem of devising appropriate communication patterns between asynchronous processes. Again we must conclude that the related concerns have nothing in common with the originally proposed solution.

```
TRANSACTION T_i:
    remove a node u from S;
    select a node v where v.len > u.len + w(u,v);
    if selection succeeded then
        v.len ← u.len + w(u,v);
        if v ∉ S then insert v into S;
    endif;
```

*Figure 1. A solution to the single-source shortest path problem by multiple transitions.*

## 2.5 A parallel solution based on common content-addressable storage

As we have seen, exploring a parallel solution based on shared variables or message-passing, leads to a significant divergence from the originally proposed solution. If, on the other hand, we adopt the notion of a common, content-addressable storage structure, a considerably more faithful and elegant parallel solution is obtained, as we shall illustrate below.

A solution to the single-source shortest path problem can be formulated in terms of a transaction $T$ that consists of two parts: (1) a query requesting nodes $u$ and $v$ that satisfy the constraint $v.len > u.len + w(u,v)$, and (2) a subsequent update of $v.len$. A parallel solution is obtained by simply introducing multiple instances of transaction $T$, concurrently acting on the set $S$ and the nodes of the graph. Using pseudo-code, each transaction $T_i$ can be defined as indicated in Figure 1. This solution can be specified even more concisely by omitting any details concerning the set $S$ as shown in Figure 2.

In the solutions presented in Figure 1 and 2, we make explicit use of the atomicity property of transactions, meaning that [3]:

- transactions access shared data without mutual interference, and
- upon normal termination, all effects of a transaction are made permanent, otherwise the transaction has no effect whatsoever.

We argue that these simple semantics of transactions, combined with the notion of common, content-addressable storage, provide a powerful means for expressing highly parallel solutions. Also important is the fact that the parallel solutions are not biased towards machine architectural features, so that they can be expressed at an adequate level of abstraction.

TRANSACTION $T_i$:
select nodes $u$ and $v$ where
$v.len > u.len + w(u,v)$;
if selection succeeded then
replace $v.len$ by $u.len + w(u,v)$;

*Figure 2. A solution to the single-source shortest path problem without explicit use of a set S.*

# 3 Vista: a query-based parallel specification language

We introduce the specification language Vista, which has been devised for the design of highly parallel applications. In Vista we employ common, content-addressable storage structures in a way strongly influenced by database languages. In this section, we give a brief introduction to Vista; for a more thorough introduction, the reader is referred to [12].

Vista is a visually oriented language, which means that its language constructs are graphical components, sometimes annotated by a textual representation. The total number of language constructs is small and concise. Giving a tutorial introduction to the various language constructs, we return to the single-source shortest path problem as discussed in the previous section.

A Vista program that employs a solution to the single-source shortest path problem similar to the one outlined in the previous section, is presented in Figure 3. In fact, the Vista program is a formal representation of the transaction given by pseudo-code in Figure 2. As in every Vista program, its basic ingredients consist of *data*, *processing*, and query-based *communication*. In accordance to these three categories, we elucidate the program in Figure 3, which in fact, contains all basic language constructs provided by Vista.

## 3.1 Data and storage

In Vista the data manipulated in a program always resides at *storage places*. The program in Figure 3 features a single storage place named "graph," containing the nodes of a graph for which the shortest paths have to be found. Generally a storage place may contain an arbitrary but finite amount of data. A storage place is content-addressable, which means that data is retrieved from a storage place based on selected properties, rather than by direct addressing.

Each type of data manipulated in a program is represented by a unique *data descriptor*. Specifying data descriptors is analogous to the definition of a conceptual scheme in database systems. Figure 3 contains a data descriptor named "node" which represents a node in the graph.

A data descriptor strongly resembles the notion of a relation in the relational data model, in the sense that *attributes* can be attached. For example, the data descriptor "node" is attributed a pair of fields named "id" and "len" respectively. Attribute "id" identifies the corresponding node in the graph, whereas attribute "len" indicates the length of a path originating in the source node $s$. The main difference with the relational data model is that attributes in Vista may also range over complex domains, such as the set $\{\mathbf{R}^n \rightarrow \mathbf{R} \mid n \geq 0\}$ of real-valued functions, or the power set $\wp(\mathbf{N})$ of natural numbers. Vista allows only value-oriented data modelling; there is no support for identifying objects as offered, for example, in hierarchical and network-based data models.

