The VLDB Journal (1996) 5: 3–18

The VLDB Journal

Parallelizing OODBMS traversals: a performance evaluation

David J. DeWitt, Jeffrey F. Naughton, John C. Shafer, Shivakumar Venkataraman

Computer Sciences Department, University of Wisconsin Madison, 1210 West Dayton St, Madison WI, 53706, USA

Edited by Henry F. Korth and Amit Sheth. Received November 1994 / Accepted March 20, 1995

Abstract. In this paper we describe the design and implementation of *ParSets*, a means of exploiting parallelism in the SHORE OODBMS. We used ParSets to parallelize the graph traversal portion of the OO7 OODBMS benchmark, and present speedup and scaleup results from parallel SHORE running these traversals on a cluster of commodity workstations connected by a standard ethernet. For some OO7 traversals, SHORE achieved excellent speedup and scaleup; for other OO7 traversals, only marginal speedup and scaleup occurred. The characteristics of these traversals shed light on when the ParSet approach to parallelism can and cannot be applied to speed up an application.

Key words: ParSets – Parallelism – SHORE – Object-oriented database management systems

1 Introduction

The commercial success of parallel relational database systems (RDBMS) demonstrates convincingly that for RDBMS, parallelism is a highly effective tool for providing high performance (DeWitt and Gray 1992). However, it is much less clear whether parallelism can be effectively applied in object-oriented database systems (OODBMS). This is primarily because of the difference in workloads between RDBMS and OODBMS: while RDBMS typically execute queries specified in a set-oriented declarative language (SQL), OODBMS typically execute arbitrary C++ code.

A main goal of the SHORE Persistent Object Store project (Carey et al. 1994) is to exploit parallelism to improve the performance of OODBMS applications. Our goal is not, however, to solve the problem of automatically parallelizing arbitrary C++ code. Rather, our goal is to provide system primitives that make it easy for a programmer to consciously and explicitly parallelize his or her OODBMS application. This paper describes the design and implementation of one primitive we have provided the *ParSet* facility and discusses how ParSets can be used to parallelize the OO7 OODBMS benchmark traversals (Carey et al. 1993), and finally presents performance results from an implementation on a cluster of Sun workstations.

The ParSet facility is a variation of an idea that has appeared in many places before: essentially, it allows a program to invoke a method on every object in a set in parallel. [One place this idea appeared in a database context was the "filter" operation in the Bubba project at MCC (Bancilhon et al. 1987). Currently a related facility is being implemented at Kendall Square Research on top of the Matisse OODBMS (M.F. Kilian, personal communication 1994)]. Since this not a new idea, this is not the contribution of this paper; rather, the contributions of this paper are (1) a description of how the ParSet facility is actually implemented within SHORE, (2) a description of how the ParSet facility can be used to parallelize the traversals of the OO7 benchmark, and (3) performance results from an implementation that indicate how successful we were at exploiting parallelism to speed up and scale up these traversals.

Speeding up a benchmark in and of itself is perhaps of interest only to benchmarking fanatics, and the point of this work is not that OO7 can be sped up. Rather, the point is that OO7 allows us to explore a range of application characteristics and to see how various application characteristics impact on parallel performance. The OO7 traversals we tested range from sparse traversals that touch only a small percentage of the database, to dense traversals that touch most of the database, to dense traversals that touch most of the database and update the visited objects. We also tested performance on a number of database sizes.

Like any parallel application, we found that the performance we saw depended upon the ratio of the size of the sequential portions to the parallel portions of the traversals (smaller is better), and upon the size of the chunks of work that were executed in parallel (larger is better). While one traversal exhibited virtually no improvement due to parallelism, it was encouraging that for other traversals the ParSet declustering facilities coupled with the ParSet conditional set apply were enough to generate good speedup and scaleup on up to 16 commodity workstations connected by a standard ethernet. Furthermore, the ParSet facility was a natural way to parallelize these traversals; the programmer effort to

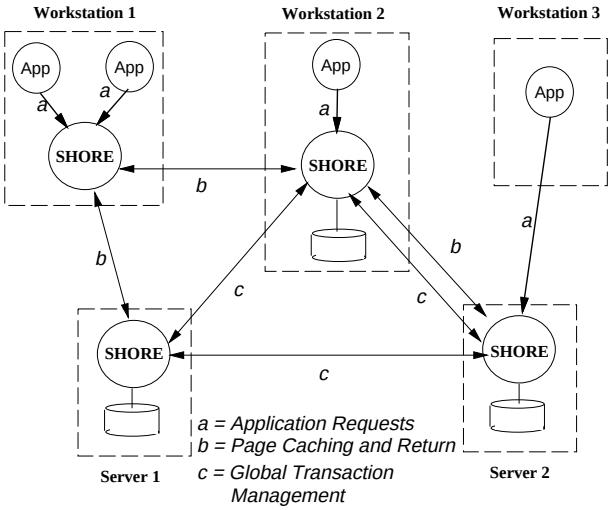

Fig. 1. SHORE process structure

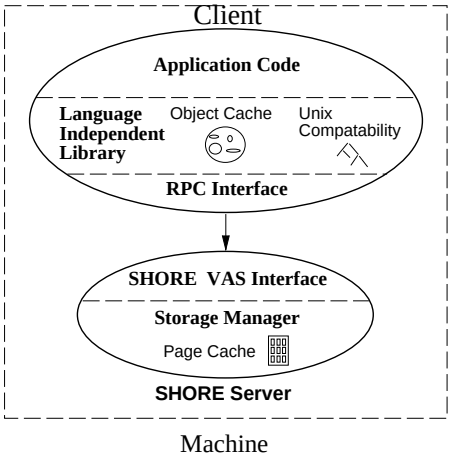

Fig. 2. Application/primary server interaction

convert from a sequential to a parallel implementation was minimal, roughly 3 h of coding.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the SHORE system sufficient to enable a reader not familiar with SHORE to understand the ParSet implementation. Section 3 describes the ParSet facility and how it is implemented in SHORE. Section 4 covers the OO7 benchmark database and traversals and how we used ParSets to parallelize these traversals. Section 5 presents the results of performance tests on the implementation of the ParSets, and Sect. 6 is the conclusion.

2 SHORE overview

SHORE (Scalable Heterogeneous Object REpository) is a new persistent object system under development at the University of Wisconsin that represents a merger of object-oriented database (OODB) and file system technologies. In this section we present an overview of the SHORE system in order to make this paper self-contained. This section is a greatly abridged version of material in Carey et al. (1994).

A primary goal for SHORE is to provide a robust, high-performance, persistent object system that is flexible enough to be employed in a wide range of applications and computing environments. To meet this requirement, SHORE's implementation is based on a novel *peer-to-peer* process structure. Figure 1 shows the overall process structure of SHORE in a distributed, workstation-server environment.

SHORE executes as a group of communicating processes called *SHORE servers*. SHORE servers consist exclusively of *trusted* code. In contrast to the trusted code, applications are considered to be untrusted, and thus, they execute as separate processes (labeled App in Fig. 1). Applications manipulate SHORE objects, while SHORE servers deal primarily with fixed-length pages. Pages, in turn, are allocated from disk volumes, each of which is managed by a single server. Individual pages can be dynamically replicated through the use of a caching mechanism. Caching allows data items to be replicated for performance reasons while ensuring that the responsibility for the integrity of any given data item resides with a single server.

SHORE servers perform two basic functions: (1) they manage persistent object storage, and (2) they provide applications with access to persistent objects that are managed either locally or by other remote SHORE servers. Each application process communicates with a single SHORE server, known as its *primary server*, through a SHORE value added server (VAS) interface. SHORE servers that manage persistent data items (shown with attached disk volumes in Fig. 1) are said to be the *owners* of those data items. Servers are responsible for providing concurrency control and recovery guarantees for the data that they own. While all SHORE servers are capable of acting both as primary servers and as owners of data, a given SHORE server will not necessarily perform both tasks.

A key concept behind the SHORE architecture is that an application can be given access to the entire distributed persistent object space through direct interaction only with its primary server. This is accomplished through the use of a *caching-based* architecture. The interaction between an application and its primary server is shown in Fig. 2. Applications are linked with a library of interface routines for communication with SHORE. All communication between an application and SHORE is initiated by application requests.

Applications request objects from their primary servers. The primary server obtains the physical data page (or pages) on which the object is stored and then transfers the object to the application. If the requested object resides on a disk volume that is managed by the primary server, then the request can be satisfied locally. If, however, the object resides on a page that is owned by a different server, then the primary server must request a copy of the page from the owner. Upon receiving a page, the primary server places the page in its local page cache and returns a copy of the requested object to the application. The application then places the object in its object cache. Before an application is allowed to read or update an object, however, the proper locks (read or write) must be obtained on behalf of the application. These locks are obtained by the primary server through interaction with the owner server (if the two are different).

