Task Allocation Algorithms for Distributed Data Base and Distributed Computing System

Dr. Virender Khurana
Senior Lecturer, Vaish College of Enginerring, Former Coordinator and Head, Vaish College Rohtak

Lovely Mittal
Ex-Lecturer Vaish College Rohtak

Manoj Garg
Senior Lecturer, Vaish College of Enginerring, Former Coordinator and Head, Vaish College Rohtak

Abstract

In this paper, we describe an algorithm to detect a stable property for a dynamic distributed system that does not suffer from any of the limitations described above. Our approach is based on maintaining a spanning tree of all processes currently participating in the computation. The spanning tree, which is dynamically changing, is used to collect local snapshots of processes periodically. Processes can join and leave the system while a snapshot algorithm is in progress. We identify sufficient conditions under which a collection of local snapshots can be safely used to evaluate a stable property. Specifically, the collection has to be consistent (local states in the collection are pair-wise consistent) and complete (no local state necessary for correctly evaluating the property is missing from the collection). We also identify a condition that allows the current root of the spanning tree to detect termination of the snapshot algorithm even if the algorithm was initiated by an "earlier" root that has since left the system. Due to lack of space, formal description of our algorithm and proofs of various lemmas and theorems have been omitted and can be found There are basically two major kinds of modern architectures: two-tier client/server and three-tier—also commonly called n-tier. Each one has many variations. At a high level, these architectures focus on the partitioning system processing. They decide on what machine and in what process space a given bit of code executes. Client/server is often a generic umbrella term for any application architecture that divides processing among two or more processes, often on two or more machines. Any database application is a client/server application if it handles data storage and retrieval in the database process and data manipulation and presentation somewhere else. The server is the database engine that stores the data, and the client is the process that gets or creates the data. The idea

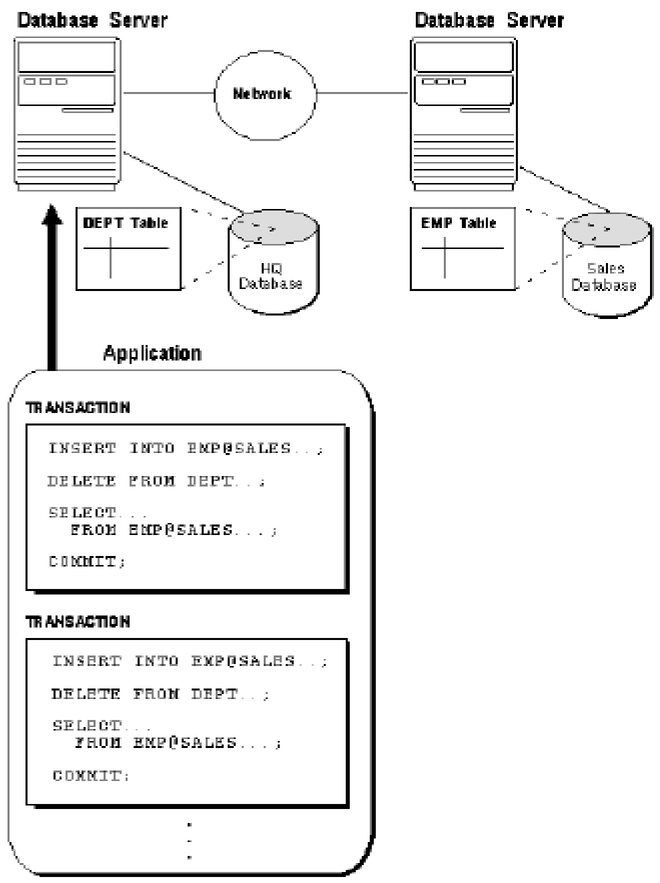
behind the client/server architecture in a database application is to provide multiple users with access to the same data.