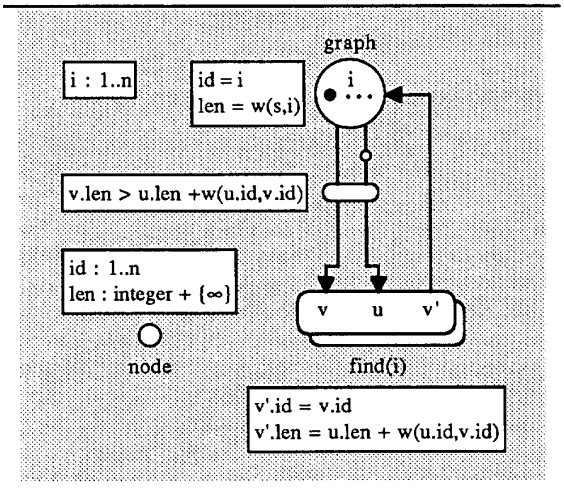


*Figure 3. A Vista program for solving the single-source shortest path problem.*

Data descriptors are used to derive the actual *data items* manipulated in a program. A data item inherits the attributes from its associated data descriptor, and assigns a specific value to each attribute. The program in Figure 3 contains a collection of data items, corresponding to the nodes $v_1,\ldots,v_n$ of a given graph. The nodes are generically specified as a *replicated* data item. Replication is indicated by an ellipsis, annotated by a typed variable called the *replication index*. In this case, replication index $i$ ranges over nodes $v_1,\ldots,v_n$. Initially, attribute "id" of each data item is set equal to a unique node in the graph, whereas attribute "len" is set equal to $w(s,i)$, the weight of the link from source $s$ to node $i$.

## 3.2 Processing

The computation or processing performed by a program is represented by *operations*. The program in Figure 3 contains an operation named "find" which updates the length of a path to a given node $v$. The computation performed by an operation is modelled as an atomic transformation from input into output. The input is accepted through a (possibly empty) set of *input ports*. Likewise, the output is generated by a (possibly empty) set of *output ports*. In Figure 3, operation "find" consists of a pair input ports labelled "v" and "u," and a single output port labelled "v'."

The actual computation performed by an operation is expressed by an associated set of *constraints*. A constraint is represented as a first-order formula which specifies a relation between input and output. The free variables that occur in the formula are of the form $p.a$, where $p$ is the name of a port, and $a$ is an attribute. For instance, the update performed on the length of a path to node $v$ by operation "find" is expressed by the constraint "v'.len = u.len + w(u.id, v.id)."

## 3.3 Query-based communication

A means of communication between storage places and operations is established by *links*. A link always extends from a storage place to an input port, or conversely, from an output port to a storage place. The program in Figure 3 features a total of three links connecting operation "find" to storage place "graph."

The links connected to the input ports of an operation are grouped into a single *channel*, that serves as a guard to the operation. Channels exploit the content-addressable nature of storage places by communicating data items on the basis of selected properties. Similar to operations, a set of constraints can be associated with a channel, which strongly resembles the notion of queries used in database programming. Only data items that satisfy the constraints are considered suitable candidates for communication. For instance, the channel connected to operation "find" in Figure 3 selects a shorter path by means of the constraint "v.len > u.len + w(u.id, v.id)."

## 3.4 The execution model

Having outlined the basic language constructs that comprise a Vista program, we now turn to its underlying execution model. The execution model as we present it here, is based on non-deterministic interleaving, and was partially inspired by [7].

The execution of a Vista program can be briefly stated as follows. Execution starts from the initial data items. On each execution step, an operation is selected non-deterministically and executed. This step is repeated indefinitely, subject to the restraint that selection is fair, that is, each operation is selected for execution infinitely often.

An operation is executed as follows. A distinct data item is selected at the source of each link connected to an input port of the operation. The selected data items must satisfy the constraints attached to the channel in which the links are grouped. If sufficient data items cannot be selected, then execution of the operation is simply skipped. Otherwise, the selected data items are deleted, and a new data item is inserted at the destination of each link connected to an output port of the operation. It is required that the newly inserted data items satisfy the constraints attached to the operation. Note that generally an operation exhibits non-deterministic behaviour, since the attached condition need not define a unique relation between input and output.

Termination of a Vista program is defined as a side-effect of the execution model. Execution is said to be *terminated* if each operation must be skipped for execution. In that case selection continues indefinitely, but will, from that moment on, never result in actual execution. Note that in accordance to the execution model, the program in Figure 3 terminates when the lengths of all shortest paths have been found, as can be readily deduced from the query "v.len > u.len + w(u.id, v.id)."

## 3.5 Parallelism by replication

Parallelism inherent to the solution represented by the program in Figure 3, can be fully exploited by introducing multiple instances of operation "find." This is similar to the notion of having multiple transactions as earlier presented in Figure 2.

In Vista, multiple instances of an operation are indicated by drawing a "stacked" operation. In Figure 3 this notation is used to denote multiple instances of operation "find," which act asynchronously and in parallel on the nodes of the graph. Also other types of replication are supported by Vista to actuate a large number of

identical operations acting in synchronised fashion on a shared datastructure.

