Scalable, Parallel, Scientific Databases

John L. Pfaltz †, Russell F. Haddleton‡, James C. French
Computer Science Department, University of Virginia

Abstract: *Large scientific applications which rely on highly parallel computational analysis require highly parallel data access. We describe an object-oriented, scientific database system that achieves nearly linear scale-up over large, million object data sets. Of primary importance are those features which seem central to the development of this, or any other parallel database system. These include techniques of object distribution, of multi-operator parallelism, and of indirect object referencing. It also appears to require a query server architecture instead of the more common page server configurations.*

1. Introduction

Scientific databases must, almost by definition, be very large.[1] But size alone does not necessarily distinguish between scientific and commercial databases. Instead, scientific databases seem to be separated from their commercial cousins by three fundamental characteristics. First, the observed entities are frequently more complex with significant interrelationships. Consequently, object-oriented technology often seems more appropriate. Second, scientific databases are seldom transaction oriented. Recorded data is to be preserved for posterity; it is not constantly changing to reflect the current status of a system. Retrieval of ''interesting'' data is far more important. Third, retrieval of data is often ''volumetric,'' that is, based on two or more conjunctive range searches. Figure 1-1, in which objects are retrieved from a 2 parameter vector space is representative.[2]

```
retrieval_set <- { x in observations |
                   p1 <= x.pressure    <= p2 and
                   t1 <  x.temperature < t2      }
```

Figure 1-1
Typical volumetric query

Spatial queries constitute one special case of volumetric retrieval. These alone are so important that special access methods, such as grid files [Hin85, NHS84] and R+ trees [SRF87] have been developed solely to efficiently execute access with respect to these volumetric patterns [Mat94].

Our characterization of scientific databases in terms of these three properties arises from our experience with very large sets of radio astronomy data [KGF94, KFG94], in which volumetric retrievals based on from 3 to 7 parameters are common; and with very large sets of global change data [EKP94] that are generated by simulation models. Here there are many interactions between constituent objects.

Some may feel that we have ignored other important properties, such as the temporal nature of certain kinds of scientific data, or of its frequent statistical structure. This may be true. But, few would deny that the three characteristics we have enumerated — complexity, lack of update transactions, and volumetric retrievals — are nearly ubiquitous.

This paper discusses a scalable, parallel, object-oriented database implementation called ADAMS [Pfa93, PfF93] that has been designed to handle volumetric queries in such very large scientific databases.

2. Parallel Operations

In a parallel database, the data items must be distributed over the n constituent nodes, or processors. In the relational model, it is customary to distribute the tuples with respect to the hash of some attribute value, usually a key value [BoD81, BAC90, ShW91]. Depending on the

---

† Supported in part by DOE Grant # DE-FG05-95ER25254.

‡ Current address: SAIC, McLean, VA

[1] If the data sets are small, it is often simpler to represent them using spread sheet technology or by in-memory data structures.

[2] Queries in ADAMS use the standard predicate calculus notation, $\{ x \in S \mid P(x) \}$, to denote sets of persistent objects.

actual attribute values and the hash function, the distribution may, or may not, be relatively uniform.

In the object-oriented model, all objects are uniquely identified by an *oid*. The *oid* is a bit string.[3] In our implementation we simply use the low order d bits to distribute objects over the $n = 2^d$ processors. More sophisticated hashing functions could be used; but to date we have found this to be quite adequate. Consequently, the object denoted by oid_i will be distributed to processor k where $k = oid_i \bmod n$. Distribution with respect to *oid* has several advantages. First, we can partition sets of objects, such as any S and T, into n constituent subsets, say $S = \{ S_1, S_2, \cdots S_n \}$ and $T = \{ T_1, T_2, \cdots T_n \}$ where $S_k = \{ oid_i \mid oid_i \in S \wedge oid_i \bmod n = k \}$. Here we implicitly assume that a set is always a list, tree, or other collection of *oid*'s (or pointers, or object references) and never the objects themselves. With this distribution, the standard set operators, *union*, *intersection* and *complement* (or difference), have nearly trivial parallel implementations. For example, it is not hard to see that $S \cup T \equiv \bigcup_{k=1}^{n} S_k \cup T_k$. These set operations can be performed on each processor in a completely data parallel manner, with no cross talk. And given a relatively uniform distribution of S and T across their subsets S_k and T_k, this yields an n-fold speed up. With large sets, say $| S | > 5{,}000$, all our experience confirms these expectations. With small sets, say $| S | < 500$, one may see skewed distributions. But, when data sets are small, parallel performance is less critical.

