A Flexible Interface for Interprocessor Communication in a Distributed Memory Multicomputer: The Multifaceted Communication System

Maria D. Maggio David W. Krumme

Technical Report 90-5
Nov. 1990

Abstract

The Multifaceted Communication System for the distributed memory NCUBE is an interprocessor communication system that supports many features useful for application programs. These features include asynchronous, fully synchronous, non-blocking, and unbuffered I/O, as well as gather/scatter operations, broadcasting, a general pipe mechanism, and file operations. Mechanisms are provided to simplify the addition of new features when they become necessary.

1 Introduction

The Simplex operating system [Krumme90] is a locally written operating system for the NCUBE distributed memory multicomputer. It provides user programs with two ways to send data between nodes: a message-passing protocol patterned after that of NCUBE's Vertex operating system, which is a weakly synchronous interface with data buffering, and an original design for asynchronous unbuffered I/O based on Petri net control structures. Experience has indicated that different combinations of existing and potential features are required in different settings, resulting in a wide variety of possible system calls. However, we feel that instead of choosing among different communication paradigms, the programmer should be choosing different *options* for a single underlying communication paradigm. This paper presents a design for a communication interface that embodies that view. To highlight its role in providing a multiplicity of possible uses and interfaces, we call it the "Multifaceted Communication System."

A second objective in the design is modularity: communication activity should be fully supported in the Multifaceted paradigm, and links to other modules should work smoothly. For example, the Multifaceted System should answer the communication needs of the operating system kernel as well as those of user programs. It should interface with control systems such as signal-passing mechanisms, ordinary process control, and Simplex's Petri net control structures. It should directly or indirectly support higher-level communication abstractions such as file I/O and pipes.

Sections 2 and 3 of this paper describe the semantics of the various options in the Multifaceted paradigm. Sections 4 and 5 give particular information about pipes and files. Section 6 suggests possible uses of the Multifaceted System in application programs. Appendices A - D provide precise specifications for the user interface in the form of individual manual pages. Appendix E offers ideas about implementing the Multifaceted paradigm within an operating system kernel.

2 Read/Write Semantics

We now review the various issues and options relevant to the design of the **read** and **write** system calls, which are the core of the Multifaceted design. We view **read** and **write** as *actions* on *objects*, where the objects can be messages, pipes, or files. In this section we present the basic properties of these objects, and in the next section we enumerate the options that characterize the different actions that can be performed on them.

2.1 Messages

Communication between nodes under the Vertex operating system is accomplished by the **nwrite** and **nread** system calls, which have the following semantics. The user issues an **nwrite** call. The data contained in the user's buffer is copied into a system buffer before the **nwrite** system call returns. The kernel proceeds asynchronously to transmit the message to the destination processor, causing the message to be copied into a system buffer on the receiving side. The user on the receiving side at some point issues an **nread** system call. The data is then copied from the system buffer into the user's buffer. The **nread** call returns after the data has been copied into the user buffer.

These semantics are well-suited to most programs that send small messages in quantities that do not threaten to exceed the available buffer space. But for many programs different semantics would be helpful. For example, an algorithm that that sends large volumes of data among processors would be better served by a communication paradigm that could transmit the data directly from user buffer to user buffer, avoiding the copying of data between user and system buffers. An algorithm whose synchronization depends on the flow of data can encounter performance problems and even deadlock under the standard message passing paradigm, because the only form of flow control in that paradigm is the contention for system buffer space.

We would ideally like to provide the user with every reasonable variant of message passing semantics. The features we want include asynchronous I/O (in which system calls initiate transactions without waiting for them to complete), non-blocking I/O (in which systems calls abort whenever waiting would normally be required), fully synchronous I/O (in which write calls only return upon the reading of the message), unbuffered transmissions (in which data is copied directly to and from user buffer space), broadcasts of messages (in which messages are automatically replicated), and gathering and scattering transmissions (where data can be transmitted from or to the columns of a matrix as it is handled by the messaging system). In many cases, it makes sense to combine two or more of these options as in an asynchronous unbuffered communication or a gathering broadcast. The problem addressed in this paper is the design of a system-call interface that admits all these possibilities, that is simple enough to be usable, and that can be implemented with reasonable efficiency.

2.2 Pipes

Our concept of pipes follows the System-V UNIX concept of named pipes. However, we also envision a possible tree-like arrangement of connections so that a pipe can be filled from many processing nodes to be emptied by a singe process, thus supporting the collection of data at a point.

