データベースシステム　104−13
（1995．7．18）

オブジェクト並列原理に基づく超並列メインメモリデータベースシステムとデータベースプログラミング言語MAPPLE

今崎憲児，小野剛，牧之内顕文
imasaki@csce.kyushu-u.ac.jp
〒812-81 九州大学工学部情報工学科

並列コンピュータの巨大なメモリとたくさんのCPUは全てのデータが主記憶上に存在させることが可能なためメインメモリデータベースシステム(MMDBS)に適している．これによって，高速並列トラザクション処理が可能となる．この論文では分散メモリ並列コンピュータ上の超並列メインメモリオブジェクト指向データベース管理システムMAPPLE/DBについて述べる．最初に，MAPPLE/DBのためのデータベース定義・操作言語MAPPLEについて述べる．次にMAPPLE/DBのパフォーマンスをOO1ベンチマークを用いて示す．

Massively Parallel Main Memory Database System based on Object-Prallelism and Database Programming Language MAPPLE

Kenji Imasaki, Tsuyoshi Ono, and Akifumi Makinouchi

Department of Computer Science and Communication Engineering,
Kyushu University
Makinouchi Lab. Department of Computer Science and Communication Engineering, Faculty of Engineering, 812-81, Japan.

The massive main memory and a large number of CPUs of parallel computers are thought to be suitable for memory resident database systems(MMDBSs) since the whole data can be stored in main physical memory and high-speed parallel transaction processing is possible. This paper presents the design of MAPPLE/DB, a massively parallel, main memory object-oriented database management system on multicomputers. First, MAPPLE, the database programming language for MAPPLE/DB as database definition and manipulation is introduced. Next, the performance of MAPPLE/DB using OO1 Benchmark is shown in terms of its scalability.

1 Introduction

MAPPLE/DB is a massively parallel, main memory object-oriented DBMS that is under development. The first prototype of MAPPLE/DB aims at massively shared nothing parallel multicomputers such as Fujitsu AP-1000. MAPPLE/DB supports MAPPLE[AI94], a MAssively object-Parallel Programming LanguagE which enhances C++ for parallel processing as its object definition and manipulation language.

MAPPLE is designed to be used for writing data-intensive applications such as CAE, CAM, and CAM. Data-parallel programming languages [HQ91] in which arrays are a basic unit of parallel execution, are stuck with some limitations[AI94] for such applications.

MAPPLE adopts object-parallelism[GL91], which is enhancement of the data-parallelism. MAPPLE allows users to process arbitrary aggregates in parallel, to allocate objects dynamically to processors at run time and to handle the fluctuation of the number of data. These features make MAPPLE well-suited for data-intensive applications including database manipulation.

MAPPLE supports object persistence that most of the current OODBMSs do. However, 'persistence' for MAPPLE/DB differs from the one for ordinary disk-based database systems, in that persistent objects in MAPPLE/DB resides primarily in main memory. However, they are similar in that they are alive after the programs creating the objects, are terminated.

To support such persistent objects in MAPPLE/DB, persistent heaps are used. Any object created in a persistent heap becomes persistent and survives the termination of programs creating them. In addition, the persistent heap supports logging of the objects' modification and recovering them when they are lost.

The goal of MAPPLE/DB is to exploit the available resources of multicomputers: massive main memory(GByte order all together), a large number of CPUs and parallel accessible disks if any.

With massive main memory it becomes feasible to store entire databases in main memory, making MMDBSs a reality. Actually, some MMDBSs are aware of the impact of the massive main memory in parallel multicomputers[?][?]. In those systems, MMDBSs can provide much better response time and transaction throughput because data can be accessed directly in memory compared to conventional disk residential databases and a large number of CPUs enable user to process normal DB operations in a parallel manner and to get much better response time and transaction throughput. Although MAPPLE/DB is a MMDBS which needs no I/O operation for normal processing of DB operations, I/O operations are necessary for logging and checkpointing to guarantee the recoverbility of persistent heaps. Parallel accessible disks might be useful for such I/O operations.

This paper focuses on the performance scalability of normal DB operations for data-intensive applications such as CAD/CAM. We implemented an engineering database benchmark programs using MAPPLE and measured the performance on a Fujitsu AP1000 parallel computer.

This paper is organized as follows. The next section briefly introduces the 64-node parallel shared nothing multicomputer Fujitsu AP1000, and MAPPLE that is used as database definition and manipulation language in MAPPLE/DB. Section 3 describes the performance of MAPPLE/DB with OO1(Object Operation Version 1) Benchmark[CS92]. Finally, we sum up with the conclusion in section 4.

2 Hardware and Software Support

MAPPLE/DB is implemented on a parallel shared nothing multicomputer Fujitsu AP1000. This section summarizes its features and then introduces the database language MAPPLE.

2.1 The Architecture of AP1000

AP1000 is a highly parallel MIMD computer with distributed memory. It consists of 16 to 1024 processing elements(SPARC 25MHz), called cells, connected by three independent networks called T-Net, B-Net and S-Net. The host controls the flow of execution and the I/O operations.

AP1000 is equipped with a low-level software library which allows users to explicitly communicate and synchronize between processors.

2.2 MAPPLE

MAPPLE is a massively object-parallel programming language based on C++. C++ was chosen to base MAPPLE. In MAPPLE programs, objects to be processed are retrieved from 'containers' (i.e., databases). Containers may have different structures depending on the access patterns of applications. Since MAPPLE is based on C++, it allows users to implement their containers as 'set object' of C++. This makes it possible to tune the applications.

2.2.1 Object-Parallelism

Object-parallelism[GL91] is an extension of the data-parallel approach. In the object-parallel approach, users define objects which have data structures and methods. Then, those objects are allocated to PEs by the system. The same kind of methods are applied to each object in parallel. Each processor executes the method asynchronously until termination of each method. Figure 1 illustrates the object-parallelism in MAPPLE. In this approach, the so called frontend-backend configuration, in which the monitor would run on a frontend machine and the actual data management would be done in the backend, is adopted.

2.2.2 Categories of objects

In MAPPLE, objects are classified into two categories: *set objects* and *atomic objects*. Set objects are collections of objects and have interfaces for set operations such as *new*, *insert* and *delete* functions, and objects other than set objects are atomic. Users can implement set objects using list, array, hash table and so forth for their needs. In addition, we call objects contained in a set object *Element objects* which can be either set objects or atomic objects.

Each object, either set or atomic, is allocated in a PE. If the object is a set object, all elements of it must be allocated in the same PE where the set object is allocated.

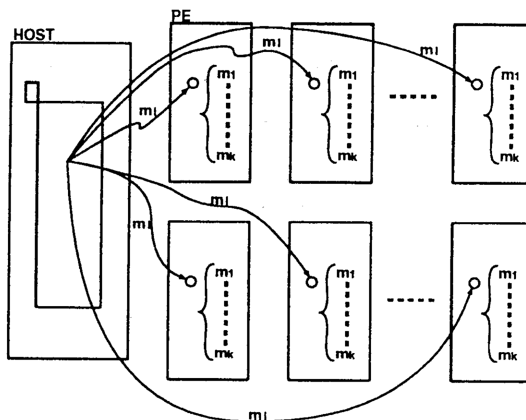

Figure 1: Execution of a method m_i in MAPPLE

To manage objects distributed in different PEs, *Union objects* are introduced for the management. A union object is a kind of set objects but differs from set objects in that it contains objects allocated in different PEs. In addition, elements of a union object must have same type and only one object in a PE can participate in the union object. A union object allows users to execute distributed methods of a same type on its all element objects distributed on all PEs in parallel. Figure 2 illustrates various objects in MAPPLE.

