A Distributed Parallel Object Manager for Smalltalk

Hasan Aytekin Asuman Dogac
Software Research and Development Center of
The Scientific and Technical Research Council of Turkey
Middle East Technical University
06531 Ankara Turkey
E-mail: asuman@vm.cc.metu.edu.tr

Abstract

In order to exploit the inherent parallelism in distributed systems through object-oriented programming a Distributed Parallel Object Manager for Smalltalk is implemented. In the previous Distributed Smalltalk implementations, the remote operational model is used where the sending process blocks, the receiver process performs the operation, a value is returned, and the sending process then resumes. In the Distributed Parallel Object Manager implementation, the sending process is not blocked, and there exists an environment associated with the local object at the remote host. Parallelism is achieved by using a variation of asynchronous communication; however the problems of asynchronous communication are avoided. Parallelism is implemented as a property of objects; not their classes. For parallel objects, since blocking is not desired between the task of sending a message and receiving its result, when a message is sent to an object, the value returned is a pointer to the result area of the object's message response.

I. Introduction

Smalltalk-80 [GOLD83] system operates on a single object domain environment. In this environment, a single user with a single object address space is provided. In Smalltalk-80 several processes can be active at the same time where each object can act on a collection of objects. However because of the lack of synchronisation and mutual exclusion in this model, parallel programming is not so attractive. This holds because a process must expect interaction from other processes at every point of execution.
In order to facilitate object sharing between users on different machines distributed versions of Smalltalk have been implemented [B 87, D 89].
In the distributed object manager model, the traditional remote operation model is used. The traditional operation model requires that the sending process blocks until a value is returned. This blocking mechanism reduces the efficiency of the remote operations. Another disadvantage of the remote operational model is that it does not create an environment at the remote host thus it is not possible to keep the most recent status of objects for a specific application. In other words, if an environment is not associated with a given application, it is not possible to send a follow up message related with this application. As an example, an application may be dealing with pointers and may want to preserve the state of pointers between messages. In such a case an environment is necessary.
Object oriented languages are sequential in nature as observed from the following restrictions[A 89]:
1. Execution starts with exactly one object being active.
2. Whenever an object sends a message, it does not do anything before the result of that message has arrived.
3. An object is only active when it is executing a method in response to an incoming message.
Parallelism can be integrated into an Object-Oriented language basically in two ways [A89]:
1. By relaxing the sequential restriction 2. This can be done by asynchronous communication. Instead of letting an object wait for the result after sending a message, the sender is allowed to go on with his own activities.
2. By relaxing the sequential restriction 3. Each object is allowed to be active at its body. When an object is created, it takes place parallel with the other objects in the system. An object can communicate with another object by interrupting the receiver object at certain explicitly indicated points. This requires synchronisation of both the sender and the receiver. Hence, a synchronous communication model is used.

Among the parallel object oriented language implementations, PROCOL[v 89] uses asynchronous communication, whereas POOL2[A 89] uses synchronous communication. The implementation of PROCOL is realised in the C programming language under the UNIX operating system. As UNIX is a multi-process system, objects can execute concurrently. POOL2 operates on DOOM machine which is a special purpose, multi processor architecture designed for object oriented programming. Neither of the languages supports inheritance.
In order to exploit the inherent parallelism in distributed systems we have introduced the Distributed Parallel Object Manager for Smalltalk that implements a variation of asynchronous communication where the sending process is not blocked, and there exists an environment associated with the local object at the remote host in order to provide a more efficient processing environment.

II. Previous Work

In [B 87], a Distributed Smalltalk is described which is operational on a network of Sun workstations. The system allows objects on different machines to send and respond to messages. The distributed aspects of the system are user transparent. The communication mechanism is a variation of the traditional remote operation model: the sending process blocks, the receiver process performs the operation, a value is returned, and the sending process then resumes.
The system also allows some capability for sharing objects among users. The classes and instances must be co-resident. Thus there is a restriction on object mobility, that is, when an object moves, its class must be present at the destination.
Remote objects in the system are supported through the use of proxyObjects and the RemoteObjectTable. A proxyObject represents a remote object to all objects in the local address space. There is one proxyObject per host per remote object referenced by that host. The RemoteObjectTable keeps track of all local objects that are remotely referenced.
In [D 89], another design for distributed Smalltalk is described. The solution given [B 87] is achieved through defining proxy objects as full-fledged Smalltalk objects, which necessitate the modification of the virtual image. However the solution of [D 89] is through the modification of the interpreter and the object manager levels. A local object manager runs on each workstation and provides the programmer with a collection of primitives. These primitives allow objects to be named and shared without the programmer being aware of their actual location.
The implementation uses the shared memory mechanism: each Smalltalk program is represented by a UNIX process and accesses Smalltalk objects that are stored in a common block of shared memory. Intersite communication is achieved by using the UNIX 4.2 bsd "socket" mechanism. In [v 89], a parallel object language, PROCOL, is described. PROCOL uses C as the host language. A PROCOL object type (abstract data type) is defined by means of a piece of program. Inheritance is not supported by the language. Objects are created by means of the new primitive. Communication is based on one way message transfer. The sender of the message waits until the message has been accepted by an intended receiver. A potential receiver is likewise suspended until it acquires the required message.
Receipt of the message consists of copying the values of the message's components to variables local to the object. Immediately after receipt of the message, sender and receiver resume execution. Any processing of the received message is done after the sender has been released. It is clear that asynchronous communication is used in PROCOL. The implementation of PROCOL is realised in the C programming language under the UNIX operating system. As UNIX is a multi- process system, objects can execute their operations concurrently.
POOL2 [A89] is designed to allow objects to run in parallel. It operates on DOOM machine which is a special purpose, multi processor architecture designed for object oriented programming. In POOL2 an object has an activity of its own, which is called its body. Execution of the body is started as soon as the object is created, and it takes place in parallel with the other objects in the system. An object can communicate with the other object by interrupting the receiver object at certain explicitly indicated points. This requires synchronisation of both the sender and the receiver. Therefore, a synchronous communication is used.
Both PROCOL and POOL2 fail to support inheritance which is an indispensible aspect of object oriented languages. In [AL 90] inheritence and subtyping is introduced to POOL family languages, however it is still impossible to inherit the bodies.

III. The Design of the Distributed Parallel Object Manager

In designing the distributed parallel object manager, the main idea is to establish a session between a local

object and a remote class. A session established may have three different types of communication partners at the remote class: a new object from the remote class or an object whose oop is given in a global variable or all instances of a given class. The term session in this case means that, whenever a local object is created for a remote class, an environment at the remote host is created and associated with a descriptor id to the local object. The hand shaking process is as shown in Figure 1.

Figure 1. Hand Shaking

When a message is sent to the local parallel object, it is passed to the remote host without causing any operation in the local host and executed at the local object's own environment at the remote host. The result is returned to the local object's result field. Since this environment is resident and waiting for a message to execute until a reset is received, the state of the object's class and instance variables is preserved and updated whenever a message is received. Sending a message to a ParallelObject and receiving its result is illustrated in Figure 2.

Figure 2. Sending a message to a ParallelObject and receiving its result

In order to achieve the purposes stated above a new Class called ParallelObject is defined as a subclass of Class Object. The main difference between the behaviour of a conventional object and the parallel object is the way the value is returned for a message sent to an object. For conventional Smalltalk objects, a value is always returned as a result of a message sent to the object. For parallel Smalltalk objects, blocking is not desired during the task of sending a message and receiving its result. Therefore when a message is sent to an object, the value returned is a pointer to the object's message response result area. There is a pointer to the related result field for each message sent to the object. The user is free to acquire the result of a message at any time. The criterion for acquiring the result of the message is the user's choice.

If the user acquires the result of the message immediately after sending a message, this corresponds to the remote operational model and the process is blocked until a value is returned. But as described above, the process of sending a message, its execution, and the value returned are all independent operations. Therefore, if the result of a message sent to the remote object, is not immediately required then the process is not blocked.

Since a parallel object owns an environment at the remote host during its lifetime, the local garbage collector never swaps this environment. Whenever the local parallel object is reset, the environment at the remote host is terminated, and left to the local garbage collector at that site. Therefore no modification is necessary for the present garbage collection schema.

A design criterion for terminating the parallel object's environment is sending a reset to the local parallel object. The user should send a reset message to the the local parallel object whenever it is not further required. If the user forgets to send a reset message, the environment timeouts and hence terminates itself after a predefined time unit after the current process terminates.

At the local host where the parallel object is physically resident, there is a result field pointer to the result of each message sent to the object. When a result is received from a remote host, the result field of the related message is updated. This field is never destroyed unless the user intentionally destroys it. Thus when the same message is sent to the same object with the same selectors and the arguments, the previously obtained results can be used. Therefore, the user should explicitly send a message to the parallel object to destroy a result.

An instance of a ParallelObject is defined by sending the message for:aClass of:aType to the class ParallelObject. It is clear that an instance of any class can be defined as a parallel object, parallelism is a property of instances,

not of classes. aType identifies the type of the communication partner. If aType is 1, the communication partner is a new object from aClass and the user has four alternatives in defining a parallel behaviour for the object. These are:

1. Use aClass from any available host.
2. Use aClass at host 'X'.
3. If aClass exists at host 'X', then use it, otherwise use aClass on any available host.
4. Migrate aClass to any one of the available hosts and use it there.

To implement these alternatives, the distributed object manager should support class replication on remote hosts. Also, class compatibility on remote hosts should be ensured for classes open to service. With class compatibility, we mean the following: the same classes and the same class hierarchy should exist in all of the sites. However the system does not support the transaction concept of Database Management Systems. Thus the instances of classes on different sites can be simultaneously updated. The only concurrency mechanism provided is the locking of a Class at a site to prevent the update of a ClassVariable in more than one environment at the same site. This provides for preserving the consistency of data kept in class variables at a given site.

If the communication partner is identified by a global variable or if it is the instance of a given class then the remote host must be specified explicitly.

Atomicity of methods for conventional objects is already provided by Smalltalk by not interleaving the execution of the methods at a given site. When an object is created by executing ParallelObject for: aClass message at site i, an environment and a message queue is created at site j for the messages to be received for aClass. Thus even if Smalltalk had not provided for the atomicity of the methods, the message queue does this job.

IV. Implementation of the Distributed Parallel Object Manager

The methods of class ParallelObject deal with two main operations. One is the local object site operation, and the other is the creation of the environments for each parallel object at the remote site. Note that in this environment there is no need to classify the sites as client and server, the same class can be used for each of the operations. Therefore, while a site creates an environment for an object at a second site, the second site can also ask to some other site to create an environment for its own local object. Thus, there can be chains established among hosts for a single object. That is, when a message is sent to a local parallel object, it is asked from the remote host to execute this message, and the remote host may ask another host to execute some other message for some operation needed for the original local distributed object. Therefore, it is possible for a single parallel object to have a distributed behaviour on many hosts.

Among four alternatives given in the design of the previous section, the policy number 1 is implemented for the time being.

For each message sent and its result received, the layout used for object and result descriptors are as follows:

Object Descriptor:

. The host that object belongs to. (This is not used in the current implementation due to the hardware restrictions that only two PC's are connected.)
. Spoolid which is a pointer to the Requests dictionary at the local host.
. Type of the communication partner:
  1. A new object
  2. An object whose oop is in a global variable
  3. All instances of a given class
. Pointer to the referenced environment.
. Selectors of the message sent to the object.
. Arguments of the message sent to the object.

Result Descriptor:

. Target host (Not implemented).
. Pointer of the object at the local host.
. Class of the value returned.
. The value itself.

The descriptors used for the service is determined by the initial byte received from the remote host. Their descriptions are as follows:

0 - A result from remote host to local host
1 - A request from local host to remote host
2 - Parallel hand shake request

3 - Parallel hand shake result
4 - Parallel object message
5 - Parallel object result

When a message is sent to a parallel object, the value returned to the local host is a pointer to the result area. The request made for each parallel object is marked as a request with a nil value initially. The structure used for such a request is given in Figure 3.

Figure 3. Requests Dictionary

At the remote host, for each environment created, there are two global pointers to this environment in order to maintain synchronization of the requests and their executions. One of them is used to inform the environment that a message is received, and the other is a queue (an OrderedCollection) of the messages received. The implementation is such that, when a message is received for an environment, with its return address, it is inserted at the back of the MessageQueue and the related environment's semaphore from the ParallelObjectClassDict is signalled. When the environment is informed that there is a message waiting for execution, the environment removes messages in the order of their spoolid, executes them, and returns the result to the address provided with the message itself. The structures of the ParallelObjectClassDict and the MessageQueue is as follows:

ParallelObjectClassDict (a dictionary)
at: (referenced environment) ---> a semaphore
MessageQueue (a dictionary)
at: (referenced environment) ---> a queue
a queue ---> an ordered collection
an ordered collection cell ---> a dictionary
a dictionary at: 1 ---> return address
2 ---> the message

Figure 4. Message Queue

VI. Comparing Distributed Parallel Object Manager with Previous Work

The two basic techniques for implementing parallel object oriented languages are the synchronous and asynchronous communication as explained in the introduction.

The disadvantages of asynchronous communication model given in [A89] are avoided in Distributed and Parallel Object Manager model although it uses a variation of asynchronous communication. A problem of asynchronous design is to guarantee that the messages travelling from the same sender to the same receiver should arrive in the order in which they were sent.

In the Distributed and Parallel Object Manager model, this problem does not exist. Because, waitsFor: and valueOf: methods of the ParallelObject class indirectly force the ordering. When a result is requested by the valueOf: message, it is obtained from a predefined location in the result area after confirming that the result has arrived. That is the process explicitly waits for the result if it has not received it until then. The received results do not impose any side effects in the execution. In the Distributed and Parallel Object Manager model, a result is interpreted as a memory location having a nil or an actual value and it is in effect only when it is requested. That is, the order which they arrive is not important.
In the Distributed and Parallel Object Manager model, the processing of messages is also performed according to the order they are sent and not in the order they are received. This is achieved by processing the received messages in the order of their spoolid in our implementation.
In [A89] it is shown that asynchoronous communication can be implemented in POOL2 using bodies and synchronous communication. For every message that is to be sent asynchronously, a buffer object is created. In such a case, buffering of messages sent but not received is a problem. The problem arises when the buffer is full. In such a case, the sender process must be blocked in order to prevent it from sending more messages unless there is enough space in the buffer to hold them. This will lead to a deadlock in programs which are semantically correct.
This problem does not occur in Distributed Parallel Object Manager implementation because of the following: When a ParallelObject is created, the local process which requests an environment from the remote site is blocked. When the requested environment is created successfuly at the remote site, the local process is notified with the descriptor of the created environment. From then on, the execution of the local process continues. That is, synchronisation is done at local object and remote class at hand shaking time. Messages sent to ParallelObject are buffered at the remote site. The size of the buffer is not fixed. It can grow out to resource limits, and hence no blocking of the local process is required when sending a remote message. Also, the messages received are processed by the order they are sent.
Another problem of the asynchronous approach is what to do when the sender gets too far ahead of the receiver. In the Distributed Parallel Object Manager implementation, this problem is prevented as follows: If at any time a result is required, its state can be obtained by sending the waitsFor: aDescriptor message to the ParallelObject which tests the related semaphore to check whether the result has arrived. Therefore, the user has the choice of either to continue the process or to block it explicitely by demanding the result of the parallel operation by sending the valueOf: aDescriptor message to the ParallelObject. By demanding the result of the parallel operation explicitely, almost absolute synchronisation of the sender and the receiver can be obtained.
Thus although the Distributed and Parallel Object Manager model is asynchronous, it has some exceptions from the pure asynchronous model such as synchronous hand shaking, valueOf: and waitsFor: methods.
The synchronous communication model is not appropriate for object oriented environments like Smalltalk with inheritance and class variables. Because in the synchronous communication model object bodies are always active. Thus when more than one object is active, each may change the class variable defined for the class the objects belong to. Consider the following application: The class Employee defines the behaviour of employees and one of the class variable defines the minimum wage for all employees.This class variable can only be updated when no Employee object is active which is against the philosophy of synchronous communication. Thus if a traditional object oriented environment like Smalltalk is used with a synchronous communication model, it is not possible to guarantee the correctness.

VII. Performance

In order test the design, the implementation is realized on two 386 PC’s serially connected through RS232 serial ports, transmission rate being 1200 baud. The performance of the system is tested for two example cases. It should be noted that this transmission rate is extremely low compared to an Ethernet environment where transmission rate is 10 million baud.
Case 1:
The first case is chosen from the database field and two experiments are performed. In this case there are two relations, namely the Customer and the Order relations.There are a total of 2000 records in the Customer relation and 4000 Order records. In the first experiment a join of these two relations is performed at site 1. In the second experiment, the relations are horizontally distributed to two sites. Site 1 contains the customers with customer numbers less than or equal to 1000, and site 2 contains customers with customer numbers greater than 1000. Order relation is fragmented according to derived horizontal fragmentation, that is, orders belonging to customers with customer numbers less than or equal to 1000 is on site 1, and orders belonging to customers with customer

numbers greater than 1000 is on site 2. The join is executed at two different sites by the Distributed Parallel Object Manager with the hash partition join technique and the final result is obtained by taking the union of the partial results. The selectivity of the join operation is assumed to be .50, that is 50 % of the Customer records have corresponding Order records.
Elapsed times for the two experiments are as follows:
. Experiment 1 : 193 seconds
. Experiment 2 : 147 seconds + 795 seconds for the communication cost
The high communication cost stems from the fact that the rate of communication in our system is 1200 baud. In a 10 million baud per second network, the expected communication cost is calculated to be .02 secs.
Case 2:
The second case is chosen to be computation bound. Here the method evaluate: evaluates an expression 100,000 times and it is called four times.

```
evaluate: aVal
 | a |
 a := 0.
 1 to: 100000 by: 1 do: [ :i |
 a := aVal + aVal*3 - aVal*2 ].
 ^ a
```

We have performed three experiments. In the first one this method is evaluated at the local Smalltalk host. In the second experiment evaluate: method is executed by using the valueOf: message, thus the result of the message is immediately required hence forcing the remote operation model. In the third experiment, two evaluate: messages are executed at site 1 and two evaluate: messages are executed at site 2 by utilizing the Distributed Parallel Object Manager.
For each of the experiments, the elapsed time of execution is recorded. The programs for the experiments are as follows (since class compatibility is assumed, class Test and instance method evaluate: exists both at local and remote host) :

Program for experiment 1 :

```
| t1 t2 a b c d e f |
t1 := Time totalSeconds.
a := Test new.
b := a evaluate: 3.
c := a evaluate: 3.
d := a evaluate: 3.
e := a evaluate: 3.
f := b + c + d + e.
t2 := Time totalSeconds.
^ t2 - t1
```

Program for experiment 2 :

```
| t1 t2 a b c d e f g |
t1 := Time totalSeconds.
t1 := Time totalSeconds.
t1 := Time totalSeconds.
b := ParallelObject new for: Test.
c := b valueOf: ( b evaluate: 3 ).
d := b valueOf: ( b evaluate: 3 ).
e := b valueOf: ( b evaluate: 3 ).
f := b valueOf: ( b evaluate: 3 ).
g := c + d + e + f.
b reset.
t2 := Time totalSeconds.
^ t2 - t1
```

Program for experiment 3 :

```
| t1 t2 a b c d e f g |
t1 := Time totalSeconds.
a := Test new.
b := ParallelObject new for: Test.
c := b evaluate: 3.
d := a evaluate: 3.
e := b evaluate: 3.
f := a evaluate: 3.
g := (b valueOf: c) + d + (b valueOf: e) + f.
b reset.
t2 := Time totalSeconds.
^ t2 - t1
```

Elapsed times for the three experiments are as follows: Experiment 1 : 25 seconds, Experiment 2 : 28 seconds, Experiment 3 : 15 seconds
It is clear that parallel execution for these example cases resulted in high performance gains. These gains are especially high for computation bound processes because the rate of overhead associated with task switching related with computation bound processes is low.

VIII. Conclusions

Distributed versions of Smalltalk are available in the literature. These implementations use a remote operational model. In order to exploit the inherent parallelism in distributed systems through an object-oriented language, a Distributed Parallel Object manager is implemented. Parallelism in execution resulted in expected performance gains.
Parallelism is achieved by using a variation of asynchronous communication and by defining a new class, called ParallelObject class under Object class in Smalltalk's class hierarchy. Parallelism is a property of objects; not

their classes. For parallel objects, blocking is not desired between the task of sending a message and receiving its result. Thus when a message is sent to an object, the value returned is a pointer to the result area of the object's message response. The problems of asynchronous communication do not exist in the Distributed Parallel Object Manager for Smalltalk.

The system supports parallelism with the following main advantages:

1. One of the implementation details is the use of a message queue for each ParallelObject environment. Since all the environments created are suspended with a ready state and activated whenever a message is received for it , the switching of processes is minimized. Therefore, the overall Smalltalk system efficiency is not reduced.
2. The result of any message sent to a ParallelObject can be obtained at any time. The result of a message sent to a ParallelObject, and acquiring the value itself are independent tasks. Furthermore the result of a message sent to a ParallelObject is stored in the system memory. It is not destroyed upon acquiring the value returned. Unless the programmer intentionally destroys the result field, the same result can be used at any time without sending a new message which would be an identical one which had produced the result on hand.
3. For a programmer it is possible to check whether a value is returned for a message sent to a ParallelObject executed at a remote host. Therefore, to block or to continue the process is the programmer's own responsibility.
4. All of the operations carried out for the distributed parallel object management are transparent to all users. All user operations, including the user at the remote host, is not affected while the system serves for a remote object.
5. In Distributed Parallel Smalltalk Object manager a user may establish a session between a local object and a remote class. Thus it is possible to keep the most recent status of objects for a specific application.

References

[A86] America, P., "Definition of the programming language POOL-T", ESPRIT Project Report, October 1986.

[A89] America, P., "Issues in the Design of a Parallel Object- Oriented Language", Formal Aspects of Computing, Vol.1, pp.366- 411, 1989.

[AL90] America, P. and van der Linden, F.. " A Parallel Object- Oriented Language with Inheritance and Subtyping ", ECOOP/OOPSLA'90 Proceedings,pp. 161-168, October 1990.

[B87] Bennett, J. K., " The Design and Implementation of Distributed Smalltalk", in Proc. of the Second ACM Object- Oriented Programming Systems, Languages, and Applications, Oct. 1987.

[D89] Decouchant, D., " A Distributed Object Manager for the Smalltalk-80 System", Object-Oriented Concepts, Databases and Applications", ACM Press, 1989.

[GOLD83] Goldberg, A. and Robson, D.. Smalltalk-80, The Language and its Implementation. Addison-Wesley Publishing Company, 1983.

[v 87] van des Bos, J. and Laffra C., " PROCOL : A Parallel Object Language with Protocols", in Proc. of the Fourth ACM Object-Oriented Programming Systems, Languages, and Applications, Oct. 1989.