2.3 Files

We assume some processing nodes may have direct access to file systems, and we provide all nodes with access to such file systems through the Multifaceted communication primitives. This is done by forwarding data to be read and written to the nodes with direct access. We avoid making unnecessary assumptions about the nature of the file systems involved, describing only the mechanisms which allow one node to access files under the control of another.

3 Options for Reading and Writing

The options that are needed fall into three categories: (1) control and synchronization (non-blocking, fully synchronous, and asynchronous coordination in addition to the standard, weakly synchronous approach); (2) data movement (unbuffered, gathering and scattering methods, and data splitting with partial reads); and (3) source and destination addressing (broadcasting, specifying a process, and using multipurpose descriptors called handles).

Several of these options are mutually exclusive (e.g. broadcasting a partial read). Whenever this document does not specify the results of conflicting or illegal options, the user should consider the results as undefined.

3.1 Asynchronous I/O

The standard message passing scheme of the NCUBE in both Vertex and Simplex can be thought of as weakly synchronous. When a write call returns to the user, the data has been copied to a system buffer but has not necessarily been sent to its final destination. When a read call returns, the data has been copied to the user buffer. With writes and especially reads, the system call may require considerable delay before it completes. The new *asynchronous* option allows the caller to avoid such waiting: it initiates kernel actions but returns to the user as soon as possible, regardless of whether the actions have been completed. The caller supplies an optional argument that tells the kernel what to do when the actions are completed, and in this way the caller can be notified of the completion. For example, input completion means that the data is valid, while output completion means that buffers can be safely modified. A method of implementing this optional argument is described in Appendix E.

An asynchronous call will return without waiting if message buffers are not available in the write case, or if data is not yet available in the read case. However, since the kernel must allocate a data

structure to keep track of the request, an asynchronous call can block while waiting for kernel memory to become available for this purpose. Such waiting would not be common, nor would it normally take an extented amount of time. The *non-blocking* option can be used if the user does not wish to block even for this reason.

The *asynchronous* option is available with all other features.

3.2 Synchronous I/O

The *synchronous* option is only valid for write system calls. It is ignored for any read system call, any broadcast, and any operation on a pipe. If the user requests this option, the write call will not return until the message has been received by the recipient. Note the message is deemed to be received once the user program begins to copy the message into its buffer on the receiving side.

3.3 Non-Blocking I/O

The *non-blocking* option is available with all other options, and it specifies that the call should never wait for any resource or condition that is not immediately available. Under this option, instead of waiting the kernel returns an error code indicating that the call would have blocked, and perhaps also encoding the reason that the call would have blocked.

It is legal to request the *non-blocking* option on a call that cannot block.

3.4 Unbuffered I/O

Since multiple copying of data often causes a loss of efficiency in algorithms, unbuffered I/O is provided as an option. Unbuffered I/O is only supported for neighbors or local messages (i.e. messages

sent to the current node). An unbuffered read or write will wait for a matching unbuffered request from the sender or recipient respectively. A buffered request will not be matched with an unbuffered request.

Broadcasting an unbuffered message is illegal. Unbuffered I/O for a pipe has entirely different semantics. See the section describing pipes for more details.

3.5 Gather/Scatter

In buffered message passing, the copying of data between user and kernel buffers affords an opportunity to collect or distribute data that occupies diverse, noncontiguous locations in the user program. Therefore the Multifaceted System allows the user to refer to data using an I/O vector which is a series of quadruples: *address*, *size*, *stride*, and *repeat count*. Each quadruple defines a gather or scatter operation such as would be used with a column of a matrix stored row-wise: *address* is a starting address, *size* is the number of bytes in each block, *stride* is the spacing between blocks, and *count* is the number of blocks.

All other features are available using these I/O vectors.

3.6 Partial Reads

In the current version of Simplex (as well as in Vertex) if the user in a read call requests fewer bytes than the number contained in the message, the extra bytes are discarded. The *partial* option allows the user to read a message without disposing of any unread bytes. The message size is adjusted so that future reads will not read the same data again. The message is in no way reserved: another reader could issue a read call and get the remainder of the message. The user of this feature must insure the integrity of the messages.

The *partial* option is ignored for any write operation or any operation on a pipe.

3.7 Broadcasting

The user may wish to broadcast a message to every node in the currently allocated cube or to some subcube within the allocated cube. For this reason, the structure describing destinations contains, in addition to the target node's ID, a broadcast mask. Each bit set in the mask defines a dimension, and the target subcube is defined to be those nodes reachable from the target node through the dimensions specified in the mask. Note that a normal individual destination results when the mask is zero. As a special case, if the target node is -1, the current node is used as the target.