We emphasize that the Vista program in Figure 3 is strictly declarative in nature. It merely states the computations to be performed on a common storage structure and their respective antecedents. In particular, the program does not state how the operations retrieve and store parts of the data structure, nor in what order. Due to this nonnavigational manipulation of data, potential parallelism in the solution represented by the program is fully preserved and uncovered.

## 4 Related research

The notion of a common, content-addressable storage is recently receiving somewhat more attention from researchers in the field of parallel processing. The most successful model so far is that of the Linda tuple space introduced by Gelernter et al. [1,5,15], and recently publications have appeared on the Swarm model [10,21,22]. In this section we concentrate on the Linda programming model, and compare this model to the approach followed by Vista.

### 4.1 The Linda tuple space model

In Linda, processes interact by reading, removing, and inserting tuples into a common storage called tuple space. A tuple is an ordered collection of typed fields. For example, the tuple ("str", 3.14, 2) consists of a string, a real, and an integer. When a process executes the statement *out*("str", 3.14, 2), the tuple is generated and added to the tuple space without blocking the process. Tuples can be removed from tuple space by the operation *in*. For example, if a process executes the statement *in*("str", ? *x*, 2) the tuple space is searched for a tuple with three elements which matches exactly on the first and third element, and which matches on the type of the second element. If no such tuple is found, the process blocks until a matching tuple appears. The statement *rd*("str", ? *x*, 2) works similar to the *in* statement except that a matching tuple is copied rather than removed from the tuple space.

Linda can also be used to create so-called live data structures. Each process in a live data structure program computes a part of the data structure to be built and subsequently transforms itself into a passive element of the intended data structure (see [5] for further details). Live data structures can be created by using the *eval* statement. For instance, the statement *eval*("str", *i*, *compute*(*i*)) creates a process that eventually adds a tuple ("str", *i*, *res*) to the tuple space, where *res* is the result of the function *compute*(*i*).

Linda has been successfully implemented on a number of parallel machines, including shared-memory as well as distributed-memory configurations. The model has been added to several standard languages such as C, Fortran, but also to object-oriented languages such as Eiffel, and functional languages such as Scheme.

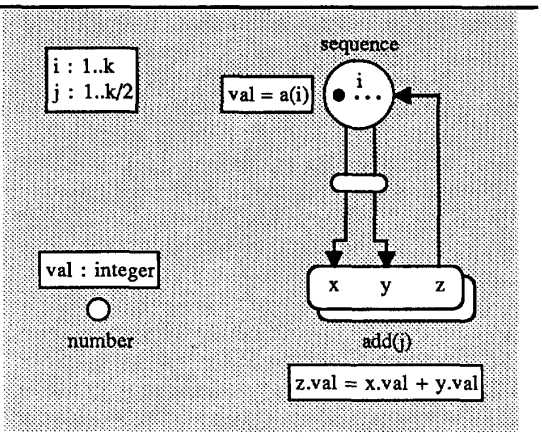


*Figure 4. A Vista program for adding a sequence of numbers.*

### 4.2 A comparison between Vista and Linda

Like Vista, the Linda programming model strongly resembles relational database programming models. The tuple space can be viewed as a relational database in which queries and updates are formulated in terms of the operations *rd*, *in*, and *out*. However, despite the fact that the simplicity of the model provides a powerful means to construct parallel applications in which architectural features of the target machine can be neglected, the model lacks some important features that are essential to programming models based on a common, content-addressable storage.

In essence the problems with Linda originate from the fact that processes cannot query more than one tuple from tuple space at a time. To illustrate this, we consider the simple problem of adding a sequence of numbers $a(1),\ldots,a(k)$. A Vista program to solve this problem is given in Figure 4. The sequence is stored as a collection of data items of type "number," neglecting the particular order of the sequence. The replicated operation "add" repetitively inquires a pair of numbers from storage place "sequence," after which it reinserts their sum. Eventually the computation terminates, leaving exactly one number at storage place "sequence," which value equals the sum of the given sequence.

An obvious comparable implementation in Linda would be to create a total of $k-1$ processes, each executing the following sequence of statements.

*in*("number", ? *x*);
*in*("number", ? *y*);
*out*("number", *x* + *y*)

A serious obstacle in this simple Linda program is the immediate dependence of the number of processes on the length of the sequence of numbers. For instance, a potential deadlock results if the number of processes created is chosen larger or equal than the length of the sequence. This might pose a serious problem if the length of the sequence is not known in advance but rather depends on other computations to be carried out. Of course, various solutions can be devised to correct this problem, but none of these will be inherent to the Linda model. It is our opinion that this simple example illustrates a serious shortcoming in the expressive power of Linda. For instance, the Vista program in Figure 3 to solve the single-source shortest path problem cannot be directly expressed in Linda. Instead, we are again forced to explicitly search for shorter paths, and to devise a solution for detecting program termination.

