1994

Distributed Memo: A Heterogeneously Distributed and Parallel Software Development Environment

William T. O'Connell

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Thomas W. Christopher

Distributed Memo: A Heterogeneously Distributed and Parallel Software Development Environment

William T. O'Connell
AT&T Bell Laboratories
1200 E. Warrenville Rd.
Naperville, IL 60566
wto@uscbu.att.com

George K. Thiruvathukal
R.R. Donnelley and Sons Company
750 Warrenville Road
Lisle, IL 60532
gkt@disney.donnelley.com

Thomas W. Christopher
Illinois Institute of Technology
10 West Federal Street
Chicago, IL 60616
tc@iitmax.acc.iit.edu

Abstract

Heterogeneously distributed and parallel computing environments are highly dependent on hardware, data migration, and protocols. The result is significant difficulty in software reuse, portability across platforms, and an increased overall development effort. The appearance of a shared directory of unordered queues can be provided by integrating heterogeneous computers transparently. This integration provides a conducive environment for parallel and distributed application development, by abstracting the issues of hardware and communication. Object oriented technology is exploited to provide this seamless environment.

Index Terms - Heterogeneous computing. Parallel and distributed processing. Directories of unordered queues. Dynamic Data Migration. Portability. Extensibility.

1 Introduction

Heterogeneous computing (HC) allows parallel and distributed applications to achieve a higher level of performance at a lower cost/performance ratio. HC provides the ability to coordinate a wide range of diverse high-performance machines, each being used for computationally demanding tasks [1]. This provides the ability to differentiate between code, algorithms, and data to optimize the matching of computational tasks to the appropriate machine [2].

However, this leads to difficulty in writing parallel and distributed programs in an easy, efficient, and portable manner. The complexity arises from the differences in specialized hardware, incompatible data domain mappings, communication protocols, operating system interfaces, and other distinguishing characteristics (e.g. number of processors, shared versus distributed memory). This results in considerable effort needed to port to new platforms with little code reuse. As the hardware continues to improve and new architectures introduced, the investment in developing, maintaining, and porting HC code across platforms becomes considerable. Monetary costs are effected with longer development cycles.

In addition to the basic HC problems, emerging technology is *changing* the shape of computing. One trend indicates that many parallel and distributed applications may potentially be a workstation phenomenon rather than a specialized parallel hardware phenomenon. As shown by Eq. (1), processor performance is doubling annually since 1984 [3].

Workstations are typically the first to exploit new processor technology. Additionally, networking speeds are jumping by an order of magnitude on average, every three years [3]. Recent research in networking protocols, such as Distributed Queueing Random Access Protocol (DQRAP), has shown that M/D/1 performance numbers can be achieved over a broadcast channel [13]. This results in a near ideal hardware environment for HC.

$$MIPS \Leftarrow \frac{3}{2} \cdot 2^{year - 1984} \quad (1)$$

The fact is that workstations are economical. Most organizations can not afford specialized machines. If they can, the number is limited. It is now possible to get performance improvements of *substantial* nature with the fast processing and networking capabilities of the average workstation. In addition to the processing power, workstations have large-screen graphical I/O capabilities.

Our focal point is on conventional parallel programming over heterogeneous systems. We use Object-Oriented technology to define the heterogeneous environment of the *Distributed Memo* System (D-Memo). The system provides the following to the application:

- Under-utilized resources on a network exploited.
- Easy-to-use parallelism capabilities.
- Complete transparency for underlying machines.
- Dynamic data migration across HC machines.
- High degree of portability and reusability for both applications and the system itself.
- An environment that can easily implement data-parallel, data-flow, functional, and object oriented languages.

In this paper, we will place greatest emphasis on the philosophy and description of D-Memo. We will describe the four basic foundations that are the **key** to its portability and extensibility. We will then discuss the system's framework, which is how the pieces work together. Finally, we will survey related work and present conclusions.

2 Implementation Philosophy

Many distributed-memory Multiple Instruction, Multiple Data (MIMD) systems allocate one process to each node on the multiprocessor. The nodes send messages directly to each other for communication and synchronization. The problem with such systems is that data structures are not global, but rather localized in each node. One of the most

common programming techniques is to manipulate data structures. But on most distributed-memory systems, the data structures must be partitioned and the parts hidden in a fixed number of processes. This creates both complexity and conceptual problems in the software applications.

This common use of shared data structures can be represented as a shared directory of queues. Both directories and queues have long been known to be useful in multi-programming and multi-tasking environments. This system is based on exploiting this in a HC environment. A virtual machine is provided to the application by transparently using a network. Messages are referred to as *memo*s and queues as *folder*s. Once a process places a memo (message) into a folder (queue), any process can extract it. The combined folders provide the virtual shared directory. This allows any process to either examine, extract, or place memos into them.

The communication scheme allows processes to communicate through memo passing. They are deposited into any one of the shared folders (directory of unordered queues). If a folder does not exist, it is created. The folders provide synchronization and communication between processes. By distributing the folders over the network, a pool of segments is used to create a larger virtual shared segment. This provides the abstraction of executing on a single shared-memory MIMD machine. This high level abstraction removes the underlying concurrent memory access and communication problems of parallel and distributed programming and provides a seamless heterogeneous environment.

The System is designed to support *many* different types of architectures from both Massively Parallel Processing (MPP) machines to workstations. The system has currently been implemented on the following platforms: Sun SPARC 4 workstation, Encore Multimax, and Intel 80486 time-sharing system (System V). Work is currently underway to integrate the IBM SP-1 MPP machine.

We caution the reader that the system is not yet another remote procedure call (RPC) system nor a rehash of the existing work (see "Relation to Other Research" on page 8). Languages we have implemented on top of the API include:

- Message Driven Computing language, a pattern-driven language based on Actors [4].
- Lucid, a dataflow programming language [5].

We have found that these languages are excellent for writing parallel programs (as well as using D-Memo's API itself). Their implementation on top of D-Memo will allow us to attract a larger audience for our systems (and for implementation of their systems) through greater flexibility and portability in a HC environment.

The API provides a rich set of primitives for supporting many synchronization mechanisms and programming paradigms. Examples include named objects, arrays of objects, locks, semaphores, unordered and ordered queues, job jars, futures, incremental structures, and barriers (see "Memo Language (API)" on page 6). The API is influenced by Linda [6], but it has been scaled down in features of dubious value and augmented with features of proven value.

3 Abstracting the HC Environment

Object oriented design (OOD) was chosen for implementation, because it offers several advantages over the commonly used (or abused) structured methodologies. OOD enhances re-use opportunities through class extension (class derivation) and delegation. Since D-Memo is a major systems programming effort, the intent is to simplify migration to new platforms and protocols by abstracting the HC environment.

The OO implementation was aimed at addressing D-Memo's portability and extensibility. In addition, we needed to provide high level abstractions (foundations) for the HC environment. The first two terms are so frequently abused that we will attempt to establish some formalism.

Portability is the ability to migrate a software application to a new platform (or, in distributed computing, to a new set of platforms) with a lucid understanding of what aspects of the application must be replaced with platform-specific code. In D-Memo, platform-specific changes can *always* be traced to a class. As an example, an abstract class called `SharedMemory` exists. Operating systems that support shared memory tend to do it differently. There is some commonality, however, and this commonality is extrapolated into the abstract class `SharedMemory`. For example, on the Encore Multimax, one must specify the maximum amount of shared memory the application intends to use, then allocate and free pieces of it using specially named primitives. Then on termination, it must release the pool of shared memory. System V systems (like the Sun Sparc) manage shared memory in a similar way, although the functions to use shared memory and how they are used differ in a subtle manner. Abstract classes allow shared memory and its conventional use to have a consistent interface, although the actual implementation of each derived `SharedMemory` class may differ between systems.

Extensibility goes hand-in-hand with portability, but it pertains specifically to the design. An extensible design is one that enables portability with little or no change to the existing design. Reverting to the `SharedMemory` class example, if major modifications are required to the **base** abstract class `SharedMemory` to support a new processor, there is obviously a major problem. The OO community is constantly grappling with the issue of how to develop extensible abstractions. In attempting to define a highly general abstraction it is best to get an adequate perspective of how the abstraction is used in general. Clustering of concepts is the mechanism typically used by humans. In the case of shared memory one must study how it is done in numerous operating systems to determine the common protocol. Some systems require the application to specify its dynamic memory requirements before actually doing dynamic memory operations. Other systems do not. The abstract class must be able to cope with both cases

3.1 HC Foundations (Abstractions)

Our approach has been to define and implement four general HC abstractions. The core set is: communication, shared-memory, transferable, and locking. These abstractions are based on the Generic Modelling

Framework [7]. To support a new platform in a HC network one must consider each of these four abstractions. Often it is merely a matter of extending one or two of them to support the new platform. The OO terminology for this idea is class derivation. Each foundation is actually a cluster of related classes built with either one or two layers of inheritance. Through virtual functions, all platform-specific code is selected at run-time. This allows the basic foundations to provide a transparent environment to the upperlying software.

3.1.1 Network Communication Class

The idea of network communication being an abstract class in many respects is not completely new. The OSI model addresses the business of one application interfacing with another application through a layered architecture.

In D-Memo, it is fundamentally important to establish a connection between two processes, located on any two machines or the same machine. This abstraction is known as a Connection. To establish a connection does require the assistance of a `Routing` class, used as a collaborator. The notion of a connection, we contend, is generally useful in the context of two processes that must communicate and can be defined independent of any known networking protocol. The notion of a Connection allows processes in the system to connect to other processes by a logical network address.

To achieve a connection requires the help of other classes. A transport class provides an interface to the transport protocol supported by the host. The class provides the ability to simultaneously interact with different protocols in an application, e.g. TCP, EIU-H, and UDP. However, many of the systems do not provide a transport layer, in which case a transport layer must be derived. INMOS Transputers are a perfect example. No transport layer exists. When one wants to send a message, a channel is opened and the message is sent into it. This, however, results in poor performance. Compute-bound processes that are ready to use the CPU are blocked until the long-winded communication is ended. A derived transport layer that supports packet fragmentation and virtual connections would allow the communication cost to be amortized over time and allow some useful processing to be done in the process.

The Routing class is used to provide routing capabilities over the network. Many considerations are considered to improve network efficiency (see section 5.1).

3.1.2 Shared Memory Class

Shared memory was presented in the beginning of section 3 to help motivate the object oriented principles of portability and extensibility. It has been adequately discussed there, thus will not be discussed further.

3.1.3 Transferable Class

Because architectures today support word sizes of 16, 32, 64, and 128 bits (as well as arbitrary bit-vectors), lossy domain mappings can occur. Similar problems exist for floating point numbers, since different precision representations are common: single-, double-, and extra-precision, for example. A lossy mapping occurs when an Alpha processor (64-bit) sends an integer to an Intel 80486 (16-bit) and the value is greater than 16-bits. The problem is not byte order, but precision. To support lossless data domain mapping between heterogeneous machines, distributed software **must** learn to think in concrete domains. Instead of built-in data types like `int`, `float`, etc., the application must use absolute domains (e.g. `int16`, `uint16`, `int64`, `float32`, etc.).

The transferable classes (e.g. `int16`) define a protocol to encode and decode data structures in a language independent manner. We believe the support for persistent data structures is essential to develop serious parallel software applications, especially for non-numerical algorithms. Each transferable is an active object that will encode arbitrary data structures and scalars for transfer between compatible and incompatible domains. Transferables encode/decode themselves recursively, so that messages may be created from either previously user defined or base transferables.

The inspiration for the transferable classes is found in the Abstract Syntax Notation/1 (ASN.1) [8] and the XDR library supported by Sun RPC. There are two major differences, however, between the mechanisms supported by the transferable classes and these other methods of encoding data. The major difference is that arbitrary data structures, even self-referential structures, can be moved with ease via the transferable classes. Without going into great detail here the basic observation is that all data structures have a spanning tree. A spanning tree can be constructed in polynomial time. Thus, it is possible to encode (linearize) an arbitrary structure and to decode (de-linearize) it in polynomial time. The OSI and RPC systems both require significant programmer intervention to manage the details of encoding and decoding data structures. This should be transparent to all processes.

3.1.4 Locking Class

Similar to the problem of shared memory management, mechanisms for low-level locking tend to vary between platforms. For the case of specialized parallel machines with one or more memory organizations, different locking mechanisms may be present. Our experience with the Encore and Sequent machines is a testament to the number of available options. While there are attempts to standardize the locking mechanisms (e.g., POSIX), there will undoubtedly always be systems that expand on the capabilities provided by the standardized mechanism. In the case of the machines mentioned, there are times when it is a good idea not to use a semaphore and opt for a more efficient locking mechanism.

4 Application Frameworks

The frameworks describe the layout of the major components of the system. The servers will be described first, followed by system partitioning methods, application description files, and the registration process.

4.1 Distributed Memo System Servers

Two types of servers are used, both using the four basic

abstractions to hide the details of the underlying platform. This allows the servers to remain portable between heterogeneous machines and improves software reusability and extensibility for the system.

The two server types are the memo and folder servers. The folder servers maintain a directory of unordered queues on selected hosts (each queue representing a folder). There can be 0, 1, or more folder servers per machine, each having exclusive access to its folders. The memo servers are responsible for message routing between processes (there is one memo server per machine).

Figure 1 Intra-Machine Server Behavior

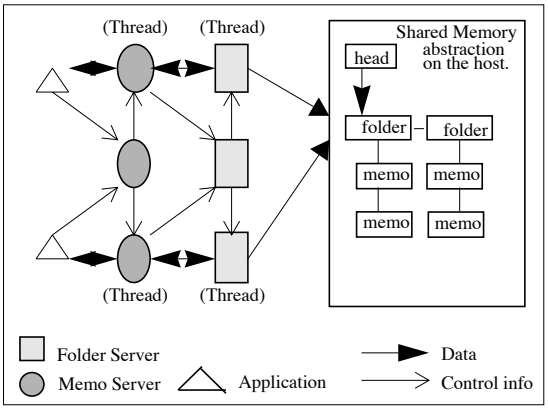
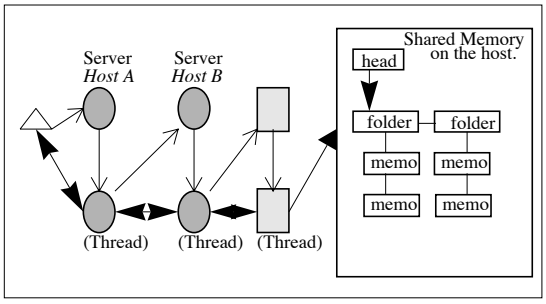


Figure 1 illustrates two application processes and one folder and memo server on a machine. It shows the communication between the different processes along with the shared memory abstraction. Note that the servers use threading to increase concurrency. As an application attempts to deposit/retrieve memos to/from a given folder, that folder name is hashed to a folder server on a particular machine. All references for memos in a particular folder will be directed to the appropriate folder server. Each request to a server will cause a thread to be created to handle the request, thus exploiting parallelism. The system uses the idea of thread caching to avoid the overhead of creating processes un-necessarily. When a thread completes its transactions, it will set a timer and wait for additional requests. If a request comes in, the thread will handle it. If not, it will terminate.

Figure 2 Inter-Machine Server Behavior



If a folder name is hashed to a folder server that does not already contain a folder entry for any memos, a folder will be created. When hashing the folder name to a particular server, the costs associated with the machines' processor(s) speed and communication links are considered. The idea is to optimize the message traffic according to processor and network link speeds (see section 4.3). Figure 2 illustrates an inter-machine communication transaction where a request is made from a process on host A to host B.

Each memo server (one per machine) listens for connection requests from either other memo servers (inter-machine traffic) or user applications. As requests arrive, the server will create a thread (if no cached thread is available) to handle the request while it goes back to listening for more requests. A path is then established between an application program and a folder server via one or more memo server threads (as shown in figures 1 and 2).

The memo servers are also used to start a distributed application on the network. This will include registering a user application during initial start-up (see section 4.4).

4.2 Partitioning (Work Distribution)

When an application is started, control is given to the appropriate application programs on the user specified network topology as defined in the application description file (see section 4.3). This file defines a directory where both the *boss* and *worker* programs' source code can be found. These two types of programs typically use the host-node paradigm; where the *boss* is the controlling process and the workers do the parallelized/distributed work (other programming paradigms are also supported). Typically, the input/output of the application is done in the controlling process (*boss*). Since the *boss* application typically has overall control of the parallel operations, it is generally used to start the parallel operations by distributing the data sets to the appropriate *worker* applications. In addition, It is typically used to determine when all necessary work has been completed.

The *worker* programs typically do all the parallel operations; each executing the same (or different) code in parallel on different processors. It's up to the application software (which supplies the *worker* and *boss* programs) to distribute the data sets using adequate medium to larger grain distribution. Because of HC environment, applications that use a small grain size distribution of work will have to consider the effects of overhead spent on communicating, versus getting work done. If the grain size is too large, parallelism will have been lost.

4.3 Application Description Files

The logical network topology is defined by an Application Description File (ADF). This description file defines the communication scheme used by the System. It not only describes the hosts that will be used for an application, but also the logical topology that connects the hosts to the network. This topology may, or may not follow the physical network inter-connections. This offers an application to selectively run on different types of topologies; e.g. Star, Cube, Ring, and Mesh. Depending on the application, it may be beneficial to have more control over the message routing than the physical connections allow.

The ADF defines what the network looks like for a

particular application. Each application running in the D-Memo system can use either the system default ADF, or register its own (see "Application Registration Process" on page 6). This allows an application to customize its own logical network and define its communication characteristics, or default to the system's ADF description. The system's default ADF is constructed when installing the system on a network. Each ADF has five sections: the application name, host machines, folder servers, user processes, and the logical point-to-point machine connections. Each section will be started with a keyword, followed by the appropriate data. Any section missing will default to the appropriate system ADF section.

The first section defines the application's name. The servers prepend the application's name with each requested folder name. This will allow more than one application to run concurrently on the system. It provides a unique folder/application name combination so that the same memo and folder servers can be shared over the network. By defining unique application names, applications will share data between only their own processes. This sharing of data is fully distributed in time and in space, as is Linda [6]. By using common application names, different programs will be able to communicate. This provides the idea of being fully distributed in time and space over *multiple* applications. Note that eventhough the memo servers are shared over applications, each memo server is loaded with unique routing tables for each application. An example of naming an application called *invert* is:

```
# Application Name
APP invert
```

Comments in the file are preceded by a '#' symbol.

The second section defines the host machines. It defines the machines that will be used in the computation for this application. Each machine is listed by it's internet address, followed by the number of processors it has, the architecture type, and the processor cost. The architecture type names can be used as variables in computing the processor cost in relation to other processors on the network. The processor cost is implemented into the routing table algorithm where a faster processor may have more weight than a slower one. Note that the internet address does *not* imply that only internet addressing modes are supported (e.g. work is currently underway to investigate integrating INMOS transputers). The following example illustrates the definition of four host machines.

```
HOSTS
# Hosts             #Procs Arch    Cost
glen-ellyn.iit.edu   1     sun4    1
aurora.iit.edu       1     sun4    1
joliet.iit.edu       1     sun4    1
bonnie.mcs.anl.gov 128     sp1     sun4*0.5
```

The above host section defines three Sparc workstations labeled with an architecture type of *sun4* with a processor cost of 1. The other host is an IBM SP-1 MPP machine with 128 processors. Notice that each individual processor on the SP-1 is less expensive to use then a Sparc.

Following the host section is the folder server descriptions. Each folder server that is used, is given a numeric name followed by host machine name that it will reside on. If a machine has more than one server, the numeric names can be combined as shown below. Note that only one is required on the network, but by distributing folder servers over the network, the application will get a better distribution of message traffic on all communication links. The following example illustrates defining nine servers.

```
FOLDERS
#       Folder Location at
0       glen-ellyn.iit.edu
1       aurora.iit.edu
2       joliet.iit.edu
3-8     bonnie.mcs.anl.gov
```

The fourth section defines the application process distribution. Each process is given a numeric name and the host it will execute on. The directory defines where the source code exists for the *boss* and *worker* processes. In this example, three different source directories were given. The *boss* directory is in fact optional, which will facilitate Single Program, Multiple Data (SPMD) applications better. Each directory listed is a root directory that must contain a Makefile. This Makefile will build the executable for that process. This can be a single directory or the root node of a directory structure. Each executable is linked to the D-Memo library which provides access to the system API. The current version requires either manual intervention or packages such as the Network File System (NFS) or Andrew File System to access the executables on each of the remote machines. This issue is being addressed (see section 4.4). The following is an example process distribution on the network where we will be doing I/O using the *boss* process on the *glen-ellyn* machine. The example actually shows two worker source code trees, one that will run on the Sparcs, the other on the SP-1.

```
PROCESSES
#Proc Directory    Located at
0     boss         glen-ellyn.iit.edu
1     worker1      aurora.iit.edu
2     worker1      joliet.iit.edu
3-22  worker2      bonnie.mcs.anl.gov
```

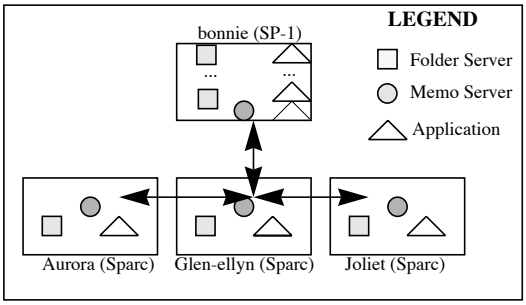
The standard executable names are *boss* and *worker* respectively. The typical use of the system is that the *boss* directory contains the executable code for the controlling process. The *worker* directory contains the executable code that does the parallel and/or distributed computation. But, different programming paradigms can be used.

The final section defines the logical point-to-point connections for the application. Each line defines either a duplex or simplex connection. Duplex connections are denoted by the '<->' symbol, simplex by '->'. A connection cost is also associated with each connection. The value represents the cost in using this link. This reflects distance and transmission speed. The point-to-point connection, defines a software level topology for a particular application. Each software defined link must have a corresponding physical connection. The following example defines a logical topology between the set of machines that will be used for the *invert* application. Note that the communication link between to the SP-1 is more

expensive than on the Sun network. Figure 3 illustrates this example.

```
PPC
# Point-to-Point Connection with cost
glen-ellyn.iit.edu <-> aurora.iit.edu   1
glen-ellyn.iit.edu <-> joliet.iit.edu   1
glen-ellyn.iit.edu <-> bonnie.mcs.anl.gov 2
```

Figure 3 Example Network Topology



This allows the user to define and restrict communication between hosts. This provision allows the users to define any one of many topology types (e.g. Star, Tree, Mesh, Point-to-Point, Cube, Systolic). The resulting effect, is allowing an application to have more control in defining its communication pattern. These connections are the basis for each application's routing table.

4.4 Application Registration Process

When an application is started up, it will register itself with all the memo servers it will interact. If, for example on a unix based operating system, and one or more of the servers are not running, they will be started up by the system *inetd* daemon. This registration process includes storing the application's name and it's routing table in each of the memo servers. This allows multiple memo applications to run concurrently, using the same servers (thus, not overloading a system with duplicate servers).

To start the registration process, the user enters "memo adf" on the command line, where "adf" is the user's ADF file. Each source code directory listed in the ADF should contain a makefile. If the binaries are out of date, they will be recompiled. The ADF tables will then be registered with each appropriate memo server.

Once the application has been registered with the system, the requested number of application processes will be started on each of the host machines. The current version does not support dynamic application cross-compiling and pumping of the executables to the destination remote machines. A current version is in design that will fully support the cross-compiling of the *boss* and *worker* executables by using a pumping method to get them to the appropriate remote host if NFS is not available.

5 System Performance

The main issue relating to performance is the memo distribution over the network. The ADF associates a processor cost in relation to the other processors on the network. By classifying each host with a ratio percentage of processing power, the system can control the distribution of memos. This is done by giving a higher percentage of proportional probability of hashing memos to a given host (in relation to the ratio percentage of the other hosts). With out this control, an even distribution would be seen over the folder servers. Thus, by associating the servers' host processor speed with the other hosts, the system will result in hashing the appropriate percentage of memos to each server.

The memo distribution also takes into consideration the network topology. When dealing with a network, the memo distribution on the network can drastically increase system response time. The routing class takes into consideration communication costs based on distances (machine localities) as specified by the ADF. Each link in the topology has a weight associated with it in which the routing class incorporates into the folder name hashing. The result is that hashing a memo to a folder server considers communication link and processor overhead. No broadcasting is done by the system.

6 Memo Language (API)

This section provides a brief introduction to the systems API (member functions of class Memo). Also shown are a sample set of shared data structures and synchronization mechanisms that are supported by the API.

6.1 Application Programming Interface

6.1.1 Keys and Values

A key is defined to be symbol, S, followed by a vector of unsigned integers, X. The key is used to represent a folder name. This is a slight departure from the familiar key definition in the context of associative tables, which is usually a string of characters. The definition is equivalent, however, and the purpose of not using a character is to provide better support for data structures. A function, `create_symbol`, is provided to create a unique symbol.

A value is defined to be a pointer to a transferable object[a]. The value represents the contents of a memo stored in the folder space. It is not necessarily to understand the notion of a transferable instance to study the later examples, but the essential point is that any data structure can be entered and extracted intact from the memo space with no programming effort for the application developer.

6.1.2 Basic Functions

Several basic abstractions are provided to extract, examine, and store memos in the system. A bullet item list illustrates them.

- `put(key,value)`

Put "value" in the folder labeled "key". Control is immediately returned to the executing process.

- `put_delayed(key1,key2,value)`

Put "value" in the folder labeled "key1". It will remain in the folder "key1" until another memo arrives into that

a. A transferable may be complex (e.g. a message) or a scalar.

folder. The “value” in folder “key1” can not be extractable by another process. Once a value has been put into folder labeled “key1”, the passed value will then be placed into folder labeled “key2” where “value” will now be accessible. This operation facilitates data flow operations. Process control will immediately be returned.

- `get(key)`

Get a value from folder labeled “key”. If no value is present in the folder labeled “key”, the executing process is blocked, until a value becomes available.

- `get_copy(key)`

Identical to the get function; however, a copy of the value in the folder labeled “key” is returned, thus enabling another process (or the same process) to issue another get operation on the folder labeled “key”.

- `get_skip(key)`

Get a value from folder labeled “key”. If there is no value present, return NIL. This function is usually used to poll for messages.

- `get_alt(array_of_keys)`

Get a value from any one of the folders labeled by an element of “array_of_keys”. If more than one folder actually contains a value, nondeterministically return a value from an eligible folder. This function blocks until a memo is returned.

- `get_alt_skip(array_of_keys)`

Similar to get_alt, but will return immediately if there are no memos available.

6.2 Data Structures

Many commonly used data structures can be shared through the system by using memos and folders. A sample set is discussed below.

6.2.1 Named Objects

A folder that holds at most one memo can represent a dynamically allocated object on the heap. Instead of pointers to objects, we use folder names.

6.2.2 Arrays

Arrays of shared objects may be created similarly. The element a[i,j] can be stored in a folder whose name is constructed as:

```
FOLDER_NAME key;
SYMBOL a;
...
a = memo.create_symbol();
key.S = a;
key.X[0] = i;
key.X[1] = j;
key.X[2] = 0;
```

The above example illustrates using the key name to build a 2-dimensional array abstraction.

6.2.3 Unordered Queues

A folder is an unordered queue, so if order is not vitally important, process can communicate simply by passing memos through a folder.

6.2.4 Job Jar

An important use of an unordered queue is a job jar. The memos in the job jar indicate tasks to perform. When ever a process creates more work to do, it drops memos in the job jar. It is often convenient to have one job jar for each process and one common jar for all. The individual job jars are used for operations that must be performed by a particular process (e.g. file I/O, a file is typically opened in only one process). The primitives `get_alt` and `get_alt_skip` can be used to get a memo from either the local or common job jar.

6.2.5 Futures and I-structures

A future is an assign-once variable used to communicate between a producer (typically a subroutine) and a consumer (it’s caller). Both the producer and the consumer may run in parallel, with the consumer only being delayed if it attempts to fetch from a variable before it has been assigned. An I-structure (an “incremental structure”) is a collection (e.g. an array) of futures. I-structures were invented for dataflow [9]. In D-Memo, any folder that will have only one memo ever placed into it may correspond to a future. The consumer executing a `get`, `get_copy`, or `get_alt` fetching from the folder will be delayed until the value has been produced. The folder will vanish once the memo is removed. Since it is usually better not to block an entire process, the consumer can delay a memo (using `put_delay`) for a job jar in the future’s folder that will trigger the desired computation when the data becomes available.

```
FOLDER_NAME future, job_jar;
...
memo.put_delayed( future,
       job_jar, operation );
```

6.3 Synchronization Mechanisms

In addition to data structures, the folders and memos can be used by applications to provide synchronization methods.

6.3.1 Locks and Shared Records

Shared records are accessed by getting them from their folders, examining and updating them, then putting them back. While the record is being updated, it’s folder is empty. If any other process try to access it, it will be blocked. The records are implicitly locked.

```
FOLDER_NAME obj_name;
OBJECT *p;
...
p = (OBJECT *)memo.get(obj_name);
/* Record locked */
...
memo.put( p );
```

6.3.2 Semaphores

The simplest implementation of a counting semaphore is identical to a lock, except that the semaphore is initialized with as many memos as needed in the counting semaphore.

6.3.3 Dataflow

Dataflow programming triggers execution of code when it's operands become available [9][10]. The system simplifies dataflow programming by providing the `put_delayed` procedure. Assume the operands are futures. One simply arranges to have an operation dropped into a jar when an operand memo arrives in a folder.

```
FOLDER_NAME operand, job_jar;
...
memo.put_delayed( operand,
       job_jar, operation );
```

7 Relation to Other Research

The idea of a virtual machine is not new, and one might wonder why yet-another-paper addressing the issue should be written. Most of the related efforts pertaining to virtual machines have been successful in presenting computational models amenable to the virtual machine concept, but little effort has been dedicated to the harder problems of data modeling and engineering.

The Linda research was used to create the illusion of a virtual machine, wherein an arbitrary number of processes communicated via a virtual shared memory known as a tuple space [6]. We believe that this tuple space is just "a flat directory of unordered queues". Using this approach, we are able to provide better programming abstractions then Linda (e.g. job jars, dataflow).

Parallel Virtual Machine (PVM) is a low-level approach taken to support the virtual machine concept [11]. A system service is provided for each machine on a heterogeneous network. The interface between two processes on the network is possible via a subroutine library. The routines in the subroutine library allow processes to communicate with one another without knowing the details of communicating with the system service. The limitations of this work are the dependence on TCP/IP at the transport and network layers, the lack of mechanisms to handle synchronization and communication reliably, and the ability to handle dynamic data migration between HC machines.

Mentat is a system that offers many of the advantages of the Linda and PVM systems with some enhancements [12]. At the application level it offers a balance between explicit and implicit parallelism by providing an extended C++ development language. Through C++ extensions and a run time system, Mentat is able to provide applications with an environment to support fine-grain and coarse-grain parallelism. The coarse-grain parallelism is supported via a "macro-dataflow" library. One issue, is the problem with handling dynamic data migration between HC machines.

8 Conclusion

We have discussed the importance of the HC abstractions that ensure we have an extensible and portable system. We addressed the need to fully use the networking capabilities and resources of an organization(s) by providing a seamless environment to the application code. The idea of the "shared directory of queues" provides a communication interface to the application that supports many types of data structures and synchronization mechanisms.

We have omitted significant details in describing the implementation of the D-Memo System but have presented the fundamental abstractions and characteristics of the system. We are building a framework to exploit all aspects of network utilization and machine processing power but, simultaneously, providing a cohesive and coherent environment for software engineering. Our intermediate results look promising, which leads us to the conclusion that further research on the D-Memo system is worthwhile.

9 Acknowledgments

The authors gratefully acknowledge use of the Argonne High-Performance Computing Research Facility. The HPCRF is funded principally by the U.S. Department of Energy Office of Scientific Computing.

10 References

[1] A. Khokhar, et al., "Challenges and Opportunities", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 18-27

[2] R.F. Freund, H.J. Siegel, "Heterogeneous Processing", *IEEE Computer*, June 1993, vol. 26, no. 6, pp. 13-17.

[3] A. Deogirikar, Keynote Speaker, TOOLS USA '93 Conf. on OO Tech. Santa Barbara, CA. Summer '93.

[4] T.W. Christopher, "Message Driven Computing and its Relationship to Actors", *Proc. ACM Sigplan Wkshop on Object-Based Conc. Prog.*, San Diego, CA. 1988.

[5] G.K. Thiruvathukal, et al., "A Simulation of Demand Driven Dataflow: Translation of Lucid into Message Driven Computing Language.", *5th Int'l Symp. on Parallel Proc.*, Anaheim, Ca. 1991.

[6] D. Gelernter, "Generative Communication in Linda", *ACM Transactions on Parallel Languages and Systems*, Vol. 7, No 1, Jan. 1985, Pages 80-112.

[7] W. O'Connell, et al. "A Generic Modelling Framework for Building Heterogenous Distributed and Parallel Environments", *Proc. 10th Int'l Conf. on Adv. Sci. & Tech.*, Naperville, Il, March, 1994.

[8] Blossom, "Decoding ASN.1 Transfer Syntax", *The C Users journal*, Sept. '91, vol. 9, num. 9, pp 57-63.

[9] Arvind. "*I-structures: An Efficient Data Type for Functional Languages*", TR LCS/TM-178, MIT, '80

[10] A.H. Veen "Data Flow Architecture", *ACM Computing Surveys*, 18, 4, Dec. 1986. pp. 365-396.

[11] Beguelin, A. et al. "*A User's Guide to PVM: Parallel Virtual Machine*", TR ORNL/TM-11826, Sept.'91.

[12] A. Grimshaw "Easy-to-Use object oriented Parallel Processing with Mentat", *IEEE Computer*, May '93

[13] W. Xu, G. Campbell, "A Distributed Queueing Random Access Protocol for a Broadcast Channel." *Comp. Comm. Review, ACM SIGCOMM*, Oct. '93.