In conjunction with a broadcast, the user may also specify the *notme* option. In this case the current node will not receive the broadcast, even if it is included in the subcube specified.

The *notme* option is ignored if a broadcast is not involved. Broadcasting in a read does not make sense, and the kernel ignores the mask in read operations. Broadcasting is available in conjunction with all other options except unbuffered and synchronous I/O. Additionally, the *broadcast* and *notme* options are ignored for I/O involving a pipe or file.

3.8 Specifying a Process

Since Simplex supports multiple processes per processor, a user may direct a write or read operation to or from a particular process. Like other wildcarding, a -1 matches any process. The process number is encoded in the source or destination structure.

3.9 Naming Sources and Destinations

With pipes and files, an operation creating or opening the object returns a descriptor to be used in subsequent references to the object. A fundamental property of message passing is that no preliminary open operation is required. This means that in each read or write, a high-level name is given for the message's source/destination, type, etc. The Multifaceted System uses a single structure called a "handle" which contains either a message descriptor appropriate to message passing or a file or pipe descriptor. All system calls accept generic handles and can distinguish the different forms.

3.10 Using these Features

A group of new system calls and library routines is provided to use these new features. The manual pages for these calls are included in Appendix A. Macros are provided in Appendix D to simplify the creation of handles for message passing.

4 Pipe Paradigm

A new paradigm for communication among nodes in an NCUBE is the pipe paradigm. These UNIX-like pipes provide a convenient way to merge data as well as optimize the overhead of small messages since data does not actually get transmitted until a buffer has been filled. Most importantly, though, the pipe paradigm provides a convenient way to provide flow control to the user without using the typical NCUBE flow control mechanism of exhausting the message passing buffers.

4.1 Simple Pipe Uses and Examples

The UNIX concept of a named pipe has been generalized from a uniprocessor model to a distributed, multiprocessor one. The following sections describe some uses and examples of simple pipes so that this multiprocessor version of a pipe can be better understood.

4.1.1 Producer/Consumer Pipe

The simplest case of a pipe establishes a producer/consumer relationship between two nodes. Pipe creation calls are issued on both nodes to establish sets of buffers. As each buffer fills on the producer node, it is queued for transmission to the consumer node. If the producer fills its buffers faster than the consumer is willing to accept them, the producer process will be blocked as necessary. The consumer reads data from its buffers as they are filled by transmissions from the producer, becoming blocked as necessary when data is unavailable. The buffer capacity of each side is determined during pipe creation and generally is not changed thereafter. (It is possible to explicitly modify the buffer structure).

4.1.2 Multi-Hop Pipe

A multi-hop pipe, for example from node 3 to node 0 via node 2 can be created as follows: On node 3, the pipe is created with 3 as the source and 2 as the destination. On node 0, it is created with node 2 as the source and 0 as the destination. On node 2, the pipe is set up with 3 as the source and 0 as the destination, indicating that the role of node 2 is to forward data received from node 3 to node 0. Note that node 3 need not know what happens to the data after it reaches node 2, and node 0 need not know how the data reaches node 2.

4.1.3 Multiple Sources of Data

It is possible to create a pipe which merges the data streams from more than one source. If nodes 1 and 2 are producers of data consumed by node 0, each would create a pipe with itself as the source and node 0 as the destination. On node 0, two pipe creation calls would be made, specifying nodes 1 and 2 as sources and dedicating some buffers to each source. Data would be mixed on a buffer-by-buffer basis as it was received on node 0. To collect data from node 3, additional pipe creation calls would be issued on nodes 2 and 3. On node 3, the source would be indicated as node 3, and the destination would be given as node 2. On node 2, the source would be stated as node 3, but the destination would given as node 0, thus setting up node 2 as a forwarder of data from 3 to 0 as in the earlier example. In this case, the data from node 3 would be merged at node 2 with its locally generated data before transmission to node 0. This pattern provides an efficient, scalable method of collecting data from a subcube onto a single node using the pipe mechanism.

4.2 Pipe Management

Pipes are created and deleted using new Simplex system calls. There are also control system calls which allow the user to expand or shrink an existing pipe or force a flush of a partially filled buffer within a pipe.