Introduction

Distributed computing is a method of computer processing in which different parts of a program are run simultaneously on two or more computers that are communicating with each other over a network. Distributed computing is a type of segmented or parallel computing, but the latter term is most commonly used to refer to processing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing require that a program be segmented—divided into sections that can run simultaneously, distributed computing also requires that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components. An example of distributed computing is BOINC, a framework in which large problems can be divided into many small problems which are distributed to many computers. Later, the small results are reassembled into a larger solution. Distributed computing is a natural result of using networks to enable computers to communicate efficiently. But distributed computing is distinct from computer networking or **fragmented** computing. The latter refers to two or more computers interacting with each other, but not, typically, sharing the processing of a single program. The World Wide Web is an example of a network, but not an example of distributed computing. There are numerous technologies and standards used to construct distributed computations, including some which are specially designed and optimized for that purpose, such as Remote Procedure Calls (RPC) or Remote Method Invocation (RMI) or

Direct and Indirect Connections

A client can connect directly or indirectly to a database server. In *Figure 1*, when the client application issues the first and third statements for each transaction, the client is connected directly to the intermediate HQ database and indirectly to the SALES database that contains the remote data.



```
INSERT INTO EMP@SALES..;
DELETE FROM DEPT..;
SELECT...
  FROM EMP@SALES...;
COMMIT;
```

```
INSERT INTO EMP@SALES..;
DELETE FROM DEPT..;
SELECT...
  FROM EMP@SALES...;
COMMIT;
```

Figure 1 an Example of a Distributed DBMS Architecture Site Autonomy

we use the letters A, B, C, D, E and F to refer to a collection of events and the letters U, V , W, X, Y and X to refer to a collection of local states. For a collection of local states X, let processes(X) denote the set of processes whose local state is in X. Also, let events(X) denote the set of events that have to be executed to reach local states in X. Two local states x and y are said to be consistent if, in order to reach x on process(x), we do not have to advance beyond y on process(y), and vice versa.

Loading Calculations

Researches have shown that even in such homogeneous distributed systems, statistical fluctuations in the arrival of tasks and task service time requirements at computers lead to the high probability that at least one computer is idle while a task is waiting for service elsewhere. Their analysis, presented next, models a computer in a distributed system by an ***M/M/1*** server. Consider a system of *N* identical and independent M/M/1 servers. By identical we mean that all servers have the same task arrival and service rates. Let ***a*** be the utilization of each server. Then P0 = 1 – ***a*** is the probability that a server is idle. Let ***P*** be the probability that the system is in a state in which at least one task is waiting for service at least one server is idle. Then ***P*** is given by the expression

N P = Σ (*N* i)Qi *H N*-I i=1

Where Qi is the probability what a given set of I servers are idle and HN-i is the probability that a given set of (N – i) servers are not idle and at one or more of them ***a*** task is waiting for service. Clearly, from the independence assumption, **Qi = Pi o**

HN-1 = {probability that N – i} systems have at least one task} – {probability that all (N-i) systems have exactly one task}

HN-i = (1 – Po) N-i – [{1 – Po } Po] N-i

Classification of Load Distributing Algorithms

The basic function of a load-distributing algorithm is to transfer load (tasks) from heavily loaded computers to idle or lightly loaded computers. Load distributing algorithms can be broadly characterized as static, dynamic or adaptive. Dynamic load distributing algorithms use system state information (the loads at nodes), at least in part, to make load-distributing decisions, while static algorithms make no use of such information. In static load distributing algorithms, decisions are hard-wired in the algorithm using priori knowledge of the system. Dynamic load distributing algorithms have the potential to outperform static load distributing algorithms tributing algorithms have the potential to outperform static load distributing algorithms because they are able to exploit short term fluctuations in the system state to improve performance. However, dynamic load distributing algorithms entail overhead in the collection, storage, and analysis of system state information. Adaptive load distributing algorithms are special class of dynamic load distributing algorithms in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state. For example, a dynamic algorithm may continue to collect the system state irrespective of the system load. An adaptive algorithm, on the other hand, may discontinue the collection of the system state if the overall system load is high to avoid imposing additional overhead on the system. At such loads, all nodes are likely to be busy and attempts to find receivers are unlikely to be successful.