Our first characterization of scientific data was that individual data items tend to have more complex relationships with other items. This is an important reason for selecting the object-oriented model. Such relationships are commonly represented by a pointer from (or in) obj_i to obj_j. These have been called ''implicit joins'' [Ber94] in analogy to the lossless joins of the relational model. In the ADAMS query of Figure 2-1,

```
result <- { x in observations |
                 p1 <= x.pressure   <= p2  and
                 t1 <  x.temperature < t2  or
                 x.instrument.elevation > 1000 }
```

Figure 2-1
Volumetric query with implicit join

two classes of objects are referenced. There are *observation* objects and *sensor* objects. The attributes *pressure* and *temperature* are properties of *observation* objects; *elevation* is an attribute of a *sensor* object. With each *observation* object is an object-valued attribute *instrument* which denotes (points to or joins) the sensor instrument on which the observation was recorded. This query retrieves observations satisfying the *pressure* and *temperature* constraints, along with all observations made at an *elevation* of more than 1,000 meters.

We will discuss the implicit join of the last disjunct in some detail because they are the *bete noire* of parallel database systems. Any database system which relates items, tuples or objects belonging to different schema or classes must implement the equivalent of an implicit join operator. But, such inter-object references can not be implemented in a completely data parallel manner. If the objects have been distributed to different processors, there must be inter-processor communication which takes time and which can saturate a network with many messages. This is the major cost of hash-join operators in large distributed relational databases.

Let $oid_j = f(oid_i)$ mean that the object oid_j is functionally related to the object oid_i, that is, that oid_j is a foreign key of the implicit join. The two objects oid_i and oid_j need not have been distributed to the same processor P_k. In fact, if objects are distributed uniformly, with probability $1 - 1/n$ they will be on different processors.

To execute the implicit join term, $x.instrument.elevation > 1000$ of Figure 2-1, each processor P_k retrieves the set Y_k of sensor objects having elevation > 1,000 meters, that is the set $Y_k = \{ oid \mid oid.elevation > 1000 \wedge oid \bmod n = k \}$. Then for each oid_j in this set Y_k, the process must retrieve the set $\{ oid_i \mid f(oid_i) = oid_j \}$, where in this case $f = instrument$.[4] While all of the sensor objects oid_j were distributed to P_k, very few of the referencing oid_i may be. These *oid*'s must be sent to their home processors before any further operations can be executed. But, we need transmit only the *oid*, not the entire object. Consequently, we can pack many *oid*'s into a single data message. In our current implementation we pack 500 *oid*'s in each full data message. Although a 32,000 byte message is long, communication has been fast and we have observed little network contention.

The effectiveness of these techniques is illustrated in Table 1, which details the behavior of the implicit join query of Figure 2-1 in a 4 processor configuration. We observe that, although a total of 76,430 objects were retrieved, only 227 data messages were exchanged between processors, with a maximum of 65 sent by P_2.

[3] In ADAMS, we use 64 bit identifiers. Some other object-oriented systems use 128 bit, or longer, *oid*s.

[4] This construction assumes that on P_k we index the set of *oid*'s that *reference* a particular oid_j assigned to P_k instead of indexing the set of all *oid*'s that oid_j references. See [Had97] for an explanation of why the latter is far less efficient in many to one joins in the face of data skew.