Pipes are created with a specified source, destination, and type. When a user attempts to create a pipe, the kernel searches for an existing pipe of the same type. If no pipe of that type exists on the current node, the kernel creates a pipe and returns a new handle to the user. If one already exists, a new handle is created to the already existing pipe. Since a pipe by definition has one or more sources and only one destination, it is illegal to create a new pipe handle with a different destination from the currently existing one. The type argument can be used if the user needs to create distinct pipes with different destinations. It is legal and perfectly reasonable to have

more than one source for a pipe. If both the source and destination of a pipe are the current node, it is a local pipe. If the source (destination) of the pipe is the current node, it is a user-fillable (user-drainable) pipe. If neither the source nor the destination of the pipe is the current node, it is a store-and-forward pipe. The user is not allowed to forward a message to the same node from which it was received. The source and destination must either be the current node or a neighbor. Finally, the user can specify the buffer size of each pipe buffer. Once a pipe of a certain type has been created, any additional create call of the same type on that node must have a buffer size equal to the existing buffer size. Pipes between adjacent nodes have the restriction that the destination node must have a buffer size as large or larger than the source node.

4.3 Data Flow in a Pipe

Once a pipe has been created, the user can fill it with data. The data is not actually transmitted until either the user fills a buffer or explicitly forces a flush of the pipe by using a system call. Each individual write is guaranteed to be atomic unless its size exceeds the local buffer size. Multiple create calls can create a pipe where data is joined from more than one source. These sources can be any combination of locally filled pipes and forwarding pipes. A pipe with multiple fillers can be thought of as the inverse of a broadcast, in that a broadcasted message is sent from one source to several recipients, while a pipe allows several sources to funnel data to one recipient. Once two or or more messages are concatenated into one buffer, they will remain concatenated until read by a user. Since the data is mixed on a buffer-by-buffer basis, these atomic write properties are preserved.

4.4 I/O on a Pipe

Data is read from or written to a pipe using the same read and write calls as with message passing, where the source or destination

is the pipe handle returned from a pipe creation operation. The kernel can distinguish pipe handles from other types of source or destination descriptors. A new option, *dispose*, is available only with the read operation of a pipe. If this option is specified, the requested number of bytes is discarded from the pipe. A fast drain on a pipe is provided if the user requests -1 bytes with the *dispose* option. In this case the entire contents of the local buffers of the pipe will be discarded. If the *dispose* option is requested in a write operation or in a read operation on something other than a pipe, it is ignored. The *synchronous* option is ignored for any operation involving a pipe.

All other features of the Multifaceted I/O system are available with pipes. However, the *unbuffered* option behaves differently than in the message passing paradigm. In the write case, it is used to insert already full buffers into the queue of buffers waiting to be transmitted. After the buffers have been sent, they are freed and available for reuse by the user program. In the case of read, the user provides empty buffers that are added to the list of buffers awaiting input. If a user requests an unbuffered write to a pipe, the user-supplied buffers are temporarily linked into a queue of buffers to be sent. If the user did not specify the *asynchronous* option, the write call blocks until the data has actually been sent. If the user did request the *asynchronous* option, the kernel behaves the same as it does in the message passing scheme. Similarly, if a user requests an unbuffered read from a pipe, the user-supplied buffers are temporarily added to the list of buffers available to receive data. Temporary user-supplied read buffers are used before permanent buffers, but after any other temporary user-supplied buffers, previously given. An unbuffered read from a pipe does not return until the buffer has been filled, unless the user specified the *asynchronous* option.

4.5 Using these Features

Manual pages for the pipe management routines are provided in Appendix B.

5 Files

The new features in the Multifaceted System support file system access by nodes of the cube. System calls are provided to manage the communication between the node requesting action and the node that has direct access to the file. The open system call returns a file descriptor which can be used in any of the read/write system calls.

5.1 Normal Mode (Non-Sequential)

Files can be opened for read, write, or read/write. A read system call using the file descriptor will generate a message requesting the specified number of bytes from the file. All book keeping involving the location of the seek pointer is kept on the node that has direct control of the file. A write system call will send the data supplied to the node which will write the data into the file. The user can also move the read/write pointer by using the seek system call using the file descriptor.

If the *synchronous* option for normal writes is requested, the write does not return until the receiving side can determine the outcome of the write request.