## 5 Concluding remarks

Current parallel programming models have evolved from models devised for specifying systems in which a collection of concurrent processes compete for common resources and services. This view on the behaviour of a concurrent system conflicts the one of a parallel system in which processes cooperate rather than compete. This conflicting view has resulted in parallel programs in which it is extremely difficult to combine efficiency, portability, and well-structuredness. These difficulties can only be overcome if parallel programming models are devised that allow a higher level of abstraction.

We proposed to adopt a parallel programming model based on common, content-addressable storage, a model which has already been used within the database community for a number of decades. The model advocates a nonnavigational approach towards data retrieval, which we have shown to preserve and uncover inherent parallelism. We introduced the visually oriented specification language Vista to express highly parallel applications, using a programming style similar to many database programming languages currently in use.

Our current research concentrates on developing efficient implementation techniques for parallel programs expressed in Vista. Some small experiments have been conducted in implementing Vista programs on sequential architectures using logic-based and object-oriented languages. Our main focus, however, is to develop efficient distributed algorithms for retrieving data items on the basis of selected properties. The first results based on a 16-node transputer system are promising, but much research is yet to be done.


## Acknowledgements

The work described in this paper has been undertaken as part of the ParTool project. ParTool is a collaborative project of Dutch industry and universities aiming at the development of a parallel programming environment. The project is sponsored by the Dutch agency SPIN.


## References


[1] S. Ahuja, Carriero, N. and Gelernter, D., "Linda and Friends." *Computer* August (1986), pp. 26-34.

[2] H.E. Bal, Steiner, J.G. and Tanenbaum, A.S., "Programming Languages for Distributed Computing Systems." *Computing Surveys 21*, 3 (1989), pp. 261-322.

[3] P.A. Bernstein, Hadzilacos, V. and Goodman, N., *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] A. Brogi and Ciancarini, P., "The Concurrent Language, Shared Prolog." *ACM Transactions on Programming Languages and Systems 13*, 1 (1991), pp. 377-387.

[5] N. Carriero and Gelernter, D., "How to Write Parallel Programs: A Guide to the Perplexed." *Computing Surveys 21*, 3 (1989), pp. 323-358.

[6] N. Carriero and Gelernter, D., "Linda in Context." *Communications of the ACM 32*, 4 (1989), pp. 444-458.

[7] K.M. Chandy and Misra, J., *Parallel Program Design: A Foundation.* Addison-Wesley, 1989.

[8] E.F. Codd, "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM 13*, 6 (1970), pp. 377-387.

[9] M. Conway, "A Multiprocessor System Design." In *Proceedings AFIPS Fall Joint Computer Conference*, 1963, pp. 139-146.

[10] H.C. Cunningham, "The Shared Dataspace Approach to Concurrent Programming: The Swarm Programming Model, Notation, and Logic," Ph.D. Thesis, Department of Computer Science, Washington University, St. Louis, MO., 1989.

[11] C.J. Date, *An Introduction to Database Systems.* Addison-Wesley, 1977.

[12] E. de Jong, Kuijlman, F. and van Steen, M.R., "An Introduction to Vista." Tech. Rep. TNO Institute of Applied Computer Science, Delft, 1991.

[13] H.M. Deitel, *An Introduction to Operating Systems.* Addison-Wesley, 1984.

[14] E.W. Dijkstra, "Cooperating Sequential Processes." In *Programming Languages*. Academic Press, F. Genuys (ed.), 1968.

[15] D. Gelernter, "Generative Communication in Linda." *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), pp. 80-112.

[16] A.H. Karp, "Programming for Parallelism." *Computer* May (1987), pp. 43-57.

[17] W. Kim, "Object-Oriented Databases: Definition and Research Directions." *IEEE Transactions on Knowledge and Data Engineering 2*, 3 (1990), pp. 327-341.

[18] M. Maekawa, Oldehoeft, A.E. and Oldehoeft, R.R., *Operating Systems: Advanced Concepts*. Benjamin/Cummings, 1987.

[19] E.F. Moore, "The Shortest Path Through a Maze." In *Proceedings International Symposium on the Theory of Switching*, 1957, pp. 285-292.

[20] J.L. Peterson and Silberschatz, A., *Operating Systems Concepts*. Addison-Wesley, 1983.

[21] G.C. Roman and Cunningham, H.C., "A Shared Dataspace Model of Concurrency - Language and Programming Implications." In *Proceedings 9th IEEE International Conference on Distributed Systems*, 1989, pp. 270-279.

[22] G.C. Roman and Cunningham, H.C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency." *IEEE Transactions on Software Engineering 16*, 12 (1990), pp. 1361-1373.

[23] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Principles of Computer Science Series, 1989.

[24] G. Wiederhold, *Database Design*. McGraw-Hill, 2nd ed. 1983.