In addition to those categories, *remote objects* and *local objects* are distinguished in MAPPLE. Remote objects are those which may be referenced by another PE while local objects are those which are referenced locally; the local objects are normal objects in C++. Elements of a union object must be global objects. They are usually set objects. Element objects of the set are usually local objects which are referenced only through the set object. However, the elements may be global, if applications need it.

2.2.3 Object IDentifier(OID)

In MAPPLE, each global object is identified by its OID which is the pair of ORT entry number which we explain later and its PE number. Local objects are referenced locally by their local

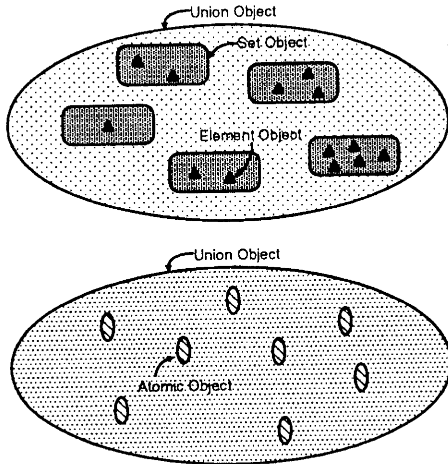

Figure 2: An illustration of various objects in MAPPLE

addresses.

2.2.4 Volatile Heap Management Objects(VHMOs) and Persistent Heap Management Objects(PHMOs)

VHMOs and *PHMOs* are system objects which manage a heap area in which objects are created on a PE. They have a same structure as shown in Figure 3. The difference between the two is that PHMO has persistence, which will be discussed later. Their common functions are the followings.

- Creating and deleting objects on its own heap area
- Receiving a message and unpacking it to get method names, arguments and OIDs
- Invoking the methods of the designated objects which a PHMO or a VHMO recognizes

In the case of a two-dimensional multicomputer as its target machine, users declare an array of PHMOs with a statement of the form: `PHMO Phmo(8,8)`. PHMOs are created on PEs and OIDs of PHMOs are returned to `Phmo` on the host processor. The two arguments define the size of PHMOs to be 8 by 8 on a multicomputer. Since a PHMO manages all objects on its own heap area, the PHMO must be created before the parallel portion of the main program is executed.

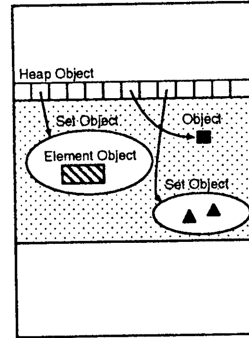

Figure 3: A Persistent Heap Management Object on a PE

A PHMO has additional functionalities that a VHMO doesn't have. They are;

- It is checkpointed at certain intervals. The modified pages are dumped out into backup files stored in disks.
- Logs are recorded whenever the data in the heap are modified.
- Data are loaded into the heap from the backup files after the data is lost because of some kinds of system failure.

To assure the addressability of the recovered data, the persistent heap area in each PE has a fixed beginning address and a fixed size. In the current design, a PE has only one persistent area. The address and size are determined by users when MAPPLE/DB is initialized. By this feature, addressing in a persistent heap is same as in a volatile heap managed by a VHMO, although some overhead for logging is expected when data are modified. Note that in the following performance measurement, this overhead is ignored, since logging is not yet implemented. However, only insertion test is affected by this

kind of overhead, and we think that the result of the test is valid in terms of performance scalability.

2.2.5 Union Objects

A union object manages distributed objects as a group. When users create a union object, an object (atomic or set) is created on each PE and the union object holds their global OIDs by receiving messages from each PE. To get the reference(global OID) to an object managed by a union object, a union object offers the method `ref(int i)` which returns the reference(global OID) to the object on PE_i. See Figure 4.

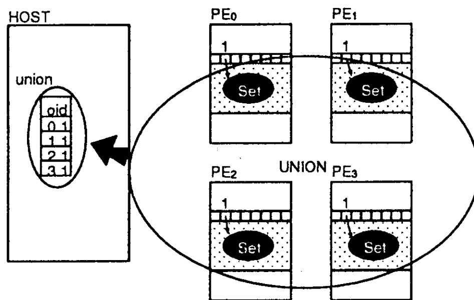

Figure 4: Union Object

2.2.6 Parallel Execution of Methods

Fujitsu AP1000 is equipped with high speed networks to broadcast messages. MAPPLE uses these broadcast networks for communication among processors based on object-parallelism.

In MAPPLE, users can execute a method of objects (set objects or atomic objects) in a union object. They can also execute a method of element objects in the set objects contained in the union object. In the latter case, a union object can be viewed as a big container stuffed with objects to be processed simultaneously.

To apply a method to objects in parallel, MAPPLE offers `for all` statement. `For all` statement has the following syntax:

```
for all [ type temp-val ]+ in union do {
  [ temp-val.method();]+
}
```

- *type:* Type of objects executing a method simultaneously
- *temp-val:* Temporary variable which indicates objects
- *union:* Pointer to a union object
- *method:* Name of the method executed in parallel

An example of `for all` statement is found in section 3.3.1.

3 Performance Evaluation Using OO1 Benchmark

OO1 Benchmark[GS92] is the benchmark for engineering DB applications. Because of its simplicity, OO1 has became one of standards for OODB benchmarking.

3.1 An Overview

The database of OO1 Benchmark consists of part objects and connections between them. Each part has five data fields: a part id, a type, an (x,y) coordinate pair, and a build date. Each part has exactly three out-going("to") connections to other parts plus a variable number of incoming("from") connections, and each connection has a type and a length. For locality in the object graph, OO1 parts are logically ordered by part id, and the "to" connections for each part are chosen so that each connection has a 90% chance of referencing a "nearby" part. The OO1 definition of "nearby" part is one within 1% of the part id space.

There are three OO1 Benchmark operations. The first is a part "lookup" operation, which looks up 1,000 random parts by their part ids. The second is an object graph "traverse" operation, which accesses 3,280 connected parts by selecting a random part and then performing a seven-level depth-first traverse(with multiple visits allowed) of the parts. The third OO1 Benchmark operation is an insert operation that adds 100 new parts to the database.

3.2 Distribution of objects

To store the entire part data, a persistent union object is defined. The union is a collection of the set objects each of which is created in the persistent heap of each PE. Part objects are allocated in the set objects distributed in all PEs. Since most of the references are to "nearby" parts, the block distribution is more reasonable than any other data distribution strategies such as interlieve or random. According to the block distribution strategy, each set in a PE has parts which have consecutive part ids. The following code generates part objects and distributes them over PEs.

```
P_Union<Part_t>*
  p_union = new P_UNION<Part_t>;
for(int i=0;i<MAX;i++)
  p_new(p_union->ref(i % PE)) Part_t();
```

where *P_UNION* is persistent union class and to insert objects into the object belonging to the class, users have to write *p_new* method.

3.3 Operations

3.3.1 Lookup

Parallel lookup operation is implemented as the following: First, the random part ids to be retrieved are generated at the host processor. Then, they are broadcasted as a lookup method argument in a packed form. Next, each PE(i.e., each set in each PE) receives it and invokes the lookup method. Since each PE knows the range of part ids of the parts stored in the PE, it discards uninteresting part ids. The method looks up the corresponding parts and inserts them into a result union object. Figure 5 describes the lookup operations.

```
Array_t ids;  target ids
Union<Result_t>  result_union;
for all Part_t p in p_union do
  p.lookup(result_union,ids);
```

Note that the result are stored in a volatile union object since the result is temporal.

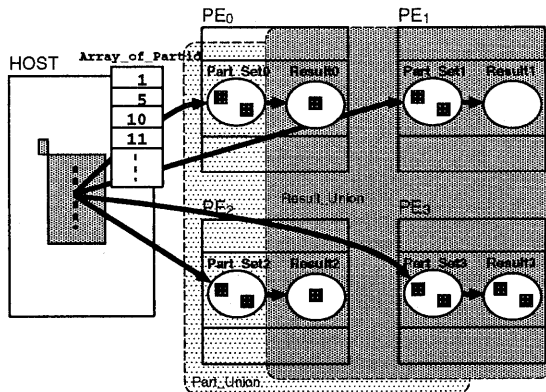

Figure 5: OO1 Lookup Operation

3.3.2 Traverse

The most difficult one of the three operations is traverse since the parts are distributed among PEs and traverse has to go across them. Implementing the operation, we had two choices on how to hold references to a part on another PE : part id or global OID. Using the former is rather a RDB style where value matching plays an important role, while the latter is a OODB style where traversing by pointers is used. Here, we adopted the former as a preliminary implementation.

The method of traverse on a PE is implemented as follows(Figure 7).

1. Getting part id to be traversed
2. performing a seven-level breadth-first search(although this may be different from the definition of the operation which is to perform depth-first search, we think the result is same except for the order.)
3. Having met the remote part id, the method inserts the part id and the current depth into a External Traverse(ET) set.
4. these steps are iterated until the method reaches at the certain number(in this case 7) of depth.

The ET set is implemented by a volatile union set which is shared among PEs. Having finished

the steps above, each processor broadcast its ET Set elements(Figure 7). Then, each PE receives corresponding ET Set elements and starts traverse again using each element. That iteration terminates when the ET set becomes empty.

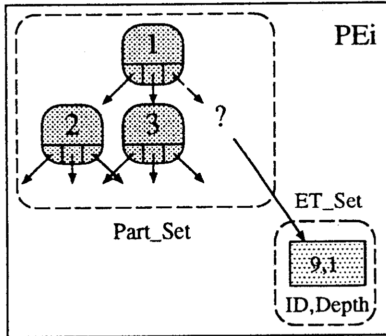

Figure 6: OO1 Traverse Operation-(1)

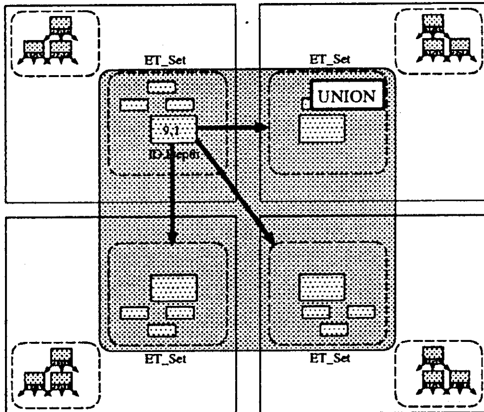

Figure 7: OO1 Traverse Operation-(2)

3.3.3 Insert

Parallel insert operation is almost same as the parallel lookup operation except the arguments of the insert method is not only part id but also the entire arguments of the constructor.

3.4 Performance

Figure 8 illustrates the performance scalability of the lookup operation on AP1000 when looking up 1000 parts out of 10,000 and 20,000 parts. The shown speedup is almost liner with the number of used PEs since there is almost no communication among the PEs.

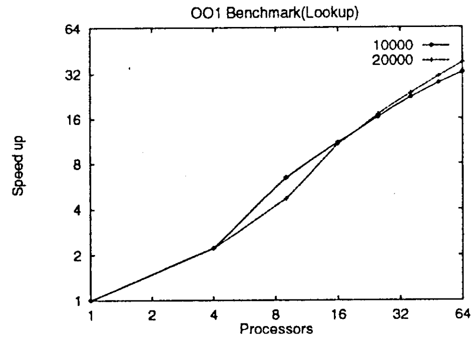

Figure 8: The speedup of OO1 lookup operation

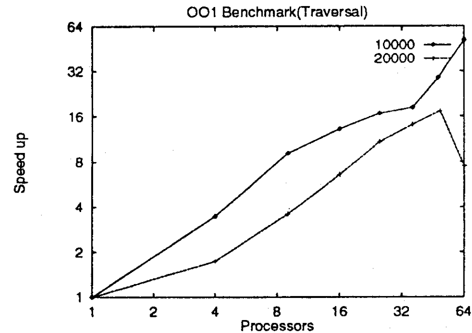

Figure 9: The speedup of OO1 Traverse operation

Figure 9 illustrates the performance scalability on AP1000 of the traverse operation. When the number of used PEs becomes greater than 40, the performance degrades because of the broadcast effects: as the number of PEs increases, the number of messages also increases; Since every part has three connections and 1 percent of them are remote references, the total

number of the messages sent among the involved PEs is (the number of part objects)*3*0.01*(the number of PEs). This is the reason why no more improvement is expected over 40 PEs.

Figure 10 illustrates the performance scalability on AP1000 of insert operation when inserting 100 parts. The speedup is almost liner with the number of PEs used since there is almost no communication among the PEs.

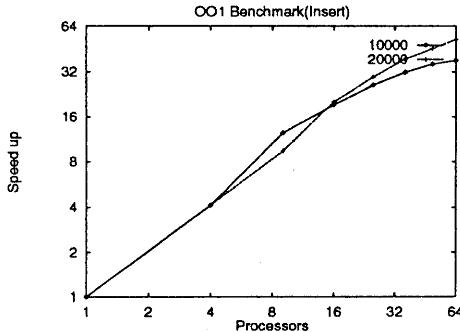

Figure 10: The speedup of OO1 Insert operation

4 Conclusions and Future Research

In this paper, we have discussed the design and implementation of the database language MAPPLE for a massively parallel main memory database system, MAPPLE/DB. The design of the system can be characterized by the two main ideas: use of parallelism and main memory data storage to provide high performance in transaction processing.

Using MAPPLE, OO1 Benchmark DB was built and the performance is shown. For both lookup and insert operation, where no communication among objects exists, the performance is almost ideal. For traverse operation, which perplex communication exists, the performance is dropped at a certain point. Our intention of the design of MAPPLE is to use broadcast making the most use of high speed broadcasting network of AP1000. However, as an alternative, it might be needed to include one-to-one communication among objects to improve performance in MAPPLE using its global OID. We leave this approach as a future work.

Currently, the second version of MAPPLE/DB is being implemented. The version will include the logging facility for crash recovery and transaction management.When it is completed, OO7 Benchmark will be a interesting benchmark for MAPPLE/DB.

References

[AI94] H. Amano, K. Imasaki, K. Fukumi, A. Makinouchi: *Design Policy and Preliminary Experiments of Object-Parallel Programming Language INADA/MPP*, Proc. of the 4th Symposium on Massively Parallel Processing, Sopnsored by Grant-in-Aid for Scientific Research on Priority Areas, pp.2-214–2-220, Mar. 1994(in Japanese).

[GL91] Gannon, D. and Lee, J. K.: *Object Oriented Parallelism : pC++ Ideas and Experiments*, Proc. of 1991 Joint Symposium on Parallel Processing, pp.13–23, 1991.

[HQ91] Hatcher,P.J. and Quinn,M.J.: *Data-Parallel Programming on MIMD Computers*, The MIT Press, pp.171–176, 1991.

[CS92] R. G. CATTELL and J. SKEEN: *Object Operations Benchmark*, ACM Transactions on Database Systems, Vol.17, No.1, Pages 1-31, Mar. 1992.

[GS92] Hector Garcia-Molina, and Kenneth Salem:*Main Memory Database Systems:An Overview*, IEEE Tansactions on Knowledge and Data Engineering, Vol.4, No.6, pp.509–pp.516, Dec. 1992.