1994

Distributed-Memo: Heterogeneously Concurrent Programming with a Shared Directory of Unordered Queues

William T. O'Connell

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

Thomas W. Christopher

Recommended Citation

Distributed-Memo: Heterogeneously Concurrent Programming with a Shared Directory of Unordered Queues*

William T. O'Connell	George K. Thiruvathukal	Thomas W. Christopher
AT&T Bell Laboratories	R.R. Donnelley Technical Center	Illinois Institute of Technology
600 Mountain Ave.	750 Warrenville Road	10 West Federal Street
Murray Hill, N.J. 07974	Lisle, IL 60532	Chicago, IL 60616
wto@research.att.com	gkt@disney.donnelley.com	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 - Parallel and distributed processing. Languages, Heterogeneous computing. Directories of unordered queues. Dynamic Data Migration. Portability.

1   Introduction

Heterogeneously distributed parallel computing allows applications to execute over an interconnected cluster instead of a single supercomputer or Massively Parallel Processing (MPP) machine. This provides the flexibility of using under-utilized high-performance workstations, single MPP machines, multiprocessor machines, and/or vector supercomputers to accomplish a task [1]. The purpose is to exploit this under-utilization by combining the power of heterogeneous high-performance machines into a unified virtual parallel machine. The last several years have seen a widespread acceptance of this alternative approach to parallel processing.

This does not imply that a cluster of workstations can replace a $30M supercomputer. However, a cluster of high-performance workstations using the proper granularity can be used with some impressive results [1]. The key to this ability is the scalability of several hundred million dollars' worth of networked computers, described by Gordon Bell as the "Ultracomputer: a Scalable computer", by creating a teraflop's worth of networked computers for an application [2].

The irony to this enormous parallel computation through scalable clustering is that software technology has taken second seat to the hardware movement which has taken place over the last two decades [3]. It has become extremely difficult to design, write and maintain parallel applications which are distributed over heterogeneous machines, especially for the general computer scientist. The scientist must not only focus on the problem space to be solved but is burdened with differences in hardware and operating system interfaces over multiple machines [3]. To further separate the real problem space from the programmer, transport protocols must be differentiated. For example, a typical application may be spread over several machines using multiple transport protocols, such as an IBM SP-1 MPP and a dedicated alpha cluster interconnected with a HiPPI switch.

In this example, process communication within the SP-1 would use the IBM EUI-H transport in order to minimize latency times using the high-speed switch. Communication within the Alpha cluster would use the HiPPI transport maximizing the capabilities of the cluster's giga-switch. Between the cluster and SP-1, the application may need to use the TCP/IP transport due to distance and reliability. As a result, the application would be cluttered with interoperability code.

The goal of supercomputing's recent software movement in high-performance computing is to make parallel processing a more attractive option. This includes providing improved systems, libraries, tools, and languages which are easier to use. One notable system that attempts this is Linda, which provides virtual shared memory over multiple machines [6].

Out intent is to show that we have taken Linda, an elegant coordination language over heterogeneous machines, saved the good ideas, disposed of the rest, and added simple but powerful mechanisms for communication and synchronization. In [5], we have illustrated mechanisms of Distributed-Memo which greatly enhance system efficiency along with how Application Description Files[a] can be used to vastly improve flexibility.

In this paper, we will emphasize the Distributed-Memo programming model by showing examples of common programming techniques and shared data structures. We will also describe the systems integrated framework structure which also provides additional communication techniques to the application [5]. We will then contrast its primary interface with the similar Linda programming model illustrating areas of enhancements and deletions. Finally, we will present conclusions.

2   Problem Spaces to be Addressed

We have taken a step back from the direct Linda approach

---

a. These files allow applications to define the topology, where folder servers and user processes will be placed, etc.

[6] to study the problem space of heterogeneously distributed parallel computing for the general programmer [3]. Our emphasis is on:

ï The need for an easy to use application programming interface (API).