	P_1	P_2	P_3	P_4	Sum
| p1 < x.press < p2 |	7,466	7,499	7,503	7,439	29,907
| t1 < x.temp < t2|	47,631	47,657	47,371	47,591	190,250
| y.elev > 1000 |	61	71	70	49	251
| result |	19,179	19,079	19,120	19,052	76,430
$read_{random}$	1,872	1,868	1,800	1,673	7,213
$read_{sequential}$	643	373	268	265	1,549
$read_{cache}$	305,583	305,530	304,663	305,374	1,221,152
$write_{random}$	25	26	25	26	102
$write_{sequential}$	1,173	1,167	1.161	1,173	4,674
$write_{cache}$	137,302	136,990	136,749	136,837	547,878
$send_{data\ message}$	54	65	63	45	227
$receive_{data\ message}$	57	54	55	61	227

Table 1
Operation counts for the query of Figure 2-1.
$n = 4$, Total database size, 1M observation objects

Examination of the various intermediate sets is worthwhile. The distribution of the sets representing the retrieval terms $p1 < x.pressure < p2$ and $t1 < x.temperature < t2$ is quite uniform, with a maximum relative deviation from the mean of 0.005. However, the distribution of the 251 instruments that were located above 1,000 meters shows considerable skew. This skew is mirrored in the counts of inter-processor data messages that were sent because the implicit join is based on these *sensor* instruments. Object distribution with respect to *oid* seems to work well with large sets of objects, less well with smaller sets.[5] This table also lists the observed number of read/write operations. We will use these values in Section 4 when we discuss the importance of cache storage and cache operators.

There is an extensive literature describing the parallel execution of database operators — especially parallel join operators, *c.f.* [BaF87, DKT90, MuR93, ScD90, ShW91]. While all improve performance; most are limited by their granularity. It is only the specific operator that is executed in parallel, not the entire statement or query. In a moderately complex query each operation must be complete before the next can begin. For example, parallel hash join operators first distribute all tuples with respect to the hash value of the join attribute before initiating the join. Thus each statement is implemented as the sequential execution of a series of parallel operators. Such repeated synchronization limits parallel performance. Ideally one wants to execute entire statements, or even sequences of statements, in parallel without synchronization [GPF90]. ADAMS employs streams and lazy evaluation to increase the granularity of its parallelism.

All sets of objects are ordered in ascending *oid* order. Those sets of objects in persistent, disk storage are represented as trees to facilitate retrieval. Sets of objects that have been retrieved from persistent storage into RAM are represented as streams, that is a linear sequence in ascending *oid* order. Because sets are expected to be large, streams are segmented. Each segment consists of σ objects.[6] Streams are generated and consumed one segment at a time using lazy evaluation. When a consuming process requires another stream segment it is delivered by the producing process. Consider the ADAMS iteration of Figure 2-2 which implements a retrieval with respect to a derived attribute $t_p_ratio > 3.1$, where $t_p_ratio = temperature / pressure$ over only those observations which were collected *today* or *yesterday*. (This assumes the application collects and updates these observation sets on a daily basis.) The loop iteration consumes objects one at a time.

```
for_each  obj  in  today union yesterday do
      if (obj.temperature/obj.pressure > '3.1')
            {
            insert obj into keeper
            }
```

Figure 2-2
Iteration over a set expression

When the last object in the segment has been consumed, it signals the *union* operator. The *union* operator, in turn, consumes segments delivered by the lowest level set access operators to the persistent sets *today* and *yesterday*. If the statement is executed in a distributed

[5] We should note that this query was chosen precisely because it illustrates considerable skew. Most queries are far more uniform.

[6] ADAMS segments consist of 1,000 objects; but σ can be implementation dependent.

environment, each processor can execute asynchronously with no inter-processor messages. Because they are asynchronous, distributional variance with respect to one operator may be cancelled later.

An alternate version of this same code is presented in Figure 2-3.

```
temp  <-  today union yesterday
for_each  obj  in  temp do
      if (obj.temperature/obj.pressure > '3.1')
            {
            insert obj into keeper
            }
```

Figure 2-3
Equivalent code to Figure 2-2

Note that it is not necessary for every processor P_k to completely produce its subset $temp_k$ of the object set *temp* and then synchronize with the other processors before beginning the iteration. As soon as the first segment of $temp_k$ has been generated, P_k can begin the iteration. This is a simple example of multi-statement parallel threads.