Each application process has a private object cache that is presided over by an object cache manager. The primary goal of the object cache manager is to make navigation of persistent data structures convenient and efficient. To this end, it attempts to keep the application's working set of objects cached in the application's memory, and it maintains an efficient mapping between object pointers and their target objects (whether those objects are in the object cache or not). This mapping is enhanced by a swizzling technique where object references are replaced by pointers into an object descriptor table. The level of indirection provided by this table provides for flexible cache management, e.g., replacing objects in the cache prior to commit, yet it still enables the rapid navigation of cached object paths.

The dual nature of the SHORE server process provides a great deal of flexibility in the structuring of SHORE systems. When acting as an owner, the SHORE server performs the role of the server in a traditional data-shipping, client-server DBMS. When acting as a primary server, a SHORE server plays the role of the client of a data-shipping system. This symmetry of nodes in a cluster of workstations running SHORE servers, coupled with the global OID space maintained by the servers, facilitates building parallel applications.

3 ParSets and SHORE

In this section we define ParSets, discuss how they are implemented in SHORE, and demonstrate their use with a simple programming example. A ParSet is simply a set of objects of the same type or an appropriate subtype. The SHORE data language (SDL), which is our interpretation of the ODMG standard data definition language ODL (Cattell 1993), is used as the type language for ParSet objects. These sets of objects may be declustered over one or more SHORE servers; they are manipulated through several operations supported in the ParSet library. For example, one may have a ParSet of employee objects declustered over eight SHORE servers; issuing an Apply operation invokes a specified method on every employee object in the ParSet, in parallel.

Data parallelism is achieved by processing the fragments of the ParSet in parallel. We employ the "master-slave" architecture to achieve this. Initially, when the program begins executing, there is only the "master" thread. When the master first executes a ParSet call, "slave" processes are created and they execute the ParSet operation on the nodes on which the objects of the ParSet objects reside. Results returned by the slaves are returned to the user. An example of a ParSet operation is Apply, which invokes a specified method on every object in the ParSet. Since there is only one slave per ParSet fragment, the degree of parallelism is equal to the number of nodes over which the ParSet is declustered.

Note that the programmer does not need to write any slave code; this is supplied by the ParSet library. The programmer instead writes the code executed by the master, together with methods that will be invoked in parallel on ParSet objects. Communication between master and slaves, together with ParSet manipulation, is transparently handled by the ParSet library.

This simple description of the ParSet facility is sufficient for a basic understanding of the rest of this paper; readers that are not interested in the details of ParSets and how they are implemented can safely skip to Sect. 4.

3.1 What is a ParSet?

The idea of a ParSet was first proposed by Kilian (1992) under the name Parallel Set, as a way of adopting the data parallel approach to C++, although the ideas used in ParSets had appeared earlier in other data-parallel languages. SHORE ParSets differ from Kilian's ParSets in a number of ways, as will be discussed below. As envisioned by Kilian, ParSets support five basic operations: Add, Remove, Apply, Select and Reduce. *Add* adds an object to a ParSet. *Remove* removes an object from the ParSet. *Apply* invokes a function on every member of a ParSet. *Select* collects the OIDs of all ParSet objects that satisfy a specified predicate. *Reduce* calculates a single value from all objects in the set. Computing a scalar aggregate such as `max` or `sum` is an example of a reduce operation.

3.2 Primary and secondary ParSets

SHORE provides two forms of ParSets: primary and secondary. (This is specific to SHORE ParSets; Kilian had no notion of primary or secondary ParSets.) We use the terms "primary" and "secondary" by analogy to their use with indexes. Primary ParSets have a physical implication in that Primary ParSets are used for declustering. Secondary ParSets are just logical collections of objects; they can denote a set of objects over which an apply operation is to be executed; they do not imply anything about where the objects actually reside.

The declustering strategy for primary ParSets must be chosen when the ParSet is created. When objects are later added to the ParSet, a decluster method is invoked to determine the node on which objects have to be placed. SHORE ParSets provides standard declustering methods such as hash, range, random, and etc, which the user can choose to override. In addition, an "unspecified" declustering strategy is supported, as there are certain cases when no mapping exists from a value of an object to a processor number. For example, if objects are stored on the nodes where they were created there need not be any logical mapping from object values to nodes.

Secondary ParSets are nothing more than collections of OIDs of existing SHORE objects. Like a primary ParSet, a secondary ParSet can be declustered over a set of SHORE servers; however, no automatic declustering is supported. Generally the objects referenced in a secondary ParSet will themselves reside in some primary ParSet but this is not required. In addition, the referenced objects need not be "local", in that they may physically reside on a different SHORE server. These placement decisions are controlled by the application programmer.

The notion of primary and secondary ParSets was introduced to allow objects to dynamically change ParSet membership so that objects reside in more than one ParSet at the

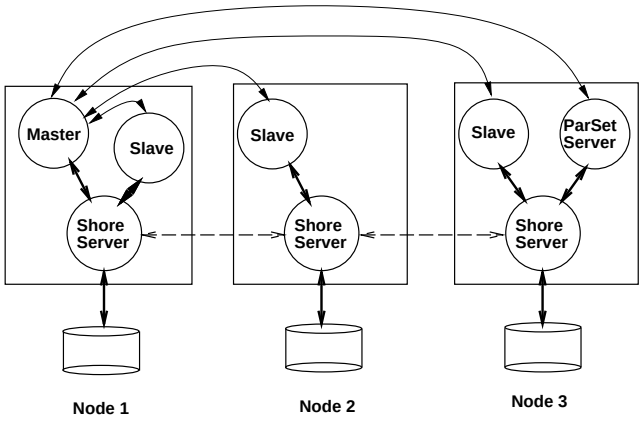

Fig. 3. SHORE parset architecture

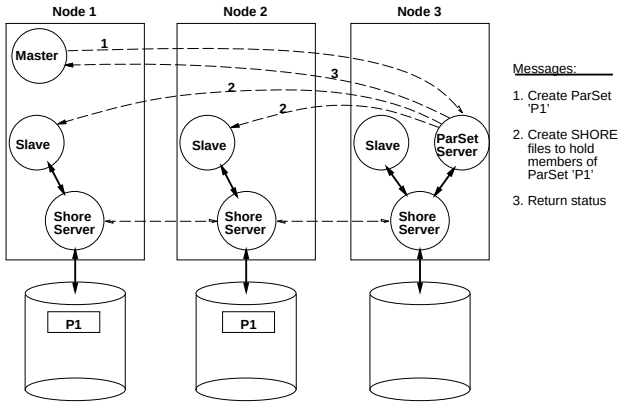

Fig. 4. Process communication on Create

same time. ParSets are realized by creating a set of SHORE storage server files, one on each SHORE server over which the ParSet is declustered. (Subsection 3.4 contains specific details on how a ParSet gets created.)

3.3 ParSets in SHORE architecture

The SHORE process structure augmented to support ParSets on a three-node shared nothing system is shown in Fig. 3. In addition to the master and slave application-level processes, one node runs a ParSet Server (PSS) process. The PSS process provides two distinct services: a catalog service and a slave managing service. The catalog service is responsible for maintaining a catalog of all ParSets that have been created. ParSet information includes type, declustering strategy, and the nodes over which the ParSet is distributed. The slave managing service maintains a separate catalog of all existing slaves for each known ParSet application. This service is also responsible for the creation and destruction of slaves on behalf of master applications. Lastly, the PSS is also responsible for maintaining the mapping between node IDs and the SHORE servers. The PSS process makes use of the SHORE storage manager to maintain a persistent catalog. The role of the PSS, in various ParSet operations, will be discussed in detail later in this section.

As mentioned earlier, the programmer need not write any slave code; the master process, which executes user-written code, and the slave processes, which carries out ParSet operations on the actual SHORE objects, are derived from the same application code. The ParSet library, which is linked with the application, contains the slave code. The PSS spawns the slave process with the binary derived from the same application code as the master process, but supplies command-line options that force the process to execute slave code instead of master code. Sharing the application code enables slave process access user functions that the master invokes in a ParSet call.

It should also be noted that the slave processes are threaded; therefore, they may service requests of more than one master process, as long as they are all derived from the same application binary. If sharing of slave processes with existing masters is not desired, the user only has to rename the application, or use a UNIX link with a different name, before executing the application.

3.4 ParSet operations

As suggested by Kilian, ParSets are implemented using parameterized classes. This enables compile-time type-checking; it also enables the ParSet library to get a handle on the SDL type-object for its ParSet members. However, to prevent code explosion, the templated classes for both primary and secondary ParSets inherit nearly all their functionality from a non-templated, base ParSet class. The following excerpt from the ParSet class definitions specifies an application's interface to our ParSet facility (see top of next page).

Applications manipulate ParSets through a transient C++ object, analogous to a file handle used manipulate a file in unix. All ParSet operations are invoked as methods of this in memory object. Aspects of declustering, persistence, and communication among servers are hidden from a parallel application that uses the ParSets. There are also several overloaded versions of some ParSet operations that are not shown in the class definitions. For example, there are versions of Apply operation without the filterSet parameter.