Load Balancing Vs. Load Sharing

Load distributing algorithms can further be classified as load balancing or load sharing algorithms, based on their load distributing principle. Both types of algorithms strive to reduce the likelihood of an unshared state (a state in which one computer lies idle while at the same time tasks contend for service to another computer) by transferring tasks to lightly loaded nodes. Load balancing algorithms, however, go a step further by attempting to equalize loads at all computers. Because a load balancing algorithm transfers tasks to a higher rate than a load-sharing algorithm, the higher overhead incurred by the load balancing algorithm may outweigh this potential performance improvement. Task transfers are not instantaneous because of communication delays and delays that occur during the collection of task state. Delays in transferring a task increase the duration of an unshared state, as an idle computer must wait for the arrival of the transferred task. To avoid lengthy unshared states, anticipatory task transfers from overloaded computers to computers that are likely to become idle shortly can be used. Anticipatory transfers increase that task transfer rate of a load-sharing algorithm, making it less distinguishable from load balancing algorithms. In this sense, load balancing can be considered a special case of load sharing, performing a particular level of anticipatory task transfers.

Selecting a suitable load-sharing algorithm

Based on the performance trends of load sharing algorithms, one may select a load sharing algorithm that is appropriate to the system under consideration as follows:

1. If the system under consideration never attains high loads, sender-initiated algorithms will give an improved average response time over no load sharing at all.

2. Stable scheduling algorithms are recommended for systems that can reach high loads. These algorithms perform better than non-adaptive algorithms for the following reasons:

a. Under sender-initiated algorithms, an overloaded processor must send inquiry messages delaying the existing tasks. If an inquiry fails, two overloaded processors are adversely affected because of unnecessary message handling. Therefore, the performance impact of an inquiry is quit severe at high system loads, where most inquiries fail.

b. Receiver-initiated algorithms remain effective at high loads but require the use of preemptive task transfers. Note that preemptive task transfers are expensive compared to non-preemptive task transfers because they involve saving and communicating a far more complicated task state.

3. For a system that experiences a wide range of load fluctuations, the stable symmetrically initiated scheduling algorithm is recommended because it provides improved performance and stability over the entire spectrum of system loads.

4. For a system that experiences wide fluctuations in load and has a high cost for the migration of partly executed tasks, stable sender-initiated algorithms are recommended, as they perform better than unstable sender-initiated algorithms at all loads, perform better than receiver-initiated algorithms over most system loads, and are stable at high loads.

5. For a system that experiences heterogeneous work arrival, adaptive stable algorithms are preferable, as they provide substantial performance improvement over non-adaptive algorithms.

Spanning Tree Maintenance Algorithm

Processes may join and leave the system while an instance of the snapshot algorithm is in progress. Therefore spanning tree maintenance protocols, namely control join and depart protocols, have to design carefully so that they do not “interfere” with an ongoing instance of the snapshot algorithm. To that end, we maintain a set of invariants that we use later to establish the correctness of the snapshot algorithm. Each process maintains information about its parent and its children in the tree. Initially, before a process joins the spanning tree, it does not have any parent or children, that is, its parent variable is set to nil and its children-set is empty. Let x be a local state of process p. We use parent(x) to denote the parent of p in x and children(x) to denote the set of children of p in x. Also, let status(x) denote the status of p in x. Further, p is said to be root in x if parent(x) = p. For a collection of local states X and a process p $\in$ processes(X), we use X.p to denote the local state of p in X. Now, we describe our invariants. Consider a comprehensive state X and let p and q be two processes in X. The first invariant says that if the status of a process is either IN or TRYING, then its parent variable should have a non-nil value. Formally, status(X.p) $\in$ {IN,TRYING} $\Rightarrow$ parent(X.p) = nil (1) The second invariant says that if a process considers another process to be its parent then the latter should consider the former as its child. Moreover, the parent variable of the latter should have a non-nil value. Intuitively, it means that child “relationship” is maintained for a longer duration than parent “relationship”. Further, a process cannot set its parent variable to nil as long as there is at least one process in the system, different from itself, that considers it to be its parent. Formally, $(parent(X.p) = q) \wedge (p = q) \Rightarrow (p \in children(X.q)) \wedge (parent(X.q) = nil)$