Lazy evaluation based on segments does have its limitations. While it would be easy to create an application in which all of today's observation objects were collected in a set *today*, and then moved to *yesterday* before collecting the next day's set, it seems a bit improbable. We would be much more likely to simply date the observation objects and retrieve on the date as in the code of Figure 2-4.

```
for_each  obj  in  { x in observations |
                       x.date = '12/15/97' or
                       x.date = '12/16/97'    }
      if (obj.temperature/obj.pressure > '3.1')
            {
            insert obj into keeper
            }
```

Figure 2-4
Equivalent code to Figure 2-2

Here we are looping over a retrieval set consisting of observation objects collected on two separate days. Because sets of objects, whether represented as streams in memory or trees on disk, are always maintained in ascending *oid* order and because simple selection by attribute value need not retrieve them in this order, the objects denoted by a retrieval term such as

```
x.date = '12/15/97'
```

must be completely retrieved and then sorted before any segment can be passed on to a subsequent operation.[7] Nevertheless, we estimate that the principle of lazy evaluation, which permits processors to execute substantial code threads asynchronously at their own speed, has improved parallel performance between 10 to 15%, and is instrumental in obtaining scalable parallelism.

[7] Applying a novel cache management algorithm during attribute selection and sorting improved performance by 35%. See [Had97] for details.

Distribution of objects by *oid*, reduction of inter-process message traffic, and lazy evaluation are three principles that can contribute to efficient parallel access to scientific data in any system.

3. Parallel Performance

''Speed up'' denotes the improvement in overall execution time obtained by distributing a specific task over *n* processors. It is the ratio

$$\frac{\mathbf{time}_1\,(\,t_size\,)}{\mathbf{time}_n\,(\,t_size\,)}\,.$$

Here *t_size* measures the ''size'' of the task and $\mathbf{time}_k$ denotes the execution time on *k* processors, $1 \le k \le n$. We say linear speed up occurs when this ratio = *n*. In theory, linear speed up represents the ideal, and is unattainable[8]; but in practice one may see superlinear speed up. The observed speed up of our running example is shown in Figure 3-1.

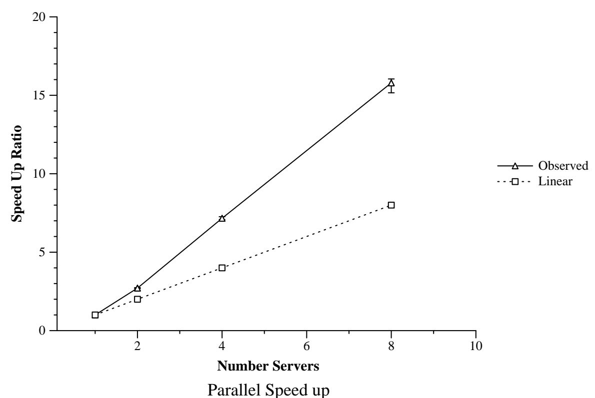

Figure 3-1
Speed up observed over a 1M data set

The dashed line indicates expected performance assuming a linear speed up. Readily, we have an apparent super-linearity which occurs because we are using many more system resources, *e.g.* *n* times the cache storage, to resolve the query.

In practice, speed up is of central importance. As theory, it is flawed. The difficulty lies in measuring the ''size'' of a task. The size of database query task is not simply determined by the number of objects considered and the number retrieved. One must also account for the resources available to execute the task, especially cache

[8] *c.f.* Amhdal's Law [AlG89]

storage in which indexes and temporary results can reside. For example, a fixed size database that can reside completely in RAM on one system [GaS93] will have very different performance characteristics on a smaller hardware configuration.

''Scale up'' denotes the increase in task size that can be accommodated in a fixed execution time by increasing the number of processors, that is the ratio

$$\frac{\mathbf{time}_1\,(\,t_size\,)}{\mathbf{time}_n\,(\,n\cdot t_size\,)},$$

where one has linear scale up if this ratio is one. Figure 3-2 illustrates the scale up of our running query. It is nearly, but not quite, linear. Even though it is a better measure of parallel performance [BCV93, DeG92], scale up is seldom measured. It is too time consuming. To create Figure 3-2, we created a random database with 125,000 observation objects and ran 10 independent queries[9] on a single processor configuration. Then we created a new random database of 250,000 objects and timed 10 more queries on 2 processors. The procedure was repeated with 500,000 and 1,000,000 object databases on 4 and 8 processor configurations respectively. Thus the ratio of task size to system resources is held constant. Repeated data creation is tedious, but essential to support our claim of a scalable parallel implementation.

4. Underlying Architecture

The ease with which the techniques of the preceding section are implemented can be affected by one's choice of architecture. We assume that these databases are shared, that there may be multiple users at any time, and that applications accessing the database will require considerable subsequent processing. This essentially dictates a client/server architecture in which each client is assumed to have an independent processor, either a PC or more powerful workstation, and that the database will be directly accessed by other processors or file servers. First, within a the client/server model we elected to use a query server architecture rather than either a page server or object server configuration. Briefly, given an object *oid*, a page server returns the entire page on which the object resides, whereas an object server returns just the object itself to the client. It is presumed that the client application can determine which objects are desired. In contrast, a query server accepts sequences of instructions in addition to argument *oid*'s with the expectation that the server will execute the instruction sequence to determine what objects, object *oid*'s, or object attribute values, should be returned to the client process.

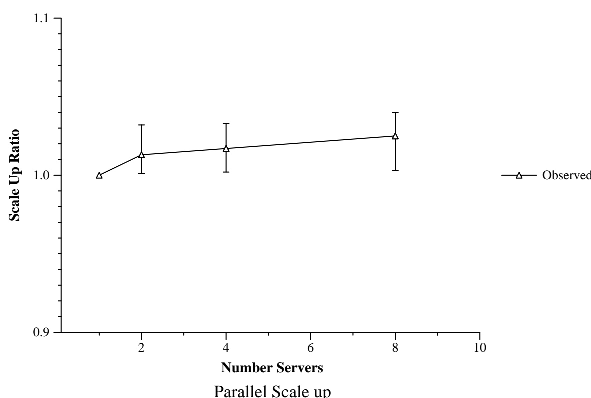

Figure 3-2
Parallel query scale up

[9] By independent we mean that after executing each query, the system was brought down, all data caches were flushed, and the system re-initialized.

We are convinced this was a wise choice. In [HaP97], we enumerate several reasons why query server architectures are superior for the kinds of scientific applications of interest to this conference. Given the operations counts of Table 1, perhaps the two most compelling reasons are that:

(1) only 76,430 extracted objects need be transmitted from the *n* server processes to the client processor. While this is only about 7.6% of the entire million object database, if one assumes a reasonably uniform distribution of the these objects then one can show that page servers would have to return nearly all of the 12,000 pages of the database, *just to accommodate the result*.

(2) the process of retrieval and its I/O kept close to the data. Access to index tree structures constitutes a significant part of the disk I/O, and searching them is then done entirely in cache. Hence, we account for some of the cache reads and writes of Table 1. In addition, all sets are kept in ascending *oid* order. It is this sorting activity that actually accounts for the bulk of the cache read and write operations. Readily, we have two orders of magnitude more cache operations in our system than disk I/O, but this seems quite acceptable given the corresponding execution times in Table 2.

time ($read_{random}$)	8,000 µ*secs.*
time ($read_{sequential}$)	1,800 µ*secs.*
time ($read_{cache}$)	94 µ*secs.*
time ($send_{data\ message}$)	131,960 µ*secs.*

Table 2
Selected operation times, SPARC2 and Seagate 1.2 GB disk

Processing large data sets "close to" the data itself seems valuable.

Further, sets and inter-processor messages are all expressed in terms of *oid*'s. These serve as surrogates for

the objects themselves. Because we do not have literal sets of objects, in the sense that relations are physically sets of tuples, the functional data model becomes quite natural. An attribute such as *pressure* can be regarded abstractly as a function whose argument is the *oid*. Now these attributes can be implemented either as functions whose value is found ''on the fly'', or as fields within a storage structure. ADAMS has elected the former. This yields expressive flexibility [PfF90]; but with some cost in efficiency.

5. Conclusions

If the defining characteristic of a large class of scientific data sets is their sheer size, then practical use of this data will require parallel processing and parallel access. We have concentrated on the latter because we know of no other parallel, object-oriented database systems.[10] The ADAMS database system has been extended to a parallel implementation in a seamless fashion. The same source code can be executed over a single processor, or multi-processor, configuration without alteration. It needs only recompilation and relinking. Moreover, this parallel extension appears to scale in a nearly linear fashion.