5.2 Sequential Mode

An optimization is provided if the user can guarantee that access to the file (either read or write) will be sequential. If the user specifies the *sequential* option for a file opened for read or write, the kernel will establish a pipe. If a file is opened sequentially for reading, the kernel will generate read requests to fill the local buffers of the pipe. This way, the data will be available more readily to the user program. If a file is opened sequentially for write, data is buffered locally. When buffers are filled, they are sent to the recipient node asynchronously. Again, this allows the user program to operate more

efficiently.

Since the implementation of this optimization involves the use of pipes, the *synchronous* option is ignored on any file opened sequentially.

5.3 Using these Features

Manual pages for the file management routines are provided in Appendix C.

6 Uses for the Multifaceted System

The following sections describe some typical uses for the features designed in this document. These examples are in no way intended to be a complete list of possible uses for the Multifaceted I/O System. They should provide the user with some ideas for ways of using these features.

6.1 Asynchronous I/O

Under the current I/O scheme for both Simplex and Vertex, data is copied to or from a system buffer before the I/O call returns. With this new option, the user can issue a truly asynchronous I/O call. If the data is available or buffer space is available, the kernel will perform the requested transaction immediately. If, however, the data is not available or buffer space is unavailable, this new option allows the kernel to return to the user immediately. The user must pass an additional argument whenever using the *asynchronous* option. This argument tells the kernel how it is to notify the user when the transaction has actually taken place.

A natural use of asynchronous I/O is with unbuffered transmissions

which inherently involve a rendezvous between sending and receiving processes. A rendezvous is a source of considerable potential delay for one of the parties. By allowing an unbuffered transmission to occur asynchronously, the process invoking it avoids waiting for the rendezvous to occur.

6.2 Non-Blocking I/O

Non-blocking I/O is a very simple but useful feature. The current I/O system has a mechanism for simulating a non-blocking read by calling **ntest** before **nread**. There is no facility for the user to prevent blocking on write due to insufficient buffer space. Not only is the mechanism for simulating non-blocking read inefficient due to the extra entry into the kernel, it is imperfect under Simplex. Under Vertex, there could only be one process on each node, but under Simplex there could be more than one. Therefore, a user could issue an **ntest** system call followed by an **nread** system call. If a second process issued a read call between the **ntest** and **nread** of the first process, the **nread** could block. This *non-blocking* option fixes all of the deficiencies with the old mechanism. The kernel is entered only once, with no redundant searching through messages. The user has the ability to prevent blocking of any type both on read calls and write calls.

A natural use of non-blocking write occurs with producer/consumer relationships. If the producer wishes to produce as much as possible so as to maintain a high backlog for the consumer's benefit, there is a risk that the message buffers will be exhausted causing the producer to wait. With non-blocking I/O, the producer can react to this event by choosing to perform other work instead of waiting, and can retry the output in the future. Alternately, if it is not convenient to retry, an asynchronous write can be issued at that point, and the completion of that write could trigger the producer's return to its normal mode of writing.

6.3 Fully Synchronous I/O

Fully synchronous I/O can be used to transmit data while at the same time providing synchronization for the user's program. However, another use for this option involves file operations. Since "normal" message passing is weakly synchronous, i.e. the call returns when the data has been copied into a system buffer, the user has no way of knowing when and if the data finally reaches the file system accessible by the node that receives the data. The user can request the *synchronous* option on writes to files. Using this option, the write call will not complete until the node with direct access to the file, can return to the writer the results of the write.

Combining the asynchronous and the fully synchronous options in sending a buffered message allows an algorithm to send the message without waiting for it to be received, while arranging notification when the message is read by the receiving process. In this way, the sender can keep track of the backlog of unread messages on the receiving side.

6.4 Unbuffered I/O

The typical use for the *unbuffered* option is when sending very large messages. In the standard message passing paradigm, messages are copied three times: from the user's buffer to the kernel buffer, from the kernel buffer on the sending node to the kernel buffer on the receiving node, and from the kernel buffer back to the user's buffer. With the *unbuffered* option, the data is copied only once: directly from the user's buffer on the sending side to the user's buffer on the receiving side. The overhead of performing the rendezvous for unbuffered I/O is not worth the expense for small messages. However, as the number of bytes to be copied grows, it becomes more economical to synchronize the sender and receiver and only copy the data once.

6.5 Synchronization

Since the *unbuffered* option must perform a rendezvous with the sending and receiving sides before any data is transmitted, it can be used for synchronization between neighboring nodes. One side would write a zero-length unbuffered message while the other side would read a zero-length unbuffered message. Neither could proceed until both were at the synchronization point.