The third invariant specifically deals with the departure of a root process.
To distinguish between older and newer root processes, we associate a rank with every root process. The rank is incremented whenever a new root is selected. This invariant says that if two processes consider themselves to be root of the spanning tree, then there cannot be a process that considers the "older" root to be its parent. Moreover, the status of the "older" root has to be DEPARTING. Formally, we now describe our control join and depart protocols that maintain the invariants joining the Spanning Tree: A process attaches itself to the spanning tree by executing the control join protocol. Our control join protocol is quite simple. A process wishing to join the spanning tree first obtains a list of processes that are currently part of the spanning tree. This, for example, can be achieved using a name server. It then contacts the processes in the list, one by one, until it finds a process that is willing to accept it as its child. We assume that the process is eventually able to find such a process, and, therefore, the control join protocol eventually terminates successfully. Leaving the Spanning Tree: A process detaches itself from the spanning tree by executing the control depart protocol. The protocol consists of two phases. The first phase is referred to as trying phase and the status of process in this phase is TRYING. In the trying phase, a departing process tries to obtain permission to leave from all its tree neighbors (parent and children). To prevent neighboring processes from departing at the same time, all departure requests are assigned timestamps using logical clock. A process, on receiving departure request from its neighboring process, grants the permission only if it is not de parting or it's depart request has larger timestamp than that of its neighbor.
This approach is similar to Ricart and Agrawala' s algorithm modified for drinking philosopher's problem Note that the neighborhood of a departing process may change during this phase if one of more of its neighbors are also trying to depart. Whenever the neighborhood of a departing process changes it sends its departure request to all its new neighbors, if any. A process wishing to depart has to wait until it has received permission to depart from its current neighbors. We show in that the first phase of the control departs protocol eventually terminates. Once that happens, the process enters the second phase. The second phase is referred to as departing phase and the status of process in this phase is DEPARTING. The protocol of the departing phase depends on whether the departing process is a root process. If the departing process is not a root process, then, to maintain the spanning tree, it attaches all

its children to its parent. On the other hand, if it is a root process, then it selects one its children to become the new root. It then attaches all its other children to the new root. The main challenge is to change the spanning tree without violating any of the invariants. Case 1 (when the departing process is not the root): In this case, the departing phase consists of the following steps:

– Step 1: The departing process asks its parent to inherit all its children
and waits for acknowledgment.

– Step 2: The departing process asks all its children to change their parent to its parent and waits for acknowledgment from all of them. At this point, no process in the system considers the departing process to be its parent.

– Step 3: The departing process terminates all its neighbor relationships.

At this point, the parent of the departing process still considers the process to be its child.

– Step 4: The departing process asks its parent to remove it from its set of children and waits for acknowledgment.

Case 2 (when the departing process is the root): In this case, the departing phase consists of the following steps:

– Step 1: The departing process selects one of its children to become the new root. It then asks the selected child to inherit all its other children and waits for acknowledgment.

– Step 2: The departing process asks all its other children to change their parent to the new root and waits for acknowledgment from all of them. At this point, only the child selected to become the new root considers the departing process to be its parent.

– Step 3: The departing process terminates child relationships with all its other children. The child relationship with the child selected to become the new root cannot be terminated as yet.

– Step 4: The departing process asks the selected child to become the new root of the spanning tree and waits for acknowledgment. At this point, no process in the system considers the departing process to be its parent.

Step 5: The departing process terminates all its neighbor relationships.

To ensure likeness of the snapshot algorithm, we require the departing process to “transfer” the latest set of local states it has collected so far (which may be empty) to another process, after it has detached itself from the spanning tree but before leaving the system permanently. The process to which the collection has to be “transferred” is the parent of the departing process in the first case and the new root of the spanning tree in the second case. In both cases, the process to which the collection is “transferred” has to wait until it has received the collection from all processes it is supposed to before it can itself enter the departing phase.