We believe a number of factors contributed to this parallel performance. They are:

(1) **distribution of objects by** *oid*. This seems to assure a relatively uniform distribution of objects and hence similar execution times for each parallel thread. While we can envision special case situations in which distribution by an attribute property could be used to efficiently ''cluster'' certain kinds of data, distribution by *oid* seems superior in a general purpose system.

(2) **lazy evaluation.** If all the processors must synchronize after each operation, or after each statement, then overall performance is repeatedly reduced to that of the slowest processor at each step. There is always variance in any operation. Lazy evaluation encourages the asynchronous execution of many operators with a subsequent decrease in overall variance.

(3) **functional model.** In retrospect, this is not essential to obtain parallel access capability. But, the flexibility offered by this alternative view and representation of data provided a kind of freedom to develop this system.[11] The same flexibility has other benefits in a database system, and may offer other kinds of parallelism [PFG92] that were not exploited in this version of ADAMS.

(4) **extensive use of** ***oid*****'s as surrogates.** Some object-oriented systems [CDR86, KGB90] populate some sets with actual objects and others with *oid* pointers. Judicious implementation of this form can increase efficiency. But, the heterogeneous representation also makes parallel implementation more difficult and inter-processor communication much more expensive.

If the study of scientific databases is to emerge as the kind exciting discipline we think it can, it must provide scientists with the tools to conduct their research effectively. Statistical tools and temporal logic tools are clearly essential. So too are tools that permit a scientist to sift through immense amounts of data, whether for data mining or for graphical visualization or for subsequent statistical analysis. These tools to cope with vast amounts of data must include parallel access and delivery. The ADAMS prototype demonstrates that scalable object-oriented parallelism is possible and suggests a number of techniques that can be incorporated in such systems.

[10] Shore [CDe94] has considered implementing a form of parallel access using ParSets [DNS94], but has never released this feature. Versant [VER97] is a commercial object-oriented, query server system, but it has no parallel version either.

[11] Efficient, scalable relational systems using our techniques are quite feasible; but none exist. We think the barriers are primarily psychological.

6. References

[AlG89] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[BaF87] C. K. Baru and O. Frieder, Implementing Relational Database Operations in a Cube-Connected Multicomputer System, *IEEE Conf. on Data Engineering CH2407-5*(Feb. 1987), 36-43.

[BCV93] B. Bergsten, M. Couprie and P. Valduriez, Overview of Parallel Architectures for Databases, *The Computer Journal 36*,8 (1993).

[Ber94] E. Bertino, A Survey of Indexing Techniques for Object-Oriented Database Management Systems, in *Query Processing for Advanced Database Systems*, J. C. Freytag, D. Maier and G. Vossen (editors), 1994, Morgan Kaufmann.

[BoD81] H. Boral and D. J. DeWitt, Processor Allocation Strategies for Multiprocessor Database Machines, *ACM Trans. Database Systems 6*,2 (June 1981), 227-254.

[BAC90] H. Boral, W. Alexander, L. Clay, G. Copeland and al., Prototyping Bubba, A Highly Parallel Database System, *IEEE Trans. Knowledge and Data Engineering 2*,1 (Mar. 1990), 4-24.

[CDR86] M. J. Carey, D. J. DeWitt, J. E. Richardson and E. J. Shekita, Object and File Management in the EXODUS Extensible Database System, *Proc. 12th VLDB Conf.*, Kyoto, Japan, Aug. 1986, 91-100.

[CDe94] M. J. Carey, D. J. DeWitt and et.al., Shoring Up Persistent Applications, *SIGMOD Record 23*,2 (June 1994), 383-394.

[DeG92] D. DeWitt and J. Gray, Parallel Database Systems: The Future of High Performance Database Systems, *Comm. ACM 35*,6 (June 1992), 114-120.

[DNS94] D. J. DeWitt, J. F. Naughton, J. C. Shafer and S. Venkataraman, Parsets for Parallelizing OODBMS Traversals: Implementation and Performance, *Proc. of the Third Inter. Conf. on Parallel and Distributed Information Systems*, Austin, TX, Sep. 1994.

[DKT90] S. M. Deen, D. N. P. Kannangara and M. C. Taylor, Multi-join on Parallel Processors, *Proc. 2nd Intern'l Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990, 92-102.