3.4.1 ParSet creation – create(tid, dbname, parsetName, nodes, declusterMode)

Note that there are two similar but different operations on ParSets. The first is the creation of the ParSet itself; this is of course done exactly once per ParSet. The second operation is an Open call that creates a transient C++ object for accessing and manipulating a ParSet. This Open call corresponds to opening an existing file, while the Create call corresponds to creating a new file. The steps involved in creating a ParSet are illustrated in Fig. 4.

Any process (slave or master) may create a new ParSet. To create a ParSet, the initiating process sends a message to the PSS to create a ParSet by invoking the ParSet library function *PrimaryParSet::Create()*. The parameters include: database name, ParSet name, declustering method (hash,

```
class ParSet {
 protected:
  // Routines called by all public ParSet methods
  int Open(XactId tid);
  int Apply(XactId tid, Set* resultSet, void* (*functName)(void*, void *),
            void* functArgs, int argSize, const TSet<OID>& filterSet);
  int PSApply(XactId tid, ParSet& resultSet, void* (*functName)(void*, void *),
              void* functArgs, int argSize, const TSet<OID>& filterSet);
  int Select(XactId tid, TSet<OID>& resultSet, int (*predicate)(void*, void*),
            void* predicateArgs, int argSize, const TSet<OID>& filterSet);
  int PSSelect(XactId tid, ParSet& resultSet, int (*predicate)(void*, void*),
               void* predicateArgs, int argSize, const TSet<OID>& filterSet);
  ReduceStruct Reduce(XactId tid, PSReduceOps operation, void (*valExtractor)(void*, void*),
                      void* extArgs, int argSize, const TSet<OID>& filterSet);
  static void Slave();  // Code executed by slaves (called automatically by ParSet::Init())
 public:
  enum PSDecluster { Undef, Hash, Modulo, Random, UserDefined };  // Supported decluster methods
  // Desired Reduce operation (specified by user in 'Reduce' member call)
  enum PSReduceOps { Max, Min, Sum, Avg, Count };
  static int Delete(XactId tid, char* dbName, char* parSetName);
  int Remove(XactID tid, OID oidSet);
  static int Init(int& argc, char* argv[]);
  int Close(void);
  static int Finish(char killSlaves = TRUE);
};

template <class Type>
class PrimaryParSet : public ParSet
{
 public:
  static int Create(XactId tid, char* dbName, char* parsetName,
                    const TSet<int>& nodes, PSDecluster declusterMode);
  PrimaryParSet(char* dbName, char* parSetName);
  int New(XactId tid, char* obj, int num, OID* oids);
  int Open(XactId tid, functPtr constructor, int argSize, functPtr decluster);
  // Templated and overloaded wrappers for Apply, PSApply, Select, PSSelect and Reduce not shown.
};

template <class Type>
class SecondaryParSet : public ParSet
{
 public:
  static int Create(XactId tid, char* dbName, char* parsetName, const TSet<int>& nodes);
  SecondaryParSet(XactId tid, char* dbName, char* parSetName);
  int Insert(XactId tid, OID* oid, int num, int node);
  // Templated and overloaded wrappers for Apply, PSApply, Select, PSSelect and Reduce not shown.
};
```

range, . . .), and set of nodes for declustering. The declustering method is implicit for a secondary ParSet and is not specified.

The PSS first checks with the catalog that the ParSet name is unique within the context of the specified database. If so, it creates a catalog entry for the ParSet recording all necessary information. The PSS then sends "create ParSet fragment" messages to the slave processes on the nodes (nodes 1 and 2 in this example) requesting that a SHORE file be created to hold objects in the ParSet. It is possible that the list of nodes for the ParSet may include a node on which the application currently does not have a slave process. When this happens the slave manager is requested to create new slave processes on behalf of the application. An acknowledgement is sent to the master once the file fragments are created.

3.4.2 ParSet open – Open(tid)

Before use, a ParSet must first be opened. This is done automatically before every ParSet call that manipulates the ParSet. Opening a ParSet results in the following sequence of operations shown in Fig. 5.

The master process first consults the catalog service to determine which nodes contain relevant fragments of the specified ParSet. Since this information is constant, the master need only contact the catalog service once per ParSet; catalog information is cached within the application. The master then consults its local cache for information regarding slaves (slaves are unique to applications, not ParSets). If new slave processes are needed, the master sends a request to create slaves to the slave service on the PSS. Note that unless Slave processes crash, this operation will also be performed at most once per ParSet (hence, communication will seldom be as messy as in Fig. 5). Information about the

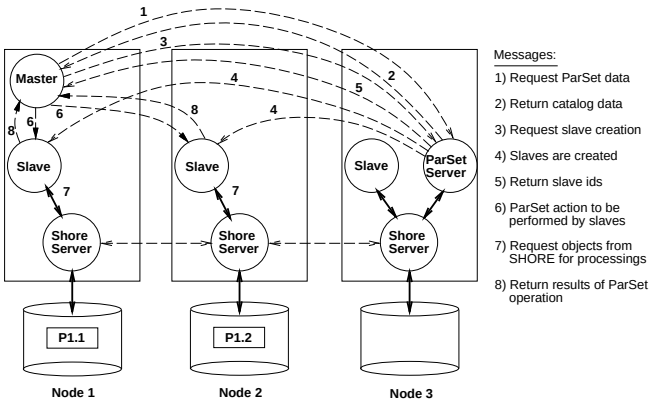

Fig. 5. Process communication on Open

new slaves is then added to the master's local cache. For example, in Fig. 5, an Open on ParSet P1 by an application running on node 1 results in slave processes being forked on nodes 1 and 2.

As mentioned earlier, the PSS maintains slave information for each registered application as a part of its slave service. Any process, including slaves, may consult the slave service to determine the location of the slave processes. This capability is needed to allow a slave process to operate on a ParSet. For a slave to open a ParSet, it needs to be able to communicate with the other slaves and possibly create new slaves where none exist.

3.4.3 New – New(tid, objs[], num, oids[]) – primary ParSets only

In Kilian's initial Parallel Set this operation was called Add, but for a primary ParSet this is a bad choice as it makes it sound like the object already exists. Objects in primary ParSets must be created in the context of the ParSet in which they are to reside.

This operation creates new objects of the ParSet type in the context of the specified ParSet by executing the following sequence of operations:

1. The application process (either master or slave) uses the decluster method associated with the ParSet to map the object values contained in objs[] to node IDs. These node IDs may not be on the same node on which the application process is running and may in fact all be different. The object values are then partitioned according to their mapped node IDs.
2. The application process constructs messages of the form <*new, DBname, ParSetName, objectValues, numObjs, ObjectType, objectSize*> for each partition and sends them to the slave processes executing on the corresponding nodes.
3. The slave on each node then calls the appropriate constructor on the object values it was sent. This creates new objects of the desired type which the slave then adds to the appropriate SHORE file. The OIDs received from SHORE for the new objects are then returned to the calling process via a message.
4. The invoking process collects the OIDs from all the slave processes. The OIDs are then reordered and packed into the supplied OID array in an order such that oids[i] is the correct OID for the object created using object value objs[i].

Notice that the new objects are created on the nodes where they are registered with SHORE. The objects are not created by the master. This is a limitation of the current version of SHORE; an object cannot migrate its home, so an object must be created on the node on which it resides.

3.4.4 Insert – Insert(tid, oids[], num, node) – secondary ParSets only

This operation adds OIDs of objects of to a secondary ParSet. Its execution causes the following sequence of operations to be performed: The application process sends all the OIDs in bulk to the specified node, which in turn adds them to the appropriate SHORE file. If no node is specified, the master will try to insert the OIDs locally. If the ParSet is not declustered on the local node, a node is chosen at random.

3.4.5 Apply – Apply(tid, resultSet, function, functArgs, argSize, oidFilterSet)

This method applies a function to each object in the ParSet. The applied function takes two parameters, functArgs and argSize. functArgs is a pointer to a structure containing function arguments and argSize is the size of this structure; these two are passed as arguments to the method invoked by Apply. The use of these two arguments will be clear when we get to the programming example. If oidFilterSet is not NULL, the function is applied only to objects whose OIDs are contained in oidFilterSet. The result is a set of values corresponding to the function's return type; this set is returned to the initiating process unless the resultSet is NULL. Execution of the Apply operation involves the following sequence of steps:

1. The initiating process maps the function pointer to a functId and sends messages of the form <*apply, DBName, ParSetName, resultSet, functId, functArgs, argSize, oidFilterSet*> to each slave process (the set of slave processes can be determined from the local slave-id cache).
2. Each slave process executes the following steps in parallel:
   a) The functId is converted to a pointer to the corresponding function (a memory address).
   b) oidFilterSet (if it is non-NULL) is used to construct an in-memory hash table.
   c) A scan is initiated on the file identified by shoreFileId. The following loop is executed for Apply's on Primary ParSets (see top of next page).

If the oidFilterSet is sufficiently small, the OIDs contained may be used directly to retrieve objects from SHORE. "Sufficient" implies that a selective retrieval will be faster than scanning the entire extent.

```
while (<oid, obj> = GetNextObject(shoreScanId)) {
    if (oidFilterSet != NULL) {
        // check to see if oid is an element of the filter set
        if ((probe_hash_table(oid)) == MISS) continue;
    }
    if (argSize < 0) resultSet.Add(function(obj));
    else resultSet.Add(function(obj, functArgs));
}
```

The following loop is executed for Apply Operations on Secondary ParSets:

```
while (<oid> = GetNextObject(shoreScanId)) {
    if (oidFilterSet != NULL) {
        // check to see if oid is an element of the filter set
        if ((probe_hash_table(oid)) == MISS) continue;
    }
    obj = GetObjectByOid(oid);  // actually get the object
    if (argSize < 0) resultSet.Add(function(obj));
    else resultSet.Add(function(obj, functArgs));
}
```

There is an alternative version of Apply called *PSApply()* which combines the functionalities of Apply and New. Instead of returning sets of values to the calling process, slaves executing a PSApply will use these values for the creation of new persistent objects within another primary ParSet that is specified as a parameter to this call.

3.4.6 Select – Select(tid, oidResultSet, predicate, functArgs, argSize, oidFilterSet)

This method applies a selection predicate to each object in the ParSet. Like Apply, the predicate may take an additional, user-supplied parameter. If oidFilterSet is not NULL, the predicate is applied only to objects whose OIDs are contained in oidFilterSet. The result is a set of OIDs of the objects that satisfy the predicate. This set is returned to the initiating process by each slave process. The difference between Select and Apply is that Apply invokes a method on each object in the ParSet, while Select applies a selection predicate such as `(age < 40) && (salary > 100,000)` to each element of the ParSet.

As with Apply, there is an alternative version of Select called *PSSelect()*. Similar to PSApply, PSSelect accepts a ParSet object in place of an ordinary set object; instead of returning OIDs to the calling process, OIDs are inserted into the specified secondary ParSet. The advantage here is that when the slaves perform the insertion, they attempt to insert the OIDs locally (if no local partition exists, a random node is chosen). In this way, we can force a secondary ParSet to duplicate the declustering strategy of another. This is extremely useful when PSApply is invoked on a primary ParSet, since this ensures that all objects referenced in the secondary ParSet will found locally avoiding any communication.

3.4.7 Reduce – (value,cnt) Reduce(tid, reduceOp, valExtractor, oidFilterSet, extArgs, argSize)

Reduce applies the specified function to every object in the ParSet. This function is often a method of the objects in the ParSet, and is assumed in our initial implementation to return an int or a float. (The motivation for this function call is to allow encapsulation – in many cases all the function will do is return the current value of a private data member of the object.) If oidFilterSet is not NULL, the operation is applied only to objects whose OIDs are contained in oidFilterSet. In the initial implementation, reductionOp must be one of Count, Sum, Average, Max, or Min. The result of the reduction is a (value,cnt) pair that is returned to the initiating process by each participating slave process: *value* holds the final reduced value, while *cnt* indicates the number of objects to which the operation was applied.

Note that logically, Reduce subsumes Apply, since Apply can be interpreted as a Reduce in which the combining function is a no-op (just return the set of values produced by the method calls). Likewise, Apply logically subsumes Select.

3.4.8 Remove – void Remove (tid, oidSet)

This operation simply requests that the set of objects identified by the oidSet be removed from the ParSet. For a primary ParSet, this is really a “destroy” operation since the ParSet is the only place in which the object exists. The master simply retrieves the object by its OID, checks to ensure the object resides in a SHORE file belonging to the ParSet and issues a SHORE delete on the object. For secondary ParSets, this operation is not so simple as it requires deleting the SHORE object whose value is part of the oidSet. This means that each slave will have to scan through its entire collection of OIDs for the ParSet in order to locate and delete the reference.

3.5 Programming example

To demonstrate how a programmer would use the ParSet library, we present a simple example. Note that we omit some of the trailing parameters in function calls for clarity. The database, in this example, consists of a primary ParSet of Employee objects, with data-members’ name, age, and job,

```
class EmployeeArg{
  char    name[80];
  int     age;
  JobType job;
};

class Employee{
     char name[80];
     int  age;
     JobType job;
     ... // Other data members
     Employee(EmployeeArg *consArg);};

int main(int argc, char *argv[])
{
  // Initialize ParSet environment
  ParSet::Init(argc, argv);

  //  Nodes over which ParSets will be declustered
  TSet<int> nodes;
  nodes.Add(0);  nodes.Add(1);    // add node id 0 & 1

  // Create the Primary and Secondary ParSets
  PrimaryParSet<Employee>::CreateParSet("BigCompany", "Employees", nodes, ParSet::UserDefined);
  SecondaryParSet<Employee>::CreateParSet("BigCompany", "Lawyers", nodes);

  // Instantiate ParSet handles and Open ParSets
  PrimaryParSet<Person>    employees("BigCompany", "Employees");
  SecondaryParSet<Person>  lawyers("BigCompany", "Lawyers");
  employees.Open(tid, Employee::Employee(EmployeeCore*), sizeof(EmployeeCore), &employee2node);
  lawyer.Open(tid); // tid is the transaction identifier

  // Create new, persistent Employee objects within the Primary ParSet
  int num;  EmployeeArg *args;
  LoadEmployeeArg(&num, &args);
  employees.New(tid, args, num);

  // Select OIDs of lawyers and add them to the 'lawyers' ParSet
  employees.PSSelect(tid, lawyers, &Employee::IsLawer);

  // Collect OIDs of retirable lawyers (over 65) into oidSet and Fire them
  TSet<OID> oidSet;  int retireAge = 65;
  lawyers.Select(tid, oidSet, &Employee::OlderThan, &retireAge, sizeof(int));
  lawyers.Apply(tid, NULL, &Employee::Retire, NULL, 0, oidSet);
  lawyers.Remove(tid, oidSet);

  // Find the average age of all the employees
  ReduceStruct avgAge = employees.Reduce(tid, ParSet::Avg, &Employee::Age);
  cout << "Total number of employees: " << avgAge.count << nl;
  cout << "Average age of employees: "  << avgAge.val.iVal << nl;

  // Cleanup ParSet facility and terminate slave processes
  employees.Close();  lawyer.Close();
  ParSet::Finish();
}
```

among others, and a secondary ParSet of employees who are also lawyers. An Employee object is created through a constructor that takes the structure *EmployeeArg* as an argument. First, the ParSet environment is initialized using `ParSet::Init()`. In this routine, the command line argument, `argv[]`, is used to determine whether the program executes as a master, in which case the user-written code is executed, or a slave, in which case the ParSet library's slave code is invoked (see top of page).

We create the two ParSets, Employees and Lawyers, in the BigCompany database. For both ParSets, parameters to create includes the nodes on which the objects are to be declustered. Both ParSets, in this example, are declustered over SHORE servers identified by 0 and 1. The PSS maintains the mapping from node numbers to their corresponding SHORE servers. The parameters to create the primary Parset, BigCompany/Employees, also include a declustering strategy. Here, a declustering strategy specified by the user when the ParSet is opened is used to decluster the objects. Once a ParSet is created, templated ParSet-class objects, *employees and lawyers*, are instantiated to manipulate the ParSets.

The ParSets are first opened so that they can be manipulated. The `Open()` method on `employees` takes the Employee constructor [Employee::Employee()] and declustering strategy *employee2node()* as its arguments. If the user-defined declustering strategy was not chosen, the declustering strategy will not be a parameter. The declustering method maps EmployeeArg objects to specific node IDs on the basis of odd or even age.

To create and add Employee objects into the primary ParSet, the `LoadEmployeeArg()` is called to obtain an array of arguments for the Employee constructor. `employees.New()` is invoked with the array as its argument which will create the persistent Employee objects.

As this is the first ParSet call that manipulates a ParSet, the PSS creates slave processes on nodes 0 and 1.

Having created our database of employees, we now wish to identify employees that are lawyers and add them into the lawyers' secondary ParSet. This can be achieved by collecting OIDs by invoking `employees.Select(...)` followed by inserting the collected OIDs by invoking `lawyers.Insert(...)`. Instead, we used `employees.PSSelect()`, as a performance optimization, to ensure that lawyer OIDs are placed on the same SHORE server as their corresponding Employee objects. This ensures that subsequent calls on the `lawyers` ParSet will only require examing only the local SHORE server to locate the actual Employee objects. The argument to the select, `Employee::IsLawyer()`, evaluates to *True* if the Employee is a lawyer.

At this point, we decide to enforce a mandatory retirement policy by firing all the lawyers over 65 years of age. For this, `lawyers.Select()` is used to collect the OIDs of all Employee objects who are retirable lawyers. `Employee::OlderThan()` takes an argument "age" and evaluates *True*, for Employees above that age. The set of OIDs collected is used as a filter set to apply `Employees::Retire()` method on Employees who are retirable lawyers. With a filter set, `Apply()` can be performed on either `lawyers` or `employee` ParSet, to get the same result. Using `lawyers` will result in a SHORE file-scan of OIDs intermixed with individual retrievals for actual Employee objects; this would be expensive it the Employee objects could not be found locally. Using `employees` would only require a file-scan on the Employee objects, but this is undoubtedly a much larger scan. In this example, since *PSSelect()* ensured that Employees are found locally, we invoke `lawyers.Apply()`. Note that the filter-set we have collected can be used repeatedly, e.g., it can be used to remove retired lawyers, or perhaps insert them into another secondary ParSet of pensioners.

Finally, to demonstrate reduction, the average age of all the employees is calculated by invoking `employees.Reduce()` with `Employee::Age()` (for extracting ages from Employee) and Avg as parameters. The result, which is the average age, is returned in the object `avgAge`. Finally, the ParSet files are closed and this followed by a call to `ParSet::Finish()` which informs the PSS about the termination of the application and terminates the slaves.

4 The OO7 benchmark overview

In this section we describe enough of the OO7 benchmark to make this paper self-contained; a more detailed description of the benchmark, along with performance results from a number of commercial systems, appears in Carey et al. (1993).

4.1 The OO7 database

The OO7 benchmark is intended to be suggestive of many different CAD/CAM/CASE applications, although in its details it does not model any specific application. The goal of the benchmark is to test many aspects of system performance, rather than to model a specific application.

4.1.1 Composite parts and documents

A key component of the OO7 benchmark database is a set of *composite parts*. The number of composite parts in this set is controlled by the parameter *NumCompPerModule*, which was set to 500 in our experiments. Each of these composite parts corresponds perhaps to a register cell in a VLSI CAD application, or perhaps a procedure in a CASE application. A composite part has a number of attributes, including the integer attributes `id` and `buildDate`, and a small character array `type`. Associated with each composite part is a *document* object, which models a small amount of documentation associated with the composite part. A composite part object and its document object are connected by a bi-directional association.

In addition to its scalar attributes and its association with a document object, each composite part has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. In the small benchmark, each composite part's graph contains 20 atomic parts, while in the medium benchmark, each composite part's graph contains 200 atomic parts. For example, if a composite part corresponds to a procedure in a CASE application, each of the atomic parts in its associated graph might correspond to a variable, statement, or expression in the procedure. One atomic part in each composite part's graph is designated the "root part."

Each atomic part has the integer attributes `id`, `buildDate`, `x`, `y`, and `docId` and the small character array `buildDate`. (Most of these attributes are not used in the traversals we selected for this ParSet work.) In addition to these attributes, each atomic part is connected via a bi-directional association to three other atomic parts. The connections between atomic parts are implemented by interposing a connection object between each pair of connected atomic parts. Here the intuition is that the connections themselves contain data; the connection object is the repository for that data. A connection object contains the integer field `length` and the short character array `type`. Figure 6 depicts a composite part, its associated document object, and its associated graph of atomic parts.

4.1.2 Assemblies and modules

The composite parts and their associated atomic parts (including the connection objects) and documents comprise the bulk of the OO7 database. However, a set of composite parts by itself is not sufficiently structured to support all the operations we wished to include in the benchmark. Accordingly we added an "assembly hierarchy" to the database. Intuitively, the assembly objects correspond to higher-level constructs in the application being modeled in the database. For example, in a CAD application an assembly might correspond to a register file, or an ALU. Each assembly is either made up of composite parts (in which case it is a *base assembly*) or made up of other assembly objects (in which case it is a *complex assembly*).

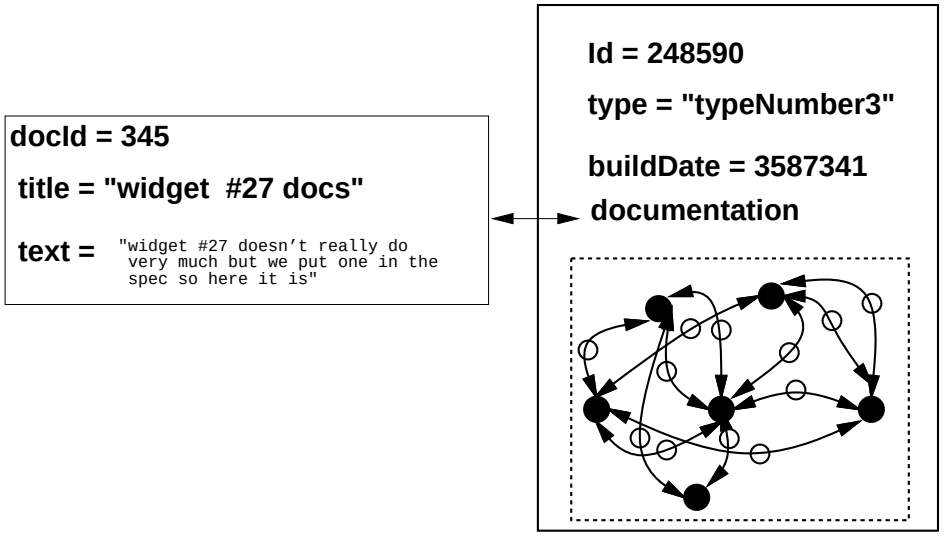

Fig. 6. A composite part and its associated document object

Table 1. Benchmark parameters

Parameter	Small	Medium
NumAtomicPerComp	20	200
NumConnPerAtomic	3–9	3–9
DocumentSize	20 KB	200 KB
ManualSize	100 KB	1 MB
NumCompPerModule	500	500
NumAssmPerAssm	3	3
NumAssmLevels	7	7
NumCompPerAssm	3	3
NumModules	1	1

The first level of the assembly hierarchy consists of *base assembly* objects. Base assembly objects have the integer attributes `id` and `buildDate`, and the short character array `type`. Each base assembly has a bi-directional association with three composite parts.

Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has the usual integer attributes `id` and `buildDate` and the short character array `type`; additionally, it has a bi-directional association with three; subassemblies (controlled by the parameter *NumAssmPerAssm*, which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). There are seven levels in the assembly hierarchy (controlled by the parameter *NumAssmLevels*).

Figure 7 depicts the full structure of the single-user OO7 benchmark database. Note that the picture is somewhat misleading in terms of scale; there are only $(3^7 - 1)/2 = 1093$ assemblies per module in the database, compared to 10 000 atomic parts per module in the small database, and 100 000 atomic parts per module in the medium database. Table 1 summarizes the parameters of the single user OO7 benchmark database.

In the ParSet work we needed to scale the database over a wider range of sizes than just small and medium; we did so by varying the number of modules.

4.2 The OO7 traversals

The OO7 traversal operations are implemented as methods of the objects in the database; a traversal navigates procedurally from object to object, calling the appropriate method on each object as it is visited. Some of the traversals update objects as they are encountered; other traversals call a "null" procedure on each object as it is visited, to simulate an application-level procedure call.

In this paper we used a subset of the OO7 traversals: two read-only traversals (traversals 1 and 6) and one update traversal (traversal 2b). Each of the read-only traversals were run in two ways, "cold" and "hot," while the update traversal was always run "cold." In a cold run of the traversal, the traversal begins with the database caches empty. We took great pains to flush the caches between runs.

4.2.1 Traversal 1: raw traversal speed

Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth-first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done.

This traversal is a test of raw pointer traversal speed, and that it is essentially equivalent to the performance metric most frequently cited from the OO1 benchmark. Note that due to the high degree of locality in the benchmark, there should be a non-trivial number of cache hits even in the cold case. Also, in order to implement the depth first search, the benchmark must tightly interleave DBMS and application data operations (e.g., to keep track of visited atomic parts while traversing a given composite part).

4.2.2 Traversal 2b: traversal with updates

Repeat traversal 1, but update every atomic part as it is encountered.

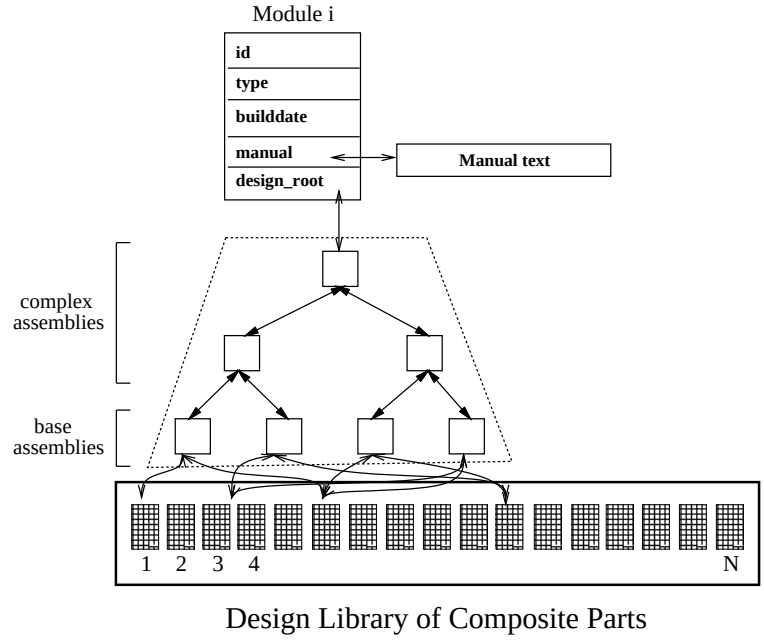

Fig. 7. Structure of a module

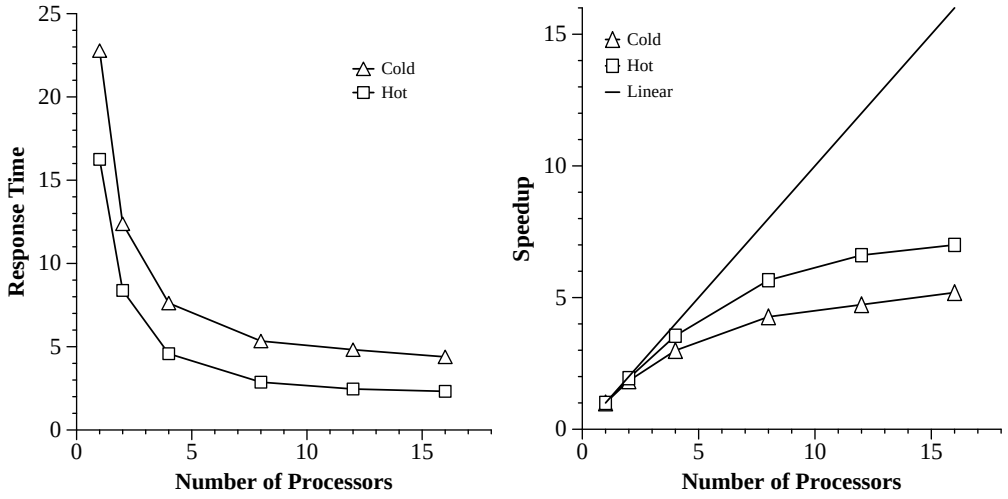

Fig. 8. Speedup for T1 on small database

4.2.3 Traversal 6: sparse traversal

Traversal 6 is the same as traversal 1 in the assembly hierarchy, but instead of performing a depth first search on all the atomic parts in each composite part, traversal 6 merely visits the root atomic part in each composite part.

4.3 ParSets and parallel OO7

How can we use ParSets to parallelize the OO7 traversals? Recall that the OO7 traversals all begin by doing a depth-first traversal of the assembly hierarchy, following the references from base assemblies to composite parts whenever they reach a base assembly. Once a composite part has been reached, the traversals differ: traversal 1 (T1) then does a depth-first traversal of the atomic part graph associated with the composite part, traversal 2b (T2b) does the same but updates a field in each atomic part as it is visited, while traversal 6 (T6) visits just the "root" atomic part in the composite part's associated graph of atomic parts. In all cases, the traversals are implemented as methods of the objects. That is, the T1 method on a complex assembly calls the T1 method of each of its subassemblies in turn, the T1 method on a base assembly calls the T1 method of each of its child composite parts in turn, etc.

In order to parallelize such a traversal we need to agree on what semantics of the traversal need to be preserved. If we require that the traversal visit all the objects sequentially in DFS order, then by definition there is no parallelism and we are done. This is rather uninteresting. Instead of requiring these semantics, we instead require the following:

1. Each traversal must start at the root of the module, and traverse the assembly hierarchy to determine which base assemblies belong to the module.

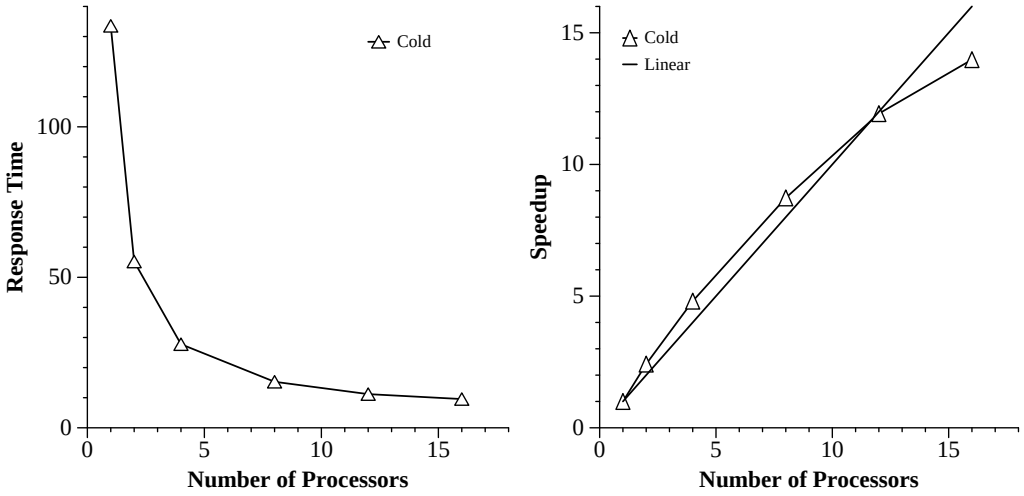

Fig. 9. Speedup for T2b on small database

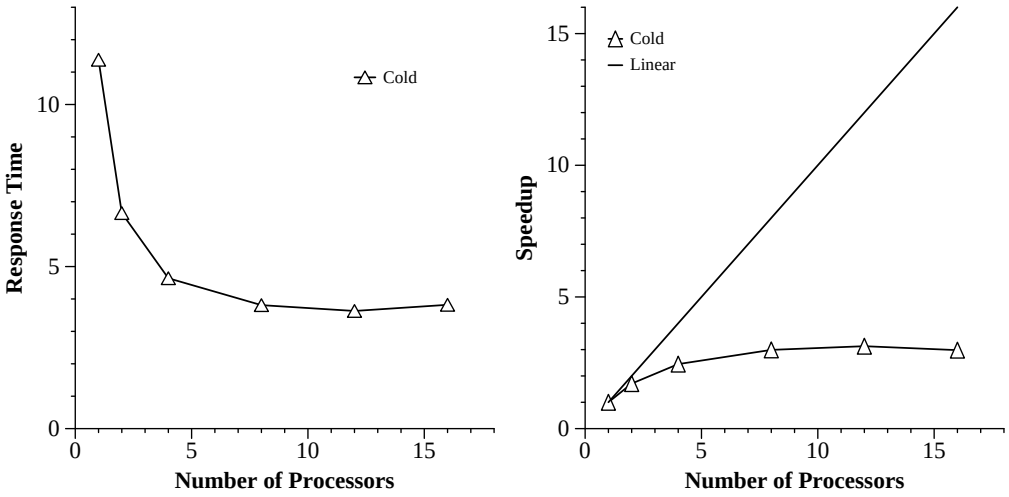

Fig. 10. Speedup for T6 on small database

2. Each composite part must have its traversal method invoked exactly as many times as it is invoked in the standard sequential implementation.
3. The composite part traversal method must visit the same objects as are touched by the sequential implementation, and furthermore it must traverse exactly the same links as the sequential implementation.

These semantics cannot be proven right or wrong, since no semantics were specified in the original OO7 traversals. Our rationale for choosing these semantics was that they ensure that:

1. Every object is visited in the parallel version exactly the same number of times as in the sequential version.
2. Every pointer is traversed in the parallel version exactly the same number of times as in the sequential version.

Furthermore, these semantics roughly match our original intention when we designed the OO7 traversals; the assembly hierarchy is there to provide access to the composite parts, and the composite parts together with their associated atomic part subgraphs form complex objects that should be acted upon as a unit.

Given these semantics, there are a wide variety of options for using ParSets to parallelize the traversals. Our choice was simple and effective: we created a ParSet of composite parts and clustered the atomic part subgraphs (which include the associate connection objects) with their associated composite parts. Then all of the traversals have the following form:

```
Perform a DFS on the assembly hierarchy;

Whenever a base assembly is reached, add
the OIDs of its referenced composite parts
to a filter set;

After the DFS of the assembly hierarchy
is complete, invoke a set Apply of the
appropriate traversal method (T1, T2b, or
T6) on the composite part ParSet with
the given filter set.
```

This ParSet Apply will ship the filter set to all the nodes that contain members of the composite parts ParSet, and invoke the appropriate traversal method on each composite part as many times as that composite part's OID appears in

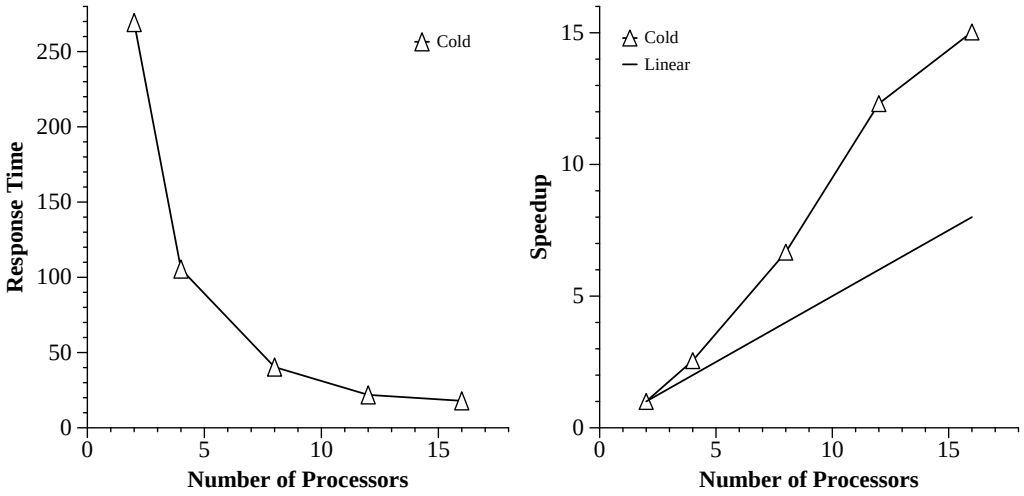

Fig. 11. Speedup for T1 on medium database

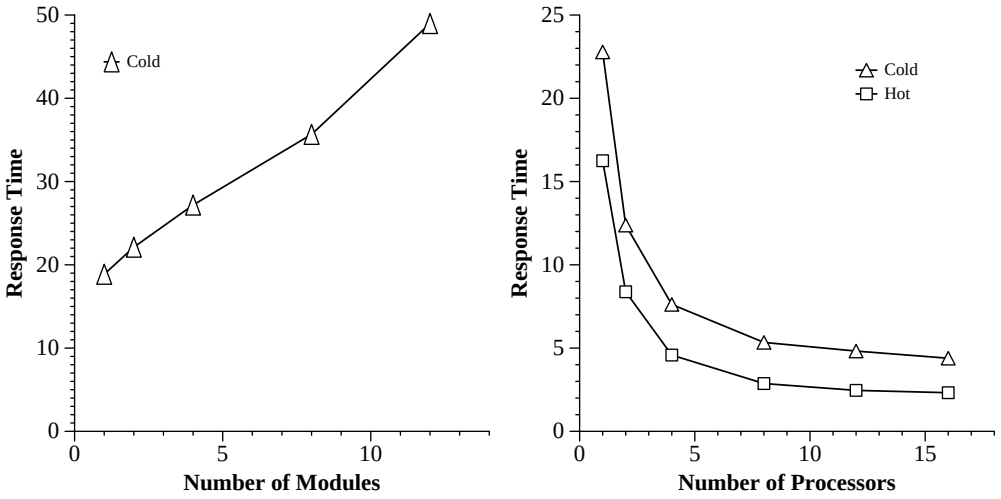

Fig. 12. Scaleup for T1 on small database

the filter set, so the implementation satisfies the specified semantics.

5 Performance results

We implemented the ParSet version of OO7 and ran a number of tests on a cluster of SparcStation 10/30 s connected by an ethernet. Each workstation had 32 MB of main memory and a Sun0424 424-MB internal disk drive. The server buffer pool was set to 10 MB; due to interference with the OS and with the SHORE client object cache, this was the largest we could use without introducing paging activity on the server buffer pool. The SparcStations in the cluster were not isolated, and since we did not have exclusive access to these workstations we did not kill the usual suite of daemons and background processes. However, we did ensure that there were no active users on the workstations when the tests were run. Furthermore, to reduce the impact of the "noise" in the data due to not having isolated workstations, we re-ran each test multiple times. The average of the trials is reported here

5.1 Speedup

In these tests we held the database size constant and varied the number of processors. For T1 on the small database, we obtained the results presented in Fig. 8. The left graph in the figure gives absolute times in seconds, while the right graph gives the normalized speedup.

The speedup is initially reasonable, but soon begins to fall far short of the ideal. The reason for this can be found in the details of T1: there is an initial sequential portion when the master program walks the assembly hierarchy collecting composite part OIDs. Only after this is complete can the master invoke the ParSet Apply, which the slaves then execute in parallel. For the cold test this took 2.2 s; as predicted by Amdahl's law, for larger numbers of processors this was a significant portion of the total execution and speedup suffered.

The hot curve shows worse speedup. The explanation here is somewhat more subtle. In the initial sequential base assembly traversal, no object is visited more than once; however, in the ParSet Apply on the composite parts, each composite part can be visited multiple times. These multiple visits mean that the ParSet Apply does a higher percent-

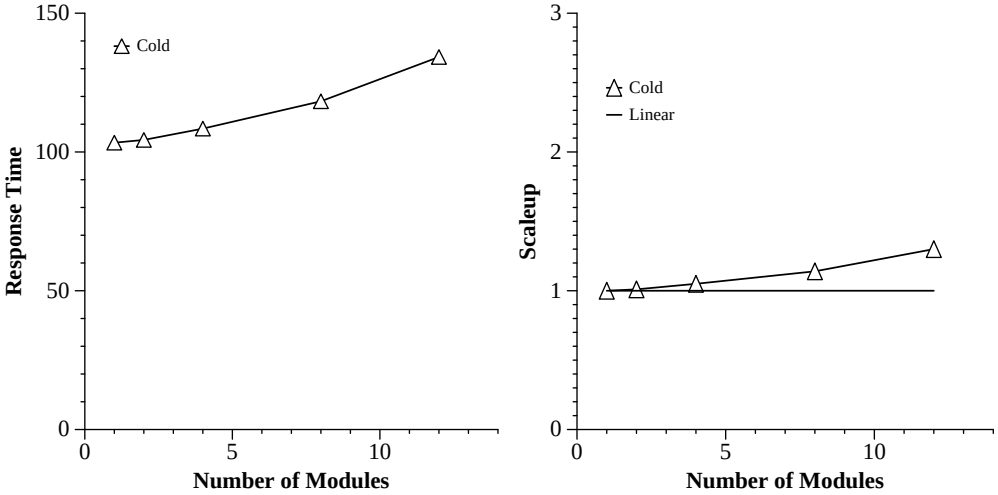

Fig. 13. Scaleup for T2b on small database

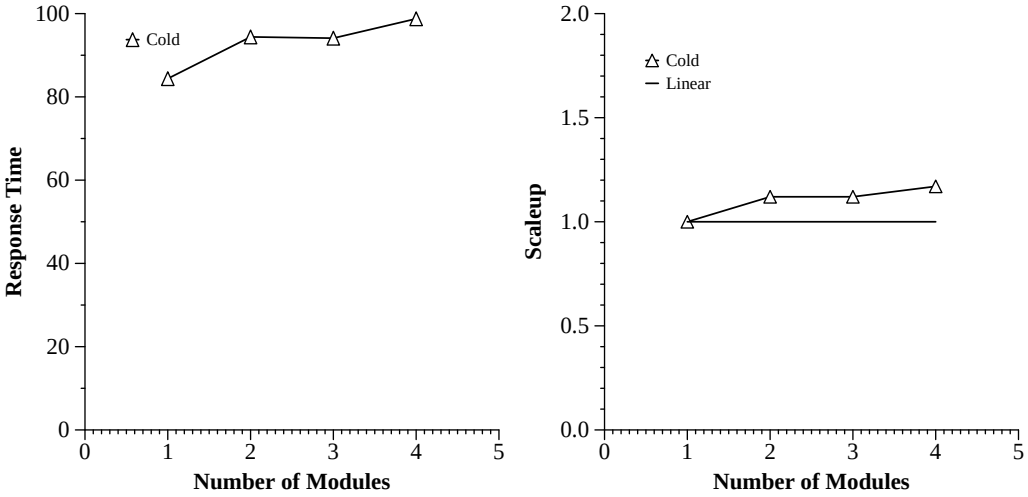

Fig. 14. Scaleup for T1 on medium database

age of work while the objects are in memory than does the base assembly traversal. This in turn means that the ratio of the hot to the cold times for the base assembly traversal is smaller than the ratio of the hot to the cold times for the ParSet Apply; in other words, the sequential portion of the cold traversal is proportionately smaller than it is for the hot traversals. This effect was repeated throughout the tests, so in the remainder of this section we present times only for the cold traversals.

Next, Fig. 9 shows the speedup curves for T2b on the small database. Here, the speedup is considerably better – in fact superlinear – than was the case for T1. The reason is that the base assembly traversal, the sequential part of the program, is unchanged from T1 to T2b, but the ParSet Apply does much more work in T2b, since it updates every atomic part as it is visited. The curve is superlinear because, with a single server, the database traversed by T2b does not fit in memory.

Finally, Fig. 10 shows the speedup for T6. Here the speedup is terrible; the reason is that here the set Apply does very little work, and the sequential base assembly traversal dominates the execution for all but very few processors.

Next, we move to the medium database, which is approximately a factor of 10 larger than the small database. The speedup curve for T1 on the medium database is given in Fig. 11. We started the curve in Fig. 11 with two processors rather than with one, since T1 took a long time to run with one processor. Here we observe wildly superlinear speedups; this is due to memory effects, since the database does not fit entirely in memory until 12 nodes. For the medium database, the database size is almost 120 MB. This superlinear speedup also occurred in T2b on the medium database, but not on T6 (since T6 only requires a tiny fraction of the database to be brought into memory). Since these spurious speedups due to memory effects are not the focus of this paper, we do not present the graphs here.