The Snapshot Algorithm

As discussed earlier, it is sufficient to collect a consistent set of local states that is complete with respect to some comprehensive state. We next discuss how consistency and completeness can be achieved. For convenience, when a process records it local state, we say that it has taken its snapshot. Achieving Consistency: To achieve consistency, we use Lai and Yang's approach for taking a consistent snapshot of a static distributed system Each process maintains the instance number of the latest snapshot algorithm in which it has participated. This instance number is piggybacked on every message it sends application as well as control. If a process receives a message with an instance number greater than its own, it first records its local state before delivering the message. It can be verified that: Theorem 3 (consistency). Two local states belonging to the same instance of the snapshot algorithm are consistent with each other.

Achieving Completeness: As explained earlier to be able to evaluate a property for a collection of local states, it is sufficient for the collection to be complete with respect to some comprehensive state. The main problem is: "How does the current root of the spanning tree know that its collection has become complete?" To solve this problem, our approach is to define a test property that can be evaluated locally for a collection of local states such that once the test property evaluates to true then the collection has become complete. To that end, we define the notion of f-closed collection of local states.

The collection of local states returned by the snapshot algorithm is (1) consistent and (2) complete with respect to some comprehensive state. The likeness of the snapshot algorithm is only guaranteed if the system be- comes permanently quiescent eventually (that is, the set of processes does not change). Other algorithms for property detection make similar assumptions to achieve liveness Without this assumption, the spanning tree may continue to grow forcing the snapshot algorithm to collect local states of an ever in- creasing number of processes. To ensure liveness under this assumption, we make the following enhancements to the basic snapshot algorithm. First, whenever a process records its local state, it sends a marker message containing the current instance number to all its neighbors. In addition, it sends a marker message to any new neighbor whenever its neighborhood set changes. Second, whenever its parent changes, it sends its collection to the new parent if the collection has become _-closed. Third, just before leaving the system, a process transfers its collection to one of its neighbors as explained earlier. Once the system becomes permanently quiescent, the first modification ensures that all processes in the tree eventually record their local states and the second modification ensures that the collection at the root eventually becomes _-closed. It can be proved that. Assuming that the system eventually becomes permanently quiescent (that is, the set of processes does not change), every instance of the snapshot algorithm terminates eventually. we present an efficient algorithm to determine whether a stable property has become true in a dynamic distributed system in which processes can join and leave the system at any time. Our approach involves periodically collecting local states of processes that are currently part of the system using a (dynamically changing) spanning tree. There are several interesting problems that still need to be addressed. The depart protocol described in the paper has relatively high worst-case depart latency. Specifically, a process may stay in the trying phase for a long period of time (because of other processes joining and leaving the system) before it is able to enter the departing phase. An interesting problem is to design a depart protocol that has low worst-case depart latency. Also, in our current approach, control neighbors of a process may be completely different from its application neighbors, which may be undesirable in certain cases. Finally, in this paper, we assume that processes are reliable and they never fail. It would be interesting to investigate this problem in the presence of failures.

Conclusion

This paper has focused on using the model for monitoring; other papers discuss information specification and relationships, and their attributes are specified in the program construction system when the parallel application is designed and implemented. Later, views are specified on entities and relationships to describe the desired monitoring information, to be used, for example, for adaptation. The low level distributed collection and analysis mechanisms can then be generated automatically from these high level specifications. Our approach involves periodically collecting local states of processes that are currently part of the system using a (dynamically changing) spanning tree. There are several interesting problems that still need to be addressed. The depart protocol described in the paper has relatively high worst-case depart agency. Specifically, a process may stay in the trying phase for a long period of time (because of other processes joining and leaving the system) before it is able to enter the departing phase. An interesting problem is to design a depart protocol that has low worst-case depart latency. Also, in our current approach, control neighbors of a process may be completely different from its application neighbors, which may be undesirable in certain cases. Finally, in this paper, we assume that processes are reliable and they never fail. It would be interesting to investigate this problem in the presence of failures. Server load balancing is a powerful technique for improving application availability and performance in service provider, web content provider and enterprise networks, but piecemeal implementation can also increase network cost and complexity. Server load balancing devices that don't operate at wire speed can also create their own performance bottlenecks. Extreme Networks provides the key benefits of server load balancing while eliminating the potential cost, complexity, and performance issues. By fully integrating server load balancing into its wire-speed multilayer switches, Extreme Networks eliminates the need for the extra "islands of functionality" that increase cost and complexity. Even with all server load-balancing functions enabled, Extreme Networks switches continue to operate at wire speed on every port and will not become a bottleneck.