[EKP94] W. R. Emanuel, A. W. King and W. M. Post, A Dynamic Model of Terrestrial Carbon Cycling, in *The Global Carbon Cycle*, M. Heimann (editor), Springer-Verlag, Berlin, 1994.

[GaS93] H. Garcia-Molina and K. Salem, Main Memory Database Systems: An Overview, *IEEE Trans. on Knowledge and Data Eng. 4*,6 (Dec. 1993), 509-516.

[GPF90] A. S. Grimshaw, J. L. Pfaltz, J. C. French and S. H. Son, Exploiting Coarse Grained Parallelism in Database Applications, *PARBASE-90 International Conf. on Databases, Parallel Architectures and their Applications*, Miami Beach, FL, Mar. 1990, 510-512.

[Had97] R. F. Haddleton, *An Implementation of a Parallel Object Oriented Database System*, Ph.D. dissertation, Dept. of Computer Science, Univ. of Virginia, Charlottesville, VA, Aug. 1997.

[HaP97] R. F. Haddleton and J. L. Pfaltz, Client/Server Architecture in the ADAMS Parallel, Object-Oriented Database System, *Proc. 1st Intern'l Conf. in Scientific Computing in Object-Oriented Parallel Environments*, Marina del Rey, CA, Dec. 1997, 257-266.

[Hin85] K. Hinrichs, Implementation of the Grid File: Design Concepts and Experience, *BIT 25*(1985), 569-592.

[KGF94] J. F. Karpovich, A. S. Grimshaw and J. C. French, Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O, *Proc. OOPSLA '94: Object-Oriented Programming Systems and Languages*, 1994.

[KFG94] J. F. Karpovich, J. C. French and A. S. Grimshaw, High Performance Access to Radio Astronomy Data: A Case Study, *Proc. 7th Inter. Conference on Scientific and Statistical Database Management*, Charlottesville, VA, Sep. 1994, 240-249.

[KGB90] W. Kim, J. F. Garza, N. Ballou and D. Woelk, Architecture of the ORION Next-Generation Database System, *IEEE Trans. on Knowledge and Data Engineering 2*,1 (Mar. 1990), 109-124.

[Mat94] J. Matousek, Geometric Range Searching, *Computing Surveys 26*,4 (Dec. 1994), 421-461.

[MuR93] M. C. Murphy and D. Rotem, Multiprocessor Join Scheduling, *IEEE Trans. on Knowledge and Data Engineering 5*,2 (Apr. 1993), 322-338.

[NHS84] J. Nievergelt, H. Hinterberger and K. C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure, *ACM Trans. Database Systems 9*,1 (Mar. 1984), 38-71.

[PfF90] J. L. Pfaltz and J. C. French, Implementing Subscripted Identifiers in Scientific Databases, in *Statistical and Scientific Database Management*, Z. Michalewicz (editor), Springer-Verlag, Berlin-Heidelberg-New York, Apr. 1990, 80-91.

[PFG92] J. L. Pfaltz, J. C. French, A. S. Grimshaw and R. D. McElrath, Functional Data Representation in Scientific Information Systems, *Intern'l Space Year Conf. on Earth and Space Science Information Systems (ESSIS)*, Pasadena, CA, Feb. 1992, 788-799. AIP Conf. Proc. #283.

[Pfa93] J. L. Pfaltz, The ADAMS Language: A Tutorial and Reference Manual, IPC TR-93-003, Institute for Parallel Computation, Univ. of Virginia, Apr. 1993.

[PfF93] J. L. Pfaltz and J. C. French, Scientific Database Management with ADAMS, *Data Engineering 16*,1 (Mar. 1993), 14-18.

[ScD90] D. A. Schneider and D. J. DeWitt, A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment, *Proc. 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990, 110-121.

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos, The R+-Tree: A Dynamic Index for Multi-Dimensional Objects, *Proc. 13th Conf. on VLDB*, Brighton, England, 1987, 507-518.

[ShW91] D. Shasha and T. Wang, Optimizing Equijoin Queries in Distributed Databases where Relations are Hash Partitioned, *ACM Trans. Database Systems 16*,2 (June 1991), 279-308.

[VER97] *Versant: OBDMS Release 4*, Versant Object Technology, 1997.