Alternatively, the *synchronous* option could also be used to send a zero-length message to provide synchronization of the user program.

6.6 Broadcasting

Some algorithms such as Quinn's Quickmerge, [Quinn88] require one node to broadcast some data to all other nodes in the cube. Using a tree-based broadcast, it is possible to perform this task in a time dominated by propagating a message $d-1$ hops. This ordinarily requires $2d-2$ memory-to-memory copies and $d-1$ node-to-node transmissions. With a kernel-based broadcast, unnecessary copying can be avoided yielding a time corresponding to just 2 memory-to-memory copies and $d-1$ node-to-node transmissions. For the NCUBE/1, this would be two to three times faster.

6.7 Gather/Scatter

A typical use for gather and scatter functions is when manipulating matrices. A matrix is generally represented by a two dimensional array with one dimension being the row and the other being the column. If the rows of the matrix are contiguous, the columns cannot be. If the algorithm being used needed a column sent to another node, the user would have to first gather the message into a temporary buffer. Then that buffer would be copied into a system buffer. On the receiving end, the user would read the message into a user buffer, then scatter the message back to the matrix. This double

copying causes a loss in efficiency. Using these routines, the user would have the kernel gather the message from the user's matrix using a start address of the start of the array, a size of each element of the array, the stride which would be the number of rows multiplied by the size of each element, and the repeat count which would be the number of columns. On the receiving side, the kernel would scatter the message directly to the user's memory. Although the typical case would have matching gather and scatters, it is perfectly legal to gather a message on the sending side without scattering it on the receiving side. This would be useful for sending a matrix while transposing it. Likewise, a user could scatter a message that was not gathered.

6.8 Using a Pipe as a Data Funnel

Some algorithms require data to propagate through the dimensions until all data is accumulated on a single node. In the case where all data is transmitted to the neighbor node in the next lowest dimension, all data will eventually arrive at node 0. Each node except the final node would create a pipe in order to locally fill. Each intermediate pipe would also create a store-and-forward pipe for each neighboring node in a higher dimension. Since nodes in the lower dimension subcubes will be accumulating more data, the user might want to make the buffer size of the pipe proportionally larger for these nodes.

6.9 Other Pipe Uses

Another use for the pipe mechanism is for a producer/consumer relationship either between two nodes or among several nodes each of which consume some data and produce some other data which is then passed on to the next consumer in the line. In this case, the user would create a user-drainable pipe to receive data, and a user-fillable pipe to send the modified data along to the next node. The pipe mechanism provides flow control so that the producer will fill only as

many buffers as have been allocated before becoming blocked. In the current message passing paradigm, the producer would only become blocked when it had exhausted the message passing buffers on the local node, possibly also exhausting the buffers on the destination node or an intermediate node.

This mechanism can also provide an efficient method of batching smaller messages into one large message. Since the overhead of message passing is high, there is an incentive to combine several smaller messages into one large message. The user can fill a buffer in whatever small increments are convenient, but only incur the expense of message passing when a buffer has been filled, or the user explicitly requests a flush.

7 Summary

The Multifaceted Communication System for the NCUBE is intended to serve three purposes: first, provide the user with functionality that currently does not exist in any form. Second, provide easy and efficient mechanisms for useful features that users currently provide for themselves often at the expense of running time and/or development time. Finally, provide mechanisms that reduce both the overhead and the total cost of sending data from one node to another. Every feature proposed in this paper meets at least one of the three objectives.

The motivation for the extension of the message passing paradigm originated in [Maggio89]. This work involved sorting algorithms for the NCUBE. The implementation and modification of an existing algorithm demonstrated the lack of operating system support for designing and implementing efficient, data intensive algorithms on the NCUBE.

The motivation for the creation of the pipe paradigm came from a need for the kernel to accumulate debugging and logging information from all nodes in the cube to one central location.

A Read and Write System Calls

The input and output routines of the Mulitfaceted System are described below using individual manual pages for the C-language interfaces. (Fortran forms also exist.) Most features are invoked through bits in an argument of flags. Not all combinations of flags are legal nor is every feature available with every call. The following flags can be given:

ASYNCH – asynchronously send/receive message.
SYNCH – truly synchronously send data (files and messages).
NONBLOCK – non-blocking I/O aborts if the call would block.
BRDCST – broadcast message to a subcube.
NOTME – with BRDCST, do not include sending node in broadcast.
UNBUF – send/receive directly from user's buffer without copying to/from kernel buffer.
PARTIAL – receive only part of a message.
DISPOSE – with pipes, throw away specified number of bytes.
WRITE – open a file for writing.
RDWR – open a file for both reading and writing.
SEQUENT – open a file for sequential only mode.