ï The need to model the heterogeneous platforms and protocols, including dynamic data modeling.

The API must present a solid coordination language to the programmer that is simple to use, yet must support major MIMD parallel programming paradigms. Also, the system must also be able to handle heterogeneity in an elegant fashion by offering portability, reusability, and extensibility for the application as well as the system as a whole, yet maintain high performance. Heterogeneity issues are accomplished through the integration of the Generic Modelling Framework by incorporating its design into the kernel of the Distributed-Memo system [5].

3   Distributed-Memo Programming Model

The Distributed-Memo (D-Memo) system provides simple synchronization and communication mechanisms for parallel processes distributed over heterogeneous machines. The communication method is provided through a shared directory of unordered queues (or a shared table of bags). A secondary method is also available by accessing methods provided directly within the implementation of the Generic Modelling Framework [4]. This implementation serves as the kernel of the D-Memo system. Such communication methods offered by this model are direct process to process communication.

Since we are primarily concerned with concurrent programming with the shared directory, our discussion will focus on it. Readers interested in access methods provided by the Generic Modelling Framework should see ì Integrated Framework for Heterogeneityî on page 6 and [3][4]. In addition, readers interested in application start up methods along with application control through ìApplication Description Filesî are referred to [5].

Many conventional distributed-memory programming models allocate one process to each node, achieving communication through message passing. The issue with such systems is that data structures are not global, but rather localized in each process. One of the most common programming techniques is to manipulate a data structure. But, when trying to use that technique on distributed-memory systems, the data structures must be partitioned and the parts hidden in a fixed number of processes. This creates a sense of artificial complexity in programs.

This notion of virtual shared memory (or a shared directory) is not new, for example distributed shared memory operating systems and file systems, such as the Andrew File System [3], have been around for some time. A notable parallel processing package supporting virtual shared memory is Linda [6]. The Linda parallel programming system attempts to provide a more natural parallel programming environment through a shared associative memory, referred to as tuple space. It is obvious that programming in a shared memory paradigm is more intuitive and easier than a distributed memory

paradigm. A comparison of major message passing and shared memory packages in [7][8] illustrate this point.

D-Memo cannot claim to provide any type of new computational model, since both directories and queues have a long history and have been known to be useful in operating systems and parallel programming. By making directories and queues globally accessible to a heterogeneous parallel processing application, we believe that the D-Memo system will be a valuable parallel programming system to support high-performance computing.

3.1   Distributed-Memo Concepts

The D-Memo system presents a virtual shared directory of unordered queues which can support a variety of shared data structures and parallel programming techniques, such as named objects, arrays of objects, locks and semaphores, unordered and ordered queues, job jars, futures, remote procedure calls, selection of alternatives, and barriers. Some of these data structures and techniques are discussed in the seminal paper on generative communication in Linda [6]. D-Memo retains the simplicity and power of Linda in its ability to support the major parallel programming paradigms but is also able to support less popular paradigms, such as *dataflow*.

In this system, queues are referred to as folders and messages as memos. The communication scheme allows processes to communicate through memo passing. Memos are deposited into any one of the shared folders (directory of unordered queues). If a folder does not exist, it is created when a memo is deposited. This allows any process to either examine, extract, or place memos into them. By distributing the folders over the network, a pool of segments are 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; made possible by the Generic Modelling Framework.

The system is installed with a default configuration file, e.g. where to place the user processes and folder servers, what the network topology looks like, and the associated processor and link costs. Each application has the ability to override any part of the configuration through the use of Application Description Files [5]. Typically, applications just override where to place processes during start-up and where the executables located.

3.2   Basic Distributed-Memo Facilities

Before discussing the basic D-Memo facilities, folder names will be described. A folder name acts as a key to a specified folder, it is used to identify a unique unordered queue. The following code fragment shows the folderís name construct in C.

```
typedef struct FolderName {
    SYMBOL S;
    unsigned long X[NUM_X];
} FOLDER_NAME;
```

A SYMBOL is an unsigned integer which is intended to be unique over the execution of the program. The system can generate a unique symbol by invoking an available member function. The S member represents the name of the data structure that this folder is part of, e.g. the name of an array. For an array, the X member contains the indices. By default, NUM_X equals three, but it can be changed if the application demands it.

3.2.1   Basic Procedures

The central operations of D-Memo examine, extract, and insert memos into folders. Table 1 shows a subset of the systems API.

Table 1:

Subset of Application Programming Interface	
Interface	Description
Put	Put a memo into a folder
PutDelayed	Facilitates dataflow and futures
Get	Extract memo (blocking)
GetCopy	Get copy of memo (blocking)
GetSkip	Extract memo (non-blocking)
GetCopySkip	Get copy of memo (non-blocking)
Alt	Extract memo from any one of multiple folders (blocking)
AltSkip	Extract memo from any one of multiple folders (non-blocking)
Key	Given a memo, return its name (key)
Free	Free a memo

In addition to the basic operations shown above, a subset of the systemís process management interfaces are illustrated in table 2. These interfaces provide the application with control over its environment.

Table 2:

Subset of Process Management Interfaces	
Interface	Description
Init	Start D-Memo system, including starting initial multi-processes
Exec	Create threads and/or different executables on this or another machine

Table 2: (Continued)

Subset of Process Management Interfaces	
Interface	Description
GetMyId	Get logical process ID for application
NumWorkers	Number of processes in application
Exit	Process exit D-Memo

To illustrate some of the basic principles of memo manipulation, several examples will be shown. The following segment of code extracts a memo object from a folder, modifies it, then places it back into the folder.

```
Message something { int16 member; } *ptr;
FOLDER_NAME what_ever;
...
ptr = (something *) Memo.Get( what_ever );
...
Memo.Put( what_ever, ptr );
```

The programmer creates a message similarly to structures in C. A translator is used to convert each message reference into an object[a]. Each object then becomes what is known as a complex transferable which is constructed using other complex or scalar transferables [4]. In this case the int16 is a scalar transferable supplied by the system, which is a 16-bit integer. The following example illustrates a more complex message where an application may build a tree structure, with possibly circular lists:

```
Message msg_tag_name {
    int16 node_value;
    Message msg_tag_name *left;
    Message msg_tag_name *right;
};
```

The application references the message definition in the code just like a structure definition. When the user passes this transferable to the system, it will recursively encode itself (which indicates that the whole tree structure will be passed as part of the message).

Since object-oriented technology is being used, each message type has a base class (base transferable object). It is the base class reference that is being used with the Get and Put operations above.

Each transferable is in fact a persistent object that knows how to encode/decode itself (including linearization of non-contiguous memory) for transmission over a network. For the case of complex objects, they are encoded/decoded recursively. This does not require programmer intervention.

For most applications, the typical message is either a single structure or a scalar transferable[b] (e.g. int16) which causes

a. This packages is built using object technology and provides an object-oriented API.

little or no real-time impact for data encoding. As applications become more elaborate, higher complexity data structures can be created and passed transparently. However, as the complexity goes up, so does the cost of linearization. For example, using the previous message definition with left and right node pointers, a complex tree can be constructed as shown in figure 1.
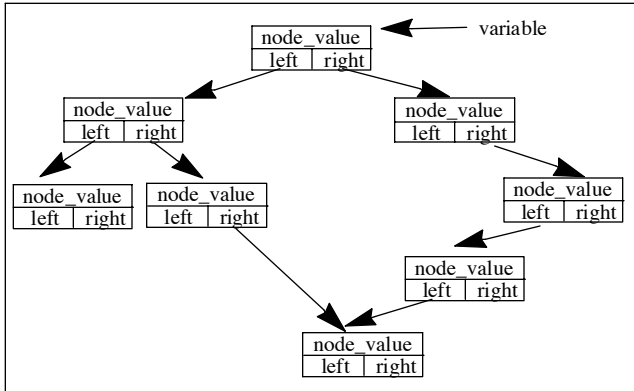
Figure 1   Elaborate complex transferable

As shown in figure 1, the application has a variable pointing to the treeís root. When passing this value to a D-Memo operation, the entire tree structure will be linearized when passed. When a process extracts this complex transferable, it will be reconstructed. Note that self-referencing lists with cycles can be constructed.