5.2 Scaleup

In these tests we grew the database in proportion to the number of processors; the motivation for such scaleup tests was to see if larger problems can be handled by adding processors to the system. Since the “official” OO7 database comes

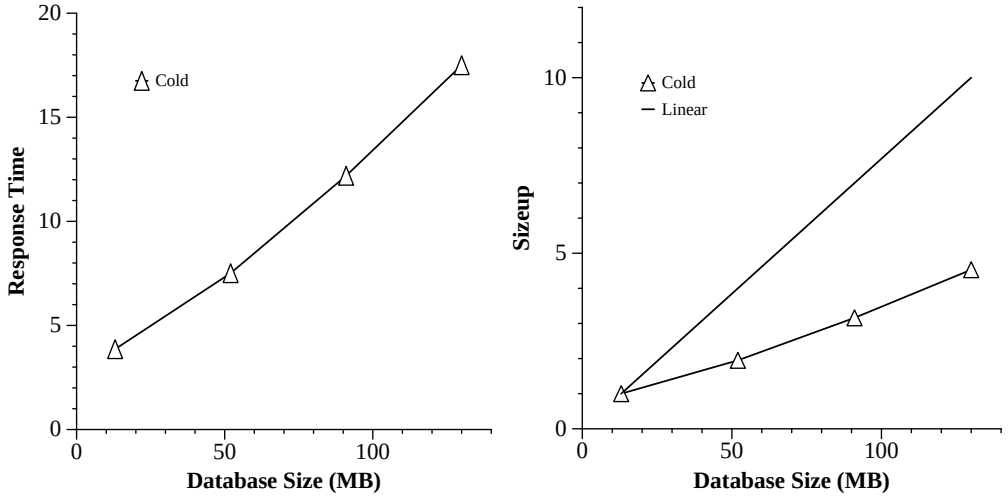

Fig. 15. Sizeup for T1 on various database sizes on 12 processors

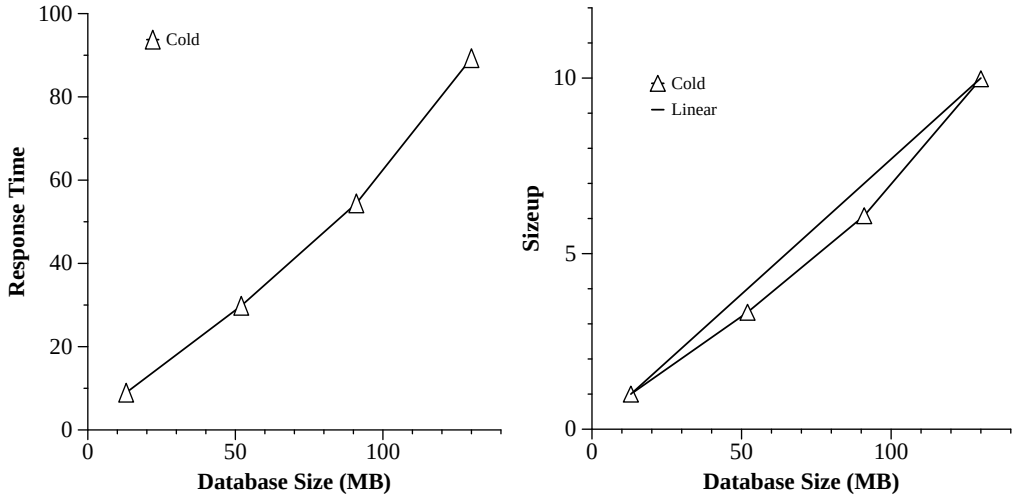

Fig. 16. Sizeup for T2b on various database sizes on 12 processors

in only two sizes (small and medium) we had to modify the database in order to be able to run these experiments. The way we did so was to increase the number of modules in the database in proportion to the number of processors. [This same scaleup is being used in the multiuser OO7 benchmark that is currently under development (Carey et al. 1994).] The only required change to the traversal code is that now the initial assembly traversal that gathers the OIDs of composite parts must traverse the assembly hierarchy of all modules.

Figure 12 shows the scaleup for T1 on multiple "small" modules. For this workload the database fits entirely in memory. As could be predicted from the speedup curves, the scaleup performance is not good (growing the problem size while adding processors does not keep running times constant). Again, this is due to the relatively large sequential traversal of the assembly hierarchy by T1.

For T2b on the small database, the scaleup is better, as expected – recall that for T2b the parallel portion of the traversal involves updates, hence it is much larger in relation to the initial sequential assembly hierarchy traversal. Figure 13 gives the results for this experiment.

Next, we move to scaleup numbers for the medium database. Figure 14 shows the results of these tests. Here, in contrast to the small database scaleup, we see excellent scaleup results: quadrupling the database size from 1 to 4 modules produces only a 20% increase in running time. This can be explained by examining the work involved in the parallel ParSet Apply portion on the two database sizes. For the small database, the ParSet Apply involves a simple traversal of a graph structure that fits entirely in memory. For the medium database, the ParSet Apply involves a traversal of a graph structure that fits only partially in memory, so paging occurs. For both the small and medium databases, the assembly hierarchy fits entirely in memory (the assembly hierarchy is in fact identical in the two databases). This means that in the medium database traversals, the ParSet Apply does a much larger fraction of the total work of the traversal than in the small database, so better scaleup occurs.

5.3 Sizeup

In these experiments we fixed the number of processors and grew the database. This time, we grew the database by vary-

ing the number of atomic parts per composite part, from 20 (small database) to 200 (large database) with two intervening points (80 and 140). Figure 15 shows the results of running T1 over these databases on 12 processors.

Note that the curve is sublinear, meaning that the algorithm is more efficient on larger databases. This is due to the parallel efficiency of T1 on the various database sizes. As the database grows, the ratio of the amount of work in the sequential portion of the traversal to the amount of work in the parallel portion of the traversal decreases. Another way to view this is that as the database grows, the fraction of the time for which 11 of the 12 processors are idle (awaiting a slave operation from a ParSet Apply) decreases.

Figure 16 shows the result of the same experiment, running T2b instead of T1. Here the sizeup is much closer to linear. The reason is that for T2b the traversal already had very good parallel efficiency for the small database, so moving to the larger database did not cause any significant improvement.

6 Conclusion

The ParSet facility within the SHORE system provides an easy-to-use means of adding data parallel execution to OODBMS applications, applicable whenever the OODBMS application invokes a method on each member of a collection of objects. Our experience with using this facility to parallelize some traversals of the OO7 OODBMS benchmark indicates that this approach to parallelism is effective for these workloads if the parallel portion of the workload is large in proportion to the sequential portion. One important case in which this criterion is satisfied is when the sequential portion of the application operates on memory-resident data, while the parallel portion of the application operates on a data set too large to fit entirely in memory.

Opportunities for future work in parallelizing OODBMS workloads abound. We plan to make persistent a parallel object-oriented language like PC++, using ParSets, and provide a suite of persistent parallel library classes, such as ParVectors, ParArrays, and ParMatrices, to go with it. We will also be studying optimization problems for implementing the library classes. We intend to experiment with multiuser workloads to see how well SHORE can take advantage of the parallelism inherent in multiple concurrent OODBMS applications. An abridged version of this paper appeared in DeWitt et al. 1994.

Acknowledgement. This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

References

1. Bancilhon F, Briggs G, Khoshafian S, Valduriez P (1987) FAD, a powerful and simple database language. In: Proceedings of the VLDB Conference, Brighton, UK
2. Cattell R (1993) The Object database standard: ODMG-93. Morgan Kaufmann, San Mateo, Calif
3. Carey MJ, DeWitt DJ, Franklin MJ, Hall NE, McAuliffe M, Naughton JF, Schuh DT, Solomon MH, Tan CK, Tsatalos O, White S, Zwilling MJ (1994) Shoring up persistent applications. In: Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data, Minneapolis, Minn, May
4. Carey MJ, DeWitt DJ, Naughton JF (1993) The OO7 benchmark. In: Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data, Washington DC, May
5. Carey MJ, DeWitt DJ, Naughton JF (1994) A status report on the 007 benchmarking effort. In: Proceedings of the ACM OOPSLA Conference, Portland OR, October
6. DeWitt DJ, Gray J (1992) Parallel database systems: The future of high performance database processing. Commun ACM 35: 85–98
7. DeWitt DJ, Naughton JF, Shafer J, Venkataraman S (1994) Parallelizing OODBMS traversals: a performance evaluation. In: Proceedings of the 1994 Conference on Parallel and Distributed Information Systems, Austin, Tex, September
8. Kilian MF (1992) Parallel Sets: an object-oriented methodology for massively parallel programming. PhD thesis, Harvard Center for Research in Computing Technology, Cambridge, Mass