The concept of Information Retailing can be perhaps best explained by drawing an analogy to the world of consumer goods. In that world, retail stores provide a valuable service to shoppers by neatly organizing and displaying products in a manageable shopping space. Goods are organized into like categories and generally, the retail outlet is organized to make it easy for consumers to find what they want and even to make the "shopping" experience as pleasurable as possible. Information retailing software is designed to provide the same type of intuitive and satisfying experience for today's "data shoppers." These shoppers are executives, managers, and analysts who need to have their hands on the pulse of their business' key performance variables. Data consumers expect that the data they need will be accessible in a unified environment and that it will be organized into meaningful categories and names. Load balancing systems help optimize server workloads in virtual data center environments. Focal Point can enhance this function by offering virtual fabric memory partitions for storage and data as well as providing low latency storage access. Focal Point can also help distribute the processing load across multiple load balancing cards or systems, providing scalable solutions. Finally, Focal Point offers advanced frame forwarding and security features ideally suited to advanced load balancing systems. Pastry and Chord are structured peer-to-peer overlay network protocols which give huge potential for building self organizing applications to today's programmers. They cover a lot of needed services which are need to build a peer-to-peer application. The lack of security management now is a great issue to actual research and can surely be solved in trade off for some performance of these protocols. But the question is whether these protocols are also this top-performing in a mobile or wireless environment. Chord shows here some negative impact on high packet loss rates in MANETs. At least the Pastry implementation MS Pastry uses a retransmission strategy on occurrence of packet loss in the underlying network. So it seems to be more applicable to this case of use. Future research is about improvements of these protocols and applications build on top of Pastry and Chord. There seems to be more research in wired environments which is especially pushed through the Microsoft Corporation on Pastry, but until now only little effort was done on building applications on Pastry and Chord in wireless environments like WSNs or MANETs. Dynamic collection and analysis of program and operating system information in concurrent systems. The monitoring system is itself parallelized and distributed; it consists of resident monitors on each network node, which collects and analyzes information local to that node, and a logically centralized monitor, who presents a user interface and correlates and **stores** distributed information, as necessary. The system's novel attributes include 1) its multiplicity of information collection mechanisms: sensors, extended sensors, and probes, and 2) its use for dynamic or static adaptation of concurrent application programs. The utility of the system is demonstrated with a workload generator program and with the adaptation of a sample parallel (and distributed) program. **A** major contribution of this research is a demonstration that an entity-relationship (E-R) model may be used for 1) the description of concurrent software and distributed or parallel hardware, 2) the specification of program views and attributes for monitoring, and 3) the determination of distributed analysis and collection to be performed for the specified views. They also expect to have the opportunity to perform standard and ad-hoc reporting, and that the results will be immediately available. And, of course, these users demand that technical issues be transparent to them As can be seen, data warehouses require quite different capabilities from OLTP environments. In addition to B+-trees, one needs bitmap indexes. Instead of a general purpose optimizer, one needs to focus special attention on aggregate queries over snowflake schemas. Instead of normal views, one requires materialized views. Instead of fast transactional updates, one needs fast bulk load, etc. A longer overview of data warehousing practices can be found in the major relational vendors began with OLTP-oriented architectures, and have added warehouse-oriented features over time. In addition, there are a variety of smaller vendors offering DBMS solutions in this space. These include Teradata and Natasha, who offer shared nothing proprietary hardware on which their DBMSs run. Also, selling Database storage subsystems are a very mature technology, but a number of new considerations have emerged for database storage in recent years, which have the potential to change data management techniques in a number of ways. One key technological change is the emergence of flash memory as an economically viable random-access persistent storage technology.

Reference

1. OpenMP Forum: OpenMP Standard, http://www.openmp.org.
2. Etnus LLC.: TotalView, http://www.etnus.com/.
3. Inmon, W.H., et al. (1999). Data Warehouse Performance. New York: Wiley Computer Publishing. Stephen Morse and David Isaac (1998). Parallel Systems in the Data Warehouse. Upper Saddle River, NJ: Prentice- Hall, Inc.
4. Sun Microsystems Database Engineering Group (1998). "Data Warehousing Performance with SMP, Cluster, and MPP Architectures".
5. The Data Warehouse Information Center (one-stop shopping for links to other DW sites.): http://pwp.starnetinc.com/larryg/ Thuraisingham, Bhavani (1999). Data Mining: Technologies, Techniques, Tools, and Trends. New York: CRC Press.
6. Silicon Graphics/Cray Research, Sun, Unisys, White Cross Systems, Informix, Oracle, and Sybase.
7. J. Magee and J. Kramer, "Dynamic configuration for distributed realtime systems," in Proc. Int. Conf Parallel Processing, IEEE, ACM,
8. C .E. McDowell and D. P. Helmbold, "Debugging concurrent programs," ACM Comput. Surveys, vol. 21, no. 4, pp. 593-623, Dec. 1989.
9. K. Schwan, R. Ramnath, S. Sarkar, and S. Vasudevan, "Cool-Language constructs for constructing and tuning parallel programs," in Proc. Inr. Conf: Comput. Languages, Miami Beach, FL, IEEE, Oct. 1986, pp. 90-103.
10. ACM Trans. Computer Syst., vol. 5, no. 3, pp. 189-231, Aug. 1987. K. Schwan, B. Blake, W. Bo, and J. Gawkowski, "Global data and control in multicomputers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm," in Concurrency: Practice and Experience.
11. K. Schwan, R. Ramnath, S. Vasudevan, and D. Ogle, "A system for parallel programming," in Proc. 9th Int. Conf Software Eng., Monterey, CA, IEEE, pp. 270-282, ACM, Mar. 1987. Awarded best paper. -, "A language and system for parallel programming," IEEE Trans. Software Eng., vol. 14, no. 4, pp. 455-471, Apr. 1988.
12. D. C. Marinescu, J. E. Lumpp, T. L. Casavant, and H. J. Siegel, "Models for monitoring and debugging tools for parallel and distributed software," J. Parallel Distributed Comput., vol. 9, no. 2, pp. 171-184, June 1990.

13. D. M. Ogle, P. Gopinath, and K. Schwan, "Tool integraton in distributed programming and execution environments- Representing and using monitored information," in Proc. IEEE Workshop Experimental Distributed Syst., Huntsville, AL, IEEE, 1990, pp.
14. K. Marzullo and M. Wood , "Making real-time systems reactive," ACM Operat. Syst. Rev., vol. 25, no. 1, Jan. 1991.
15. B. Clifford Neuman. Scale in distributed systems. In T. Casavant and M. Singhal, editors, Readings in Distributed Computing Systems, pages 463–489. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994. http://clifford.neuman. name/papers/pdf/94--_scale-dist-sys-neuman-readings-dcs.pdf.
16. M. Steinbrunn, G. Moerkotte, and A. Kemper, "Heuristic and randomized optimization for the join ordering problem," VLDB Journal, vol. 6, pp. 191–208, 1997.
17. S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating mining with relational database systems: Alternatives and implications," in Proceedings of ACMSIGMOD International Conference on Management of Data, 1998.
18. M. A. Shah, S. Madden, M. J. Franklin, and J. M. Hellerstein, "Java support for data-intensive systems: Experiences building the telegraph dataflow system," ACM SIGMOD Record, vol. 30, pp. 103–114, 2001.
19. A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts. McGraw-Hill, Boston, MA, Fourth ed., 2001.
20. R. Sears and E. Brewer, "Statis: Flexible transactional storage," in Proceedings of Symposium on Operating Systems Design and Implementation (OSDI), 2006.
21. Transaction Processing Performance Council 2006. TPC Benchmark C Standard Specification Revision 5.7, http://www.tpc.org/tpcc/spec/tpcc current. pdf, April.