When the ASYNCH option is used, an "ACTION" structure is given to tell the kernel what action to perform when the operation is finally completed, so that the calling process can react to the completion. This action structure consists of a code plus one or more arguments. The list of actions that can be invoked is somewhat open-ended in that an implementation is free to provide any set of actions. We envision the need for at least the following ones: incrementing or decrementing a counter in the user program; adding or subtracting the number of bytes transmitted to a variable in the user program; sending a signal to the calling process; storing the value(s) that would have been returned in a synchronous form of the call into an indicated variable in the user program.

A.1 WRITE

write – send a message to one or more nodes, pipe, or file

C SYNOPSIS

```
int write(buffer, nbytes, dst, flags, async (opt.))
char *buffer;
int nbytes, flags;
HANDLE dst;
ACTION *async;
```

DESCRIPTION

If `dst` indicates a message, `write` creates a message with the given destination whose contents are `nbytes` bytes beginning at the location pointed at by `buffer`; if `dst` indicates a pipe or file, the data is written to the pipe or file. The `flags` argument determines which options are used. The PARTIAL flag is ignored. If the handle is a pipe or file, the BRDCST, NOTME, and PARTIAL bits are ignored. The `async` argument is used as described above if the ASYNCH bit is set.

The return value will be 0, or negative if erroneous arguments were given.

A.2 READ

read – receive a message from a node, pipe, or file

C SYNOPSIS

```
int read(buffer, nbytes, src, flags, async (opt.))
char *buffer;
int nbytes, flags;
HANDLE src;
ACTION *async;
```

DESCRIPTION

`Read` reads the first message whose source is compatible with the source pointed to by `src` structure. If any of the source, process, or type contained in the handle is -1, then any source, process, or type is compatible, otherwise only the indicated source, process, or type is compatible. The message or at most `nbytes` is stored in the indicated buffer. The `flags` argument is the same as for `write` with the following exceptions: the BRDCST, NOTME, and SYNCH bits are ignored, if the PARTIAL flag is set, any remaining bytes will not be discarded. The message length will be decreased by the number of bytes actually read. If the handle is a pipe or file, the BRDCST, NOTME, and PARTIAL bits are ignored. The optional `async` argument is only used if the ASYNCH bit is set. It is a pointer to an ACTION structure, which controls what to do when the read completes. Currently, the only action supported is to store a value into an address.

The return value is negative if erroneous arguments were given. Otherwise, the return value is the size of the message (not necessarily the number of bytes actually read). The user can determine the number of bytes actually read by comparing the return value with the number of bytes requested. If the return value is less than or

equal to the number of bytes requested, the return value is the number of bytes read. Otherwise, the number of bytes requested is the number of bytes read. In the case of a message, the fields in the handle indicated by `src` are filled in with the actual source, process, and type of the message (unless the ASYNCH flag is used).

A.3 TEST

test – test for a message from a node, pipe, or file

C SYNOPSIS

```
int test(src, flags)
HANDLE src;
int flags;
```

DESCRIPTION

If `dst` indicates a message, `test` is similar to `read` but instead of reading a message it returns an indication of the presence of a message compatible with the given source. If there is no compatible message (synchronous `read` would block) then `test` returns a negative return value. If there is a compatible message, `test` stores the message's actual source and type just as `read` would, and returns the message's length as its return value. The routine can be used either as a part of a a non-blocking read emulation or to determine messages' lengths prior to reading them.

If `dst` indicates a pipe or a file, `test` returns the number of bytes that can be read without causing the calling process to become blocked. Thus it can be used to determine the amount of data in local input buffers for pipes and files.

A.4 WRITEV

writev – send a message to node(s), pipe, or file using an I/O vector

C SYNOPSIS

```
int writev(iovec, dst, flags, async (opt.))
IOVEC *iovec;
HANDLE dst;
int flags;
ACTION *async;
```

DESCRIPTION

`Writev` is identical to `write` except instead of passing buffer and size arguments, `iovec` is passed. This argument is a linked list of iovec structures, each with five fields:

next – pointer to next iovec structure or null if last
location – starting address from which to gather data
size – number of bytes to fetch for each repetition
stride – number of bytes separating the start of each block of bytes
repcount – number of repetitions.

