Concurrent Programming with Shared Objects in Networked Environments*

C. L. Hartley and V. S. Sunderam

Department of Math and Computer Science
Emory University, Atlanta, GA 30322

Abstract

Concurrent and distributed computing, using portable software systems or environments on general purpose networked computing platforms, has recently gained widespread attention. Many such systems have been developed, and several are in production use. This project proposes the use of object-oriented techniques to enhance application development and ease of use, and to relieve developers of the complexities inherent in message passing environments. We are pragmatic in our objectives; we wish to support the relatively well understood shared-object concurrent computation model while providing facilities designed to aid the programmer with partitioning, scheduling, and synchronization in a straightforward, efficient, and portable manner. In this paper, we describe a shared object toolkit for the PVM distributed computing system, and present preliminary results and experiences.

1 Introduction

Concurrent and distributed computing on loosely coupled systems has gained tremendous popularity in recent years, and several such software systems that support this model are in widespread use. Two important factors contributing to their success are cost effectiveness (as compared to traditional hardware multiprocessors), and high usability as a result of widely portable implementations, support tools, and straightforward interfaces. Examples of such systems are Isis [1], Linda [2], and PVM [3]. It should be noted that these systems, referred to hereafter as "environments", are fundamentally different from distributed operating systems such as Amoeba [4], Chorus [5], and Clouds [6], which are native kernel level implementations. Distributed environments have the distinct advantage of requiring no experimental language, operating system, or hardware.

Distributed environments offer several programming interfaces, but with the exception of a few (notably Linda), they are uniformly based on some variant of message passing. While the message passing model of distributed or concurrent computing is well understood, its complexity of use increases rapidly with the complexity of communication and synchronization patterns within the application. Shared variable programming is believed to be more suitable for such applications, and this school of thought has led to the development of "distributed shared memory" systems that have met with mixed success. However, the techniques that these systems require to provide DSM rely on hardware such as memory management units. For this reason, DSM has fallen within the domain of distributed operating systems (e.g. Clouds, Chorus, Commandos [7], IVY [8]). As a result these systems are often restrictive and non-portable, thereby diminishing their potential widespread adoption.

From a different perspective, object oriented programming has proved to be a worthy software engineering approach. Languages such as C++ provide features such as operator overloading that can be used to transparently model specialized hardware, thus placing distributed shared memory within the reach of distributed environments. The object oriented algorithms are also easily ported to different systems because they rely on software models instead of specific pieces of hardware.

While distributed shared memory may be an alternative to the complexities of the message passing paradigm, it is by no means a complete solution. These systems still require considerable programmer overhead associated with the management of shared variables. Furthermore, most systems provide little or no support for high level issues that arise in the implementation of distributed or concurrent applications.

*This work was supported by the Applied Mathematical Sciences program, Office of Basic Energy Sciences, U. S. Department of Energy, under Grant No. DE-FG05-91ER25105.

In particular, partitioning, scheduling, and synchronization must be handled entirely within the application domain; it is well known that these tasks often involve greater complexity than the original problem.

The primary goal of this project is to allow programmers to construct concurrent programs that efficiently utilize distributed resources, but transparently abstract details of network communication. This paper discusses our work in designing and implementing support for shared object programming in concurrent environments. The system is layered above PVM, a message based concurrent programming system. As a result our project retains the portability, graphical tools, and other facilities for development and profiling that are inherent in the PVM system. The DoPVM system described herein supports the C++ language, permits objects to be shared across distributed components of applications, and provides several mechanisms for assisting concurrent scheduling and partitioning. The latter, in particular, enables performance at levels comparable to native message passing systems, without its inherent complexity. This paper describes important system design issues and our approach to their solution, and presents preliminary results and experiences from our use of the system.

2 The DoPVM User Interface

The DoPVM system (Distributed Object Parallel Virtual Machine) is an object oriented distributed environment, manifested as user interface macros and libraries which, in conjunction with run time support software, enables distributed computing based on shared objects. Application processes share data by enrolling in a DoPVM *session.* Each session maintains a set of distributed objects that is visible to all enrolled processes. The distributed objects which are maintained by the DoPVM system (termed *system objects*) reside in a logical space known as an *object universe.* There is a universal non-negative integer ID associated with each of the system objects. Application processes access system objects by binding local data structures known as *object shells* to them. Once bound, all accesses to the object shell are in fact references to the system object to which it is attached.

The DoPVM User interface can be characterized as a collection of distributed objects grouped into two categories, *viz.* data and tools.

2.1 Data

The basis for C++ programming using the DoPVM system is a set of system defined primitive object classes *viz* integers, single and double-precision real numbers, and characters. In addition, a base class termed `shared` is provided. Using these primitives, and by inheriting attributes of the DoPVM class `shared`, the programmer may build custom data objects that are shared among concurrently executing components of distributed applications. This facility allows users to create arbitrarily complex shared data structures suited to their application. Object construction also permits application control over the granularity of sharing; by constructing objects or aggregates of appropriate granularity, applications can limit the overheads of emulating shared address space contexts over loosely coupled networks.

```
#include "DoPVM.h"

class myclass: public shared    //inheriting attributes from
   {                            //system defined class shared
   public:

   pvm_int      x[100];         //constructing distributed data
   pvm_char     y[200];         //object using the system
                                //defined data primitives
   ...
   };

myclass obj;                    //declaring object shell

main()
   {
   ...
   begin_do();                  //enrolling in a DoPVM session

   obj.set_id(ID_1);            //binding shell to the system
                                //object whose ID is ID_1

   obj.x[50]=253;               //referencing the shared object
   obj.y[150]='a';              //in a straightforward manner
                                //that hides network traffic

   obj.set_id(ID_2);            //Binding shell to another sys-
                                //tem object without loss of
                                //data to object with ID_1

   ...
   }
```

Figure 1: Object Definition, Declaration, Referencing

Each distributed data object is associated with a unique system wide ID. A user creates an object shell by defining and declaring a classtype using the method described above. This shell may then be bound to any of the system objects of the same type by setting the shell's ID. If an object shell is bound to an ID for which there is no system object, a new distributed object is created with the specified ID. The object may now be referenced as any other(non-shared) local variable. From the programmer's point of view, only a single instance exists for each shared object. Accesses and updates refer to a single logical copy, and usual consistency semantics apply. Informally, concurrent updates are serialized in unspecified order, and read accesses will retrieve latest available values — but tighter synchronization may be achieved through the use of explicit locking mechanisms that are provided. In addition, shell bindings may be dynamically changed, i.e. shells may be bound to different objects at various stages in a program's execution. Thus a programmer may either choose to declare multiple object shells to access corresponding distributed objects, or may use a single shell whose ID may be bound to different objects as needed. An underlying object daemon maintains data consistency and implements the mechanics of binding as well as consistency preservation as objects change their IDs. Figure 1 illustrates an example of the distributed object interface, showing class definitions, shell declarations, and binding.

In the DoPVM system, several optional constructs are provided that can reduce the overheads associated with sharing objects across a distributed environments. Within the data category are the *lock* member functions. When a process desires to utilize a data object for extended periods of time, locking the object causes other accesses to the object to be deferred, thus avoiding potentially inefficient, interleaved sharing. A programmer may chose to prevent write access to the object by using the basic *lock* member function. When an object is locked in this manner, other processes may still read the object. Read accesses can also be blocked using the *rlock* function. This a more stringent lock status that gives the owner complete control over the object. Processes that attempt to access locked objects simply block until the object is *unlocked* by the locking process.

To utilize the shared object facility, users need only include a special header file, and enroll in a DoPVM session by invoking the `begin_do()` library routine prior to referencing any shared object. Shared data objects that are created are guaranteed to be maintained until the end of the session, which is indicated by the exit of all its enrolled members. Objects may be shared among processes derived from different source code, i.e. different sub-algorithms of a distributed application may interact via shared objects — binding using the same identifier equivalences objects from unrelated processes in a session. This feature enables modular design of distributed applications, since users may write separate programs for each subtask of the application. The DoPVM system views executables as a special kind of distributed tool.

An important feature of the DoPVM system is that it allows distributed sharing of data objects between architectures with incompatible data representations. The DoPVM system utilizes the data marshaling capabilities of the underlying PVM infrastructure to ensure data correctness on varied hardware. All data representation issues are handled transparently. Thus a programmer may construct arbitrarily complex data objects without being concerned with inter-machine inconsistencies.

2.2 Distributed Tools

To avail of the distributed object facilities, no further program modifications are necessary. However, as with most distributed systems, significant performance gains may be obtained in exchange for small additional effort. As in the data section, there are also several optional constructs designed to further reduce system overhead associated with managing distributed objects. An example mechanism is the aggregate data object. This is a special tool that allows programmers to bind several unrelated objects into a single object that is procured as an atomic unit. Aggregate objects may be locked as any other distributed object and deadlock avoidance mechanisms are built in to prevent attempted simultaneous procurement of intersecting sets of objects. The DoPVM system also provides a mechanism for proactive transfers of objects. When the application knows object access patterns *a priori*, this feature can be used to position objects at the locations they are needed, thus reducing overheads.

The DoPVM system also provides additional facilities that are valuable in the management of distributed and concurrent computations. These include mechanisms for scheduling, problem partitioning, and support for implementing function and agenda-parallelism. An example is a special distributed queue system into which a parallel application may place pending tasks that are to be performed by the next free process. Several queueing disciplines are provided; as processes become available, this system object may be accessed to retrieve more work in FIFO or LIFO order. Distributed data objects may be placed into different queues based on their characteristics, e.g. compute-intensive, I/O intensive, graphics oriented, etc. With the assistance of the underlying PVM system, the shared object facility allows tasks to be positioned on the machine architecture that is best suited to the task requirements. While these features may be implemented within the domain of the application, they are more effective and less cumbersome when integrated into the DoPVM system.

2.2.1 The Distributed Queue Interface

There are two built-in C++ queue class definitions in the DoPVM system: `fqueue` and `lqueue`, implementing FIFO and LIFO disciplines respectively. In order to utilize a distributed queue one simply declares an instance of one of these two class types to create a queue object shell. Programs then specify a system-wide ID, as with distributed data objects. This ID may be reset during the course of the program, thereby allowing one queue object shell to change its distributed queue binding many times during the execution of an application. The DoPVM object daemons maintain consistency of queue information. After issuing the `begin_do()` function call applications may insert and retrieve objects from the queue. There is no limit to the number of distributed queues, nor any restrictions on the placement of the queues (that is, a queue may be shared by any subset of active nodes in a session).

The insertion operation for queues has been designed for maximum ease. One may insert either integers representing data object IDs or objects themselves. When a user performs the `<<` operation on the queue, the object (or the object represented by its integer ID) is inserted into the queue. Once an object has been inserted into a queue it is still readable and writable, i.e., normal object operations may continue to be performed, and the usual consistency semantics apply. In addition, processes may query the object for its queue status information, and a special status flag is be used to indicate when a particular queue is "closed", i.e., when it will no longer accept further object insertion.

Object retrieval is similar to insertion. Again one may retrieve either integer IDs representing objects, or object shells themselves. When a user performs the `>>` operator on a queue and an object shell, the queue will "load" the object shell with the next available object. If the object shell had contained an object before the retrieval, the object daemon will buffer the old object. If an object with the new ID does not yet exist, the object daemon will instantiate an object with the new ID. This new object will persist until the session terminates. The queue object's defining functions ensure that locking and residency status for the object being retrieved will be assigned to the requestor. The program may find the ID of the loaded object by using the built-in `get_id()` member function. When the `>>` operator is used with an integer variable, it will return the integer ID of the next available object. Both types of retrievals will block on an empty queue. If the queue returns the special end of input integer, it will set the object shell's ID to a special flag value.

Though the DoPVM queues can be used to implement the message passing paradigm, one should not confuse the queue communication system with traditional message passing distributed environments. When a process inserts an object into a queue, it does not indicate its destination. All data marshalling is handled implicitly by the under lying object daemons. The queue model also allows queued objects to be inspected and changed while still in the queue. Thus monitoring applications can be easily written to inspect the state of the data objects without regard to their physical location in the distributed system. A schematic of the DoPVM queue mechanism is shown in Figure 2, while an example illustrating the use of queues is shown in the appendix.

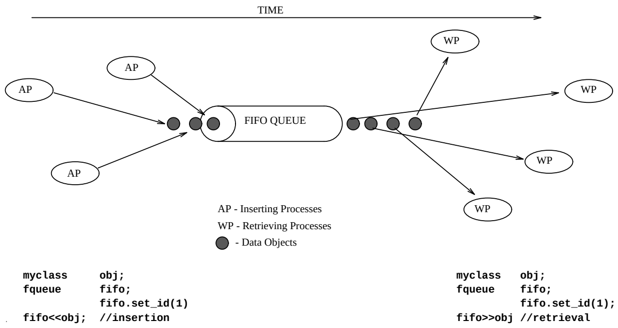

Figure 2: Distributed FIFO Queue Model

3 Design Aspects and Implementation Strategies

The PVM system is the substrate on which the shared object facility executes. Details on PVM may be found in [3]; briefly, the system consists of a collection of daemons that execute on a user-defined host pool, and cooperate to emulate a virtual concurrent machine. Applications interface to this virtual machine via constructs for process management, communication, and synchronization, that are provided in the form of user level libraries. The DoPVM system consists of header files and user level libraries that provide the user interface, and an additional daemon per process to support the shared object facilities. Both the application process and the special daemon operate as PVM user processes. Physical shared memory and pipes are used between the DoPVM daemon and applications processes that use the distributed object facility. The physical shared memory region contains storage allocated to hold objects that are being shared concurrently, while the pipes are used for asynchronous control message communication. An architectural overview of the system is shown in Figure 3.

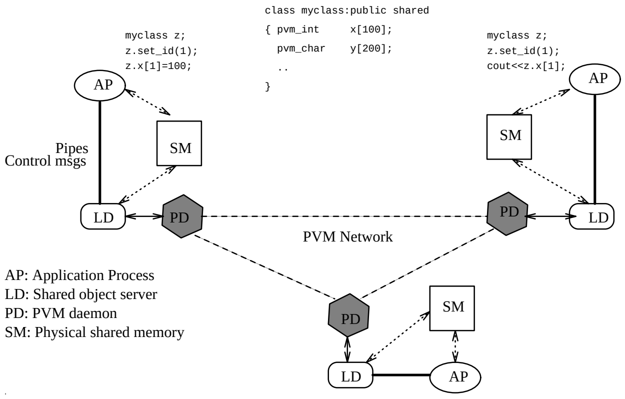

Figure 3: Architectural Overview of DoPVM System

When the application process executes, all object declarations that are based on DoPVM classes invoke special constructor functions that allocate these objects in traditional UNIX shared memory. In addition, supporting data structures are set up to permit efficient association between external requests for updates and accesses, and the local data contained in an object. The `begin_do()` library invocation that is required of all DoPVM applications performs certain initialization tasks and initiates the shared object daemon by forking. The server, which has retained the same address space as the application, continually listens for and manages requests from the remote servers (via the PVM system), and from the local application process.

Data consistency is maintained through a variation of the system described in [8]. In our scheme, the data structures known as object shells possess local storage within the application. Each object shell shares with its object daemon a set of flags that indicate its distributed object's "residency" status. These flags dictate the level of access that an application is given to the object. This access level is monitored as access requests are made by the process. There exist three levels of residency. *Non-residency* denotes to the process and its object daemon that the data in the object shell is invalid and needs updating. *Read-only* tells the processes that the object shell may be read without any network traffic. Finally *Read-write* residency allows the application to read and write the distributed object. There is an additional flag that indicates *ownership* of the object. A distributed object may not have more than one owner at a time. However, the object may change owners during the course of the DoPVM session.

When an application tries to access an object for which it does not have the correct residency status, a trap is made into the object daemon. The application process instructs its object daemon to obtain the proper residency status. The object daemon then sends a request via PVM for an update of the system object to its owner. If the request is for a read copy, the owner checks the object shell's lock status. If the object is read-locked the request is placed into a queue until the object is unlocked. Otherwise, the owner's object daemon checks the object shell's residency status. If the object has read-write residency, the object daemon demotes it to read-only. A read copy of the object is made and sent to the requester. The requesting object daemon then updates its object shell with read-only residency and returns control to the application process. If the request is for write access, the owning object must invalidate all read-only copies of the object and then send the object and ownership of that object to the requester. In this scheme the distributed object may reside on multiple nodes with read-only status or a single node with read-write status. Residency checking is completely transparent to the user. The access interface to data objects is implemented through operator overloading, e.g., the `=` operator is overloaded to check all residency flags and, if necessary, send request messages via a pipe to its object daemon. Operations with the DoPVM data types are designed to function exactly like their built in C++ analogs.

An object daemon may remove the residency of an unlocked object at any time. In order to maintain data consistency, we have developed a fast scheme for distributed mutual exclusion. We have implemented a simple two process shared memory locking scheme that is described in [9]. For every unlocked write, the application process performs a `p()` operation on the object, performs the write, and then releases the object by executing a corresponding `v()` operation. The object daemon is also required to use the `p()` and `v()` operations for residency removal. If an object is read locked, all references to it bypass this mutual exclusion code; therefore locking is desirable. Preliminary tests have shown that locked references are only 20% slower than a basic C++ datatype reference while unlocked references may be slower by as much as 75%.

The queue models are currently implemented as central stores of information owned by one object daemon. Each object daemon knows the location of the owners of all the queues with which its parent application process interacts. When an insertion is made into a queue, the inserting object daemon provides the queue owner with the object ID and a hint about its physical location. Upon retrieval, the queue owner will instruct the owner of the object to send ownership privileges to the retrieving process regardless of its current state. Before the new object is loaded, the object daemon invokes the `set_id()` member function for that object shell. This function checks to determine if there is an object resident in the shell. If there is, it must use the C++ `new` operator to buffer the object information. Upon the next reference of the buffered object, the object daemon will give ownership to the referencer and `delete` the space allocated for the buffered object. Efforts are currently being directed toward replicating the queue object in order to increase fault tolerance.

4 Programming Models

The distributed object model can be used to implement several traditional parallel models while relieving the programmer of the burden of communications complexity. This section discusses some traditional parallel models and our preliminary experiences with their implementation.

4.1 Shared Memory

The shared memory model is widely believed to be an effective alternative to the more complex message passing paradigm. DoPVM distributed objects provide this paradigm directly. The basic DoPVM strategy requires the user to fragment application data into partitions represented by data objects. The size of the objects dictate the granularity of sharing. Once an object has been declared and has been bound to a system ID, it can be accessed as any other local variable. The object becomes visible to all object shells that are attached to the ID. The DoPVM system effectively hides a very complex system of PVM message passing through C++ operation overloading.

We have implemented a generic rayshade application to test this model. We divided the canvas into a number of subcanvases to distribute the work. One program was written to actually calculate the image onto the subcanvas objects. Each instance of this program received an equal amount of objects to process. Another program was written to draw the results of these calculations. Results are discussed and compared in the Preliminary Experiences section.

4.2 Bag of Tasks Applications

The DoPVM shared variable model completely hides all details of distribution and network communication traffic from the user. However, there is still considerable synchronization, scheduling, and partitioning overhead involved. In applications in which work is not partitioned symmetrically, the user must maintain shared data structures to keep track of work that has been completed and work that is still pending. Processes that depend on the computation of others must be synchronized. The DoPVM system uses the queue model to implement worker parallelism [10]. In this scheme, some set of processes place work into a queue for some other set to process. This model has several important advantages. Processes are self-scheduling because they are able to acquire tasks as needed. This allows more efficient use of network resources, in the form of dynamic load-balancing. Processors that are lightly loaded or more powerful may process many more objects than slower processors. Computing elements are only idle when there is no work to be done. Processes are also guaranteed to be working on a unique object, thus reducing synchronization complexity.

In order to compare the results of the shared memory model and the Bag of Tasks model, we implemented another rayshade application and processed the same image. Again we divided the canvas into subcanvas objects. We also used two programs to calculate and to draw, but this time we incorporated two FIFO queues. The drawer loaded one of the queues with integers that were to be the projected IDs of the sub canvas objects. This process then retrieved objects from another queue that held the processed objects. The calculating processes retrieved objects from the first queue, processed them, and then placed them into the second queue. Note that the drawer only needs to insert integers into the queue. The system automatically instantiates an object with that ID number. This object remains available for referencing until the session ends. Sample code outlining this partitioning/scheduling scheme is shown in the Appendix.

4.3 Macro Pipelining

The Macro Pipeline model can be characterized as number of dedicated processors that are linked together in such a way that information flows in one direction. Each group of processors along the pipeline is assigned a certain segment of the problem that depends on data from the computation of the previous processor group [11]. Computer vision is a classic example of an application category that requires and benefits from this model. Some stages of the problem exhibit a high degree of parallelism that can be distributed to generic workstations, and others are less regular but require special hardware.

Pipelining is a natural extension of the Bag of Tasks model discussed above. The implementation can be visualized as having some number of stages, each connected by a distributed queue. DoPVM allows the programmer to position worker processes on the hardware that is best suited for that stage of the application. This is achieved through the use of the PVM intrinsic `initiate` that allows architecture or machine based positioning of application component instances. At each stage of the problem, there is a queue from which workers dedicated to one stage of the problem can retrieve objects and another into which they can place objects for the next stage. Thus, each problem segment can be parallelized to the degree allowed by the nature of that stage of the problem and the available hardware, while utilizing the self-scheduling and load balancing advantages of the bag of tasks model.

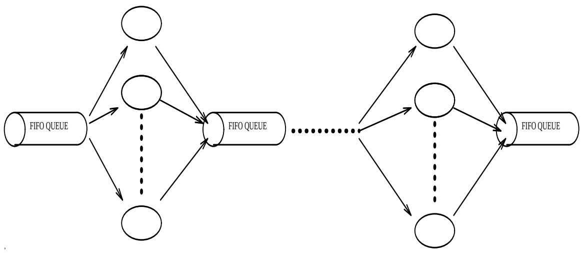

Figure 4: Distributed Processes in a Macro Pipeline

5 Preliminary Experiences and Discussion

At the time of writing, we have implemented a proof-of-concept version of the DoPVM system. The user interface is fully functional, in that shared objects of arbitrary complexity may be declared and used within C++ programs. Several test applications have been written utilizing the DoPVM system. Preliminary performance measurements from a representative application are shown in Table 1. The first two sets of measurements are elapsed times in seconds for a simple symmetric implementation of a generic ray shade algorithm. The application renders an image whose subproblems vary a great deal in complexity. While the speedups achieved are mediocre (owing primarily to imbalanced workloads), it is interesting to note that the shared object model implementation performs as well as an identical message passing version. The third set of measurements show the same rayshade algorithm using FIFO queues for scheduling and partitioning. Using the same set of machines we see dramatic improvements in the elapsed times. The Bag of Tasks model allowed the machines that received the trivial sections of the rayshade to process many more objects. In a less controlled experiment, we added some faster processors to the virtual machine configuration. While they added trivial performance gains to the first two models, we saw improvement of as much as 30% in the two and four processor categories when based on the FIFO mechanism.

# Procs	MP Rayshade	SO Rayshade	FIFO Rayshade
1	642	642	642
2	411	410	324
4	267	282	166
8	175	177	105
16	112	125	95

Table 1: DoPVM vs. Message Passing

DoPVM is an auxiliary toolkit for the PVM system that enables distributed object programming in C++ over a heterogeneous collection of computer systems. The PVM system is used as the communications substrate, and shared object programming is supported in a straightforward and efficient manner. The primary strengths of our system are the provision of a standard, object based, programming paradigm — one which is generally accepted as beneficial in concurrent software development. By providing several tools to aid in partitioning and scheduling in addition to a fully functional language interface, the system attempts to be directly utilizable for production quality concurrent and distributed applications. Our initial experiences have demonstrated encouraging performance. Tests using the bare object model have shown that efficiency is not significantly compromised in comparison to equivalent message passing implementations. The distributed tools provided with the DoPVM system have also proven to increase speedup without increasing the complexity of the application program. Based on our initial findings and the flexibility and power of the user interface, we feel that the DoPVM system is a viable environment for the development of distributed applications for multi-platform networks.

References

[1] K. Birman and K. Marzullo, "ISIS and the META project" *Sun Technology* , pp. 90-94, Summer 1989.

[2] M. Arango, D. Berndt, N. Carriero, D. Gelernter, and D. Gilmore, "Adventures with Network

Linda”, *Supercomputing Review*, Vol. 3 No. 10, Oct. 1990.

[3] V. Sunderam, “PVM: A Framework for Parallel Distributed Computing”, *Concurrency: Practice and Experience* Vol. 2 No. 4, Dec. 1990.

[4] Sape J. Mullender, “The Amoeba Distributed Operating System Part 1”, *Centrum voor Wiskunde en Informatica Newsletter*, No. 11, June 1986.

[5] F. Armand, M. Gien, F. Herrmann, and M. Rozier, “Distributing UNIX Brings it Back to its Original Virtues” , *Usenix Symposium on Distributed Multiprocessor Systems*, October 1989.

[6] P. Dasgupta, R. LeBlanc, W. Appelbe, “The Clouds Distributed Operating System:Functional Description, Implementation Details and Related Work”, *International Conference on Distributed Computing Systems IEEE*,1988.

[7] D. Decouchant, M. Riveill, C. Horn, and E. Finn, “Experience with Implementing and Using an Object-Oriented, Distributed System”, *Usenix Symposium on Distributed Multiprocessor Systems*, October 1989.

[8] K. Li and P. Hudak, “Memory Coherence in Shared Virtual Memory Systems”,*ACM Transactions on Computer Systems*, Vol.7 No. 4, November 1989.

[9] J. Peterson and A. Silberschatz, *Operating System Concepts*, Addison Wesley, Reading,1991, pp 141-144.

[10] N. Carriero and D. Gelernter, “How to Write Parallel Programs: A Guide to the Perplexed”, *ACM Computing Surveys*, Vol.21 No. 3, September 1989.

[11] S. Williams, *Programming Models for Parallel Systems*, John Wiley and Sons, Chichester, New York, 1990.

Appendix A: Example Coding for Bag Of Tasks Application

```
============================================

———
DRAWER.c
———
#include ”DOPVM.h”
extern ”C” int atoi(char*);

/*class definition.  The next class is an DoPVM shared object class.*/

class sub˙canvas:public shared
  –
  public:

  pvm˙int     red[SUB˙CANVAS˙ROWS][SCREEN˙WIDTH];
  pvm˙int     green[SUB˙CANVAS˙ROWS][SCREEN˙WIDTH];
  pvm˙int     blue[SUB˙CANVAS˙ROWS][SCREEN˙WIDTH];
  ˝;

main(int,char**argv)
 –
 fqueue      in;
           in.set˙id(1); //fqueue declaration
 fqueue      out;
           out.set˙id(2);
 sub˙canvas   chunk; //object shell declaration
 int          i,j,k,id;

 begin˙do();

 for (i=0;i<atoi(argv[1]);i++) //argv[1] is the number of
   initiate(”worker”,”SUN4”); //worker processes.  initiate
                               //is a standard PVM call to
                               //start a process

 for(i=0;i<NUM˙SUB˙CANVASES;i++) //here we are loading the queue
   in<<i;                        //with integers. These will be
                            //extracted and made into objects
                            //by the workers

 in<<-2;                    //marking end of queue

 for(i=NUM˙SUB˙CANVASES-1;i>=0;i–)
   –
   out>>chunk;             //here we are ”loading” the
                           //object shell with an object
                           //from the out fifo queue; the
                           //old object will be buffered for
                           //future references to that object

   id=chunk.get˙id();

   /*based on id chose the section of the screen to fill*/

   for(j=0;j<SUB˙CANVAS˙ROWS;j++)
     for(k=0;k<SCREEN˙WIDTH;k++)
       –
       CURRENT˙PIXEL.r=chunk.red[j][k];
       CURRENT˙PIXEL.g=chunk.green[j][k];
       CURRENT˙PIXEL.b=chunk.blue[j][k];
       ˝
   ˝
 ˝

———
WORKER.c
———

#include ”DOPVM.h”

class row:public shared
  –
  public:

  pvm˙int     red[SUB˙CANVAS˙ROWS][SCREEN˙SIZE];
  pvm˙int     green[SUB˙CANVAS˙ROWS][SCREEN˙SIZE];
  pvm˙int     blue[SUB˙CANVAS˙ROWS][SCREEN˙SIZE];
  ˝;

main()
  –
  sub˙canvas    chunk;
  fqueue        in;
  fqueue        out;
  int           i,j,id;

  in.set˙id(1);
  out.set˙id(2);

  while(in>>chunk,id=chunk.get˙id(),id!=-2)  //retrieving objects
    –                                        //until end of queue
    row=id*NUM˙SUB˙CANVASES;
    for(i=row;i<row+SUB˙CANVAS˙ROWS;i++);
      for(j=0;j<SCREEN˙WIDTH;j++)
        –
        chunk.red[i][j]=RED˙COMPONENT˙OF˙POINT();
        chunk.green[i][j]=GREEN˙COMPONENT˙OF˙POINT();
        chunk.blue[i][j]=BLUE˙COMPONENT˙OF˙POINT();
        ˝
    out<<chunk;                       //placing finished
    ˝                                 //objects in 2nd queue
  ˝
```