Another important aspect of the systemís API, is to be able to retrieve a memo from any one of multiple folders.

```
ptr = (something *) Memo.Alt( key_list,
        list_len, one_found );
```

The key_list parameter is an array of pointers to folder names where the list_len indicates the number of folders in the list. The interface will nondeterministically return one memo from within the list. The one_found parameter is the index into the array from where the memo was extracted.

3.2.2   Data Structures and Programming Techniques

Many common data structures and programming techniques are directly supported by D-Memo. The system easily supports any of the major parallel programming paradigms, such as SPMD[a], MPMD[b], host-node, and data parallel. The type of paradigm being used by an application is mainly attributed to its layout in the application description file [5]. The following list is some of the most useful programming techniques.

3.2.2.1   Named Objects

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

---

b. Scalar transferables can be passed directly without being placed in a message format.
a. Single Process, Multiple Data
b. Multiple Processes, Multiple data

3.2.2.2   Arrays

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

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

The key is now a folder name for the a[i,j] array element. Each array element is itself stored in a separate folder. All array elements are not necessarily associated with the same folder server (one or more folder servers may be used by the run-time system, each managing a set of folders [5]). By spreading folders (that will be referenced within close locality of each other in the application code) throughout the distributed machines, the system does not congest one server by creating communication hot spots. But, is able to provide a more balanced concurrent access model.

3.2.2.3   Locking Shared Data Structures

Shared records are accessed by getting them from their folders, examining and updating them, and then putting them back. While the record is being updated, its folder is empty, and any process trying to access it will be blocked until the record is replaced; the records are implicitly locked. The following exemplifies operations on shared records:

```
FOLDER_NAME record;
Message record_obj *ptr;
...
ptr = (record_obj *) Memo.Get( record );
... /* Operate on record ptr*/
Memo.Put( record, ptr );
```

In this example, the record template was already declared (possibly in a header file), the ptr variable is defined just like a C structure pointer.

3.2.2.4   Locks and Critical Sections

Naturally, an explicit lock can simply be represented by an empty memo in a folder.

```
FOLDER_NAME lock;
char8 *token;
...
/* Initialize Lock */
if ( Memo.GetMyId() == DM_MASTER )
      Memo.Put( lock, token );
...
token = (char8 *) Memo.Get( lock );
... /* Perform critical section */
Memo.Put( lock, token );
```

This is a typical example of a SPMD application, where the

master process initializes the lock. The master process is always logical process ID number zero (DM_MASTER).

Note that a message structure definition is not being used as the lock object, but a 8-bit character is (scalar transferable). Scalar transferables supplied by the system also have a base class of Message. Other examples of scalars are int8, int16, uint16, int32, int128, float32, etc.

3.2.2.5 Semaphores

The simplest implementation of a counting semaphore is identical to a lock, except that to initialize the semaphore, a process places as many empty memos into the semaphore's folder as are required by the initial count.

3.2.2.6 Unordered Queues

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

3.2.2.7 Job Jar

An important use of an unordered queue is a job jar. The memos in the job jar indicate tasks to perform. Whenever a process requires more work to do, it pulls a memo out of the job jar. Whenever a process creates more work to do, it drops memos in to 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 on a particular machine or a process running on a special purpose machine. Expanding this idea one step further, all process on a particular class machine, such as the IBM SP-1 MPP machine, may want to share a job jar. While, all processes on a Cray C-90 may share another.

The Alt interface can be used to get a memo from one of multiple job jars.

3.2.2.8 Futures and I-Structures

A *future* is an assign-once variable used to communicate between a producer (typically a subroutine) and a consumer (its caller). Both the producer and 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 ìincremental structureî) is a collection (e.g. an array) of futures. I-Structures were invented for dataflow hardware [9].

In D-Memo, any folder that will have only one memo ever placed in it may correspond to a future. The consumer executing either a Get, GetCopy, or Alt fetching from that folder will be delayed until the value as been produced. The folder will vanish once the memo has been removed.

Since it is usually better not to block an entire process, the consumer can delay a memo that will eventually be placed in its job jar after a future is populated, and continue executing. When the producer finally a memo into the future's folder, it will trigger the desired computation by delivering the delayed memo to the consumer's job_jar.

In the following example, the consumer delays a memo for its job jar. After the producer places the data into the folder named future, the delayed memo will then be placed into folder job_jar, which is the job jar for the consumer. This prevents the consumer from blocking on the memo that is not available yet.:

```
FOLDER_NAME future, job_jar;
Message operation *op;
...
Memo.PutDelayed( future, job_jar, op );
```

3.2.2.9 Dataflow

Dataflow programming triggers execution of code when its operands become available [10][12]. The D-Memo system facilitates dataflow programming by providing the PutDelayed interface. Assume the operands are **futures**. The application simply arranges to have an operation dropped into a job jar when an operand memo arrives in a folder.

Thus, using the PutDelyed interface (as shown above), when the operand (memo) arrives into the operand folder, the delayed memo (operator) will be placed into the job jar. When this job jar messages is processed, the operand will be read by the process executing the operation.

If the operation requires more than one operand, then the operation can poll for each of the operands (e.g. the MemoPresent interface) and delay itself (using Put Delayed again) on absent operand folders until all are available.

3.2.2.10 Reactive Objects

A reactive object is an object that executes only in response to messages it receives. Reactive objects are central to the Actors model of parallel computation [11]. A reactive object can be implemented with a job jar folder plus one input folder per object. A memo containing the object is delayed (using PutDelayed) on its input folder. When an input memo arrives, the object is placed into the job jar. Eventually some process fetches the object from the job jar and executes the code, which reads the input message (in the input folder) and responds to it. Once the object responds to the message, a memo is delayed again in the input folder waiting for the next input message for this object.

For efficiency, it is better to loop reading all available input messages with the GetSkip interface, rather than delaying the object after each individual incoming message.

3.2.2.11 Ordered Queues

The simplest implementation of an ordered queue is an array of futures. For example, the producer can write memos into folders as:

```
#define QUEUE ...
...
FOLDER_NAME dest;
int outposition = 0 ;
dest.S = QUEUE;
dest.X[1] = dest.X[2] = 0;
...
while ( 1 ) {
        ... Produce item M ...
        dest.X[0] = outposition++;
        Memo.Put( dest, M );
}
```

And the consumer can remove them similarly:

```
...
while ( 1 ) {
        mine.X[0] = inposition++;
        ptr = (message *) Memo.Get( mine );
        ... Consume item M ...
}
```

3.2.2.12   Remote Procedure Calls

Remote procedure calls can be simply emulated by using a specific process as a subroutine handler. This process can handle one or more subroutine calls using the Alt operation. Each subroutine will have an invocation call folder associated with it (similar to a job jar). Invocation of the subroutine is caused by writing a memo into its folder. The subroutine handler will read the memo, which contains the subroutineís arguments, and will execute the appropriate code. If a return code (message) is required from the subroutine, the return folder should be stated in the subroutine call arguments (this eliminates concurrency problems when writing to one return folder).

3.2.2.13   Barriers

In a parallelizing loop, it is often important to synchronize the processes executing the loop at one or more points within it using a barrier. If a barrier is initialized with a count N, then N processes must wait at the barrier before any of them are allowed to proceed. The barrier is automatically re-initialized so that the processes can synchronize over and over again at the same barrier. In D-Memo, a barrier can be implemented with two folders and a single memo containing an integer count. the presence of the memo in a folder allows the processes to proceed past the barrier.

The processes alternate which folder they look for the memo in. When a process comes to a barrier, it gets the memo and decrements the count in it. If the count is greater than zero, this is **not** the last process to arrive, so it puts it back and tries to get a copy of the memo in the other folder (which is not present yet).

If the process decrements the count to zero, then this is the last process to arrive at the barrier. It puts a memo with the full count of the number of processes to synchronize into the other folder and continues executing. This will allow all other processes to retrieve a copy of this memo in the other folder, concurrently, and continue executing.

Sample code using a barrier:

```
FOLDER_NAME barrier[2] =
            {{BARR0,0,0,0}, {BARR1,0,0,0}};
int which = 0;
int16 Tcount = NUM_TO_SYNCH;
...
/* Initialize barrier */
Memo.Put(Barrier[0],&Tcount);
...
while ( TRUE ) {
        int16 *count;
        ...
```

```
        ...
        /* Synchronize at the barrier */
        count = (int16 *) Memo.Get( barrier[which] );
        (*count)-- ;
        if ( *count == 0 ) {
            /* Iím the last to arrive */
            which = 1 - which;
            Memo.Put( Barrier[which], &Tcount );
        } else {
            /* I am not the last to arrive */
            Memo.Put( Barrier[which], count );
            which = 1 - which;
            count = (int16 *) Memo.GetCopy(
                        barrier[which] );
        }
        Memo.Free( count );
        ...
}
```

We should note that this code can easily be placed into a library routine, and return to the caller after all processes reach the barrier.

4   Integrated Framework for Heterogeneity

For applications to fully exploit heterogeneously distributed and parallel environments, software support must be provided that is easy to use. By eliminating *platform specific issues*, the task of writing software in this complex environment is cleaner and allows the application to concentrate on the problem space. In addition, transparency offered by proper layering provides better code reusability and portability when moving applications to new architectures.

As we described in [4], the Generic Modelling Framework describes the foundation for building the core of any heterogeneous parallel application. We have built a set of generic software modelling techniques through the utilization of object frameworks, each being an object cluster providing a generic interface.

The model is characterized by five clusters of abstractions: communication (message passing), transferables (data coercion and dynamic data migration), shared memory, locking (synchronization), and process management. To support a new computer in a heterogeneous network, one must consider these five aspects. Our view is that a new computer can be supported by learning the differences from the base definition (called a base class in object-oriented terminology) and either extending or overriding the services provided in the base definition. This extension would be done once by a specialized programmer, all software using the services of these core abstractions will then reap the benefits of transparency, reuse, portability, and interoperability.

We have built D-Memo on top of an implementation of this model in order to provide a seamless interface to the incompatibilities of hardware, operating systems, and transport protocols. The application program no longer must deal with these issues and can spend more time solving the

real problem space issues. This allows the system to be more portable and extensible over different architectures and protocols with a higher degree of software reusability.

To understand how the abstractions aid in designing D-Memo, the reader is referred to [4][5]. The reader should note that while these frameworks are being used by D-Memo, they **are also accessible** to upper layering software or software at large.

5   Contrasting Linda

Linda has been considered by many as a candidate for a more natural parallel programming system which provides a shared associative memory, the tuple space [6]. A tuple is a sequence of fields, each of which has a type and contains a value or a variable.

Tuples are written into the tuple space with an out operation, are removed with an in, and are read without being removed with a rd. For an in or rd, the tuple accessed in tuple space must match the tuple provided with the operation. The number and types of fields in both the operation and tuple space must be identical. A positional value in an operation must match the identical positional value in the tuple space. A variable in either must match a value in the other. A variable will not match a variable. The in and rd operations will block until there is a matching tuple in the tuple space. Recent implementation of Linda have provided non-blocking versions of rd and in, rdp and inp, which return a boolean indication of success[a].

Linda also has the command eval, which will start a process, referred to as an ìactive tupleî, which will return its value by becoming a tuple.

Various implementations of Linda have restricted the format of tuples, the implementation of eval, or have provided multiple tuple spaces.

The tuple space does allow data structures to be shared among processes, and the designers of Linda are quick to point out the advantages of being able to design programs around shared data structures. Still, a question persists: why does tuple space suddenly become a natural structure for programming when there is no demand for it in sequential programs? An explanation is that in almost all cases, the tuple space is being used as a shared, flat directory of queues. Since both directories and queues have long been known to be useful in multiprogramming and multitasking systems and since they can so easily be implemented in tuple space, it is no surprise that Linda is successful. But, this does suggest that a system built directly on a shared directory of queues might work as well as or better than Linda and that clarity on the data structures actually being used might suggest useful facilities that tuple space would not suggest.

The D-Memo system was designed and implemented to investigate the possibilities for parallel programming system built around a shared directory of queues. The system design

---

a. This implementation was developed by Scientific Computational Associates. Two versions exist, C-Linda and F-Linda for C and fortran programming languages respectively.

not only considered the shared directory of queues design, but an implementation on top of the Generic Modelling Framework. This allows the system to easily support high-performance heterogeneous systems by seamlessly building a virtual machine [5].

When reviewing the vast majority of published Linda algorithms, it is obvious that many of the code segments are directory-of-queue algorithms. In most Linda algorithms, the first several fields of a tuple, which we will call the **key** fields, area specified explicitly in both out and in operations. The remaining fields, the **entry** fields. are all given values by an out operation and are all assigned to variables by an in operation. Hence, the key fields correspond to the name of a queue, and the entry fields correspond to a structure (message) written into it. Translated into D-Memo, out corresponds to Put, in to Get, and rd to GetCopy.

There are a few other usages of Linda that requires slightly more than simple translation. For the same keys, there may be several different numbers and types of non-key fields. Linda provides further matching on their number and types. This can easily be encoded in D-Memo by using multiple folders, or by writing and reading discriminated unions.

The existence of tuples containing variables has no direct equivalent in D-Memo, but the examples of their use are infrequent. Linda provides an alternative, asynchronous write: server processes wait for tuples specifying their individual names. A request for service by a particular server specifies the serverís name. A request for service by any server specifies a variable in the name field, so that any server can pick it up [6]. Is cleaner and more efficient to have one queue for all servers in addition to an individual queue for each, and to have the servers wait for commands with the Alt operation. Linda lacks a read with alternatives, so this solution is unavailable to it.

Linda does try to integrate process creation and termination by eval creating active tuples. The ability to create many processes does make the lack of analogues to the Alt and PutDelayed operations less serious. However, D-Memo does support Exec where threads and other executables may be started. In addition, eval has been difficult to implement in portable forms and the many small processes it creates are expensive to run, especially on RISC processors. The assumption in D-Memo is that an application can easily incorporate data-parallel, macro-dataflow, reactive objects, or multiple small tasks using memos for state vectors and scheduling via job jars. In any case, a limited number of large processes is adequate.

With D-Memo, there is no need for a special compiler, unlike Linda which needs to compile the tuple patterns. However, if transferables are used, a translator is required to convert the message definitions to persistent objects. Linda does not support the idea of transferables.

D-Memo offers a number of **advances** over linda (in addition to clarifying the abstraction). The Alt operation allows the implementation of an analogue of a guarded input command. The PutDelayed operation facilitates dataflow and reactive object programming.

In addition, to the general D-Memo interfaces, the application also has access to the Generic Modelling

Framework interfaces. This model is the foundation for D-Memo. This includes a **simple generic interface** to the following:

ï Process creation over heterogeneous machines, both create threads and processes from different executables.

ï Direct process to process communication, over-riding the directory of queues.

ï Locking and shared memory capabilities outside the directory of queues on a machine basis.

Finally, the transferable framework provides an easy to use and powerful dynamic data migration package.

6   Conclusions

D-Memo is a parallel programming system based on communication through a shared directory of unordered queues. It makes possible a variety of shared data structures, including named objects, arrays of objects, locks and semaphores, unordered and ordered queues, job jars, futures, I-structures, remote procedure calls, and barriers. It also facilitates macro-dataflow [13] and reactive object programming.

If an application requires direct process to process communication, the system allows access to the capabilities of the Generic Modelling Framework [4]. This not only includes direct process communication, but generic shared memory and locking interfaces to the software. This provides a level of transparency to the application, so that it maintains a high level of portability, extensibility, and reusability over multiple architectures, operating systems, and protocols.

In our comparison to Linda, we suggested that Linda is successful not because of its tuple space abstraction, but rather that tuple space is valuable primarily because it allows the simulation of a directory of queues.

To conclude, D-Memo has taken the Linda programming model, an elegant but inefficient language under heterogeneity, saved the good ideas, disposed of the rest, and added simple but powerful mechanisms which greatly enhance efficiency. The result is just as elegant as Linda, but much more efficient and vastly more flexible [5].

7   Acknowledgments

8   References

[1]   A. Benguelin, et al., ìSolving Computational Grand Challenges Using a Network of Heterogeneous Supercomputersî, *Proc. fifth SIAM conf. on parallel proc. for scientific computing*, Houston, TX, Mar., 91.

[2]   G. Bell, ìUltracomputers: A Teraflop Before itís Timeî, *Comm. of the ACM*, Vol. 35, No. 8, Aug. 1992.

[3]   W. OíConnell, ìA Generic Modelling Framework for Building High-Performance Environmentsî, Ph.D. Thesis, IIT-HPLS-94-4, Illinois Inst. of Tech., 1994.

[4]   W. OíConnell, G. Thiruvathukal, T. Christopher, ìA Generic Modelling Framework for Building Parallel and Distributed Programming Environmentsî, *Proc. 10th Intíl Conf. on Adv. Sci. and Technology*, Naperville, IL, Mar. 26, 1994, FTP access: glen-ellyn.iit.edu;/pub/research/parallel/papers.

[5]   W. OíConnell, G. Thiruvathukal, T. Christopher, ìDistributed-Memo: A Heterogeneously Distributed Parallel Software Development Environmentî, *Proc. 23rd Intíl Conf. on Parallel Processing*, St. Charles, IL, Aug. 1994, FTP access: glen-ellyn.iit.edu;/pub/research/parallel/papers.

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

[7]   T. Mattson, ìProgramming Environments for Parallel Computing: A Comparison of CPS, Linda, P4, PVM, PSYBL, and TCGMSGî, *Intel Corp Research Rep*, 1993.

[8]   N. Carriero, D. Gelernter, ìLinda and Message Passing: What have we learned?î, Tech. Rept. YALEU/DCS/RR-984, Aug., 1993.

[9]   Arvind. ì*I-structures: An Efficient Data Type for Functional Languages*î, TR LCS/TM-178, MIT, ë80.

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

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

[12]   G. Thiruvathukal and T. Christopher, ìA Simulation of Demand Driven Dataflow: Translation of Lucid into Message Driven Computing Language.î, *5th Intíl Symp. on Parallel Proc.*, Anaheim, Ca. 1991.

[13]   G. Thiruvathukal and T. Christopher, ìMacrodataflow Implementation of Distributed Array Objectsî, Tech. Rpt. TR-HPLS-94-100. FTP access: glen-ellyn.iit.edu;/pub/research/parallel/papers.

[14]   G. Thiruvathukal, W. OíConnell, and T. Christopher, ìTowards Scalable Parallel Software: Interfacing to Non-von Neumann Programming Environmentsî, *Proceedings of the SIAMí95*, San Francisco, CA. Feb. 1995.