A.5 READV

readv – receive a message from a node, pipe, or file using an I/O vector

C SYNOPSIS

```
int readv(iovec, src, flags, async (opt.))
IOVEC *iovec;
HANDLE src;
int flags;
ACTION *async;
```

DESCRIPTION

`Readv` is identical to `read` except instead of passing buffer and size arguments, `iovec` is passed. This argument is a linked list of iovec structures, each with five fields:

next – pointer to next iovec structure or null if last
location – starting address into which to scatter data
size – number of bytes to store for each repetition
stride – number of bytes separating the start of each block of bytes
repcount – number of repetitions.

A.6 GATHER

gather – gather data and write it

C SYNOPSIS

```
int gather(st_addr, size, stride, repcount, dst, type)
char *st_addr;
int size, stride, repcount;
int dst, type;
```

DESCRIPTION

`Gather` is a special case of the `writev` system call with only one iovec structure and the destination and type specified as arguments rather than included in a handle.

A.7 SCATTER

scatter – read data and scatter it

C SYNOPSIS

```
int scatter(st_addr, size, stride, repcount, src, typep)
char *st_addr;
int size, stride, repcount;
int src, *typep;
```

DESCRIPTION

`Scatter` is a special case of the `readv` system call with only one iovec structure and the source and type specified as arguments rather than included in a handle.

A.8 BROADCAST

broadcast – broadcast a message to a subcube

C SYNOPSIS

```
int broadcast(buffer, size, dst1, dst2, type)
char *buffer;
int size, dst1, dst2, type;
```

DESCRIPTION

`Broadcast` provides an easy interface for a basic broadcast. `Buffer` and `size` are the same as in the `write` system call. The two destination arguments, `dst1` and `dst2`, are the base node and mask respectively. The `type` argument is the type of the message to send. By default, the NOTME bit will be turned on, so even if the current node is included in the subcube, it will not receive a copy of the broadcasted message. If `dst1` is -1, the current node is used as the base node.

B Pipe Management System Calls

The following routines are used to create, destroy, and modify pipes. Pipes can be created and used on a local node or to or from a nearest neighbor. A pipe can have only one destination but can have one or more sources from the local node or neighbors. Once a pipe has been created, the user can use the various input and output routines using the pipe handle returned by `pipe_create` as the destination or source.

B.1 PIPE_CREATE

pipe_create – create a pipe

C SYNOPSIS

```
PIPE_HANDLE *pipe_create(src, dst, type, buffer, size, nbufs)
int src, dst, type;
char *buffer;
int size, nbufs;
```

DESCRIPTION

`Pipe_create` creates a pipe with a source node of `src` and a destination of `dst` of type `type`. If a pipe already exists with this type, a new port to the existing pipe will be created and returned to the user. If the pipe does not already exist, the pipe will be created. The `buffer` is either NULL or a pointer to a user-supplied buffer. If the user does not supply a buffer for the pipe, the system will try to allocate one. The `size` and `nbufs` arguments determine the size of each buffer and the number of buffers respectively. If the system is unable to allocate enough buffer space or the user tries to create a pipe with more than one destination, a NULL will be returned. Otherwise, a pipe handle will be returned. This handle is used to read and write data from and to the pipe or perform other pipe management functions.

B.2 PIPE_DELETE

pipe_delete – delete a pipe after sending all data

C SYNOPSIS

```
int pipe_delete(handle)
PIPE_HANDLE *handle;
```

DESCRIPTION

`Pipe_delete` flushes the specified pipe and marks it to be removed when all data has been transmitted. This routine returns -1 if the pipe does not exist, NULL otherwise.

B.3 PIPE_DESTROY

pipe_destroy – destroy a pipe discarding all data still in pipe

C SYNOPSIS

```
int pipe_destroy(handle)
PIPE_HANDLE *handle;
```

DESCRIPTION

`Pipe_destroy` deallocates all buffer space associated with this pipe and removes the pipe itself. This routine returns -1 if the pipe does not exist, NULL otherwise.

B.4 PIPE_FLUSH

pipe_flush – flushes the current buffer

C SYNOPSIS

```
int pipe_flush(handle)
PIPE_HANDLE *handle;
```

DESCRIPTION

`Pipe_flush` queues the current partially filled buffer to be output. This routine returns -1 if the pipe does not exist, NULL otherwise.

B.5 PIPE_EXPAND

pipe_expand – add buffers to the existing pipe

C SYNOPSIS

```
int pipe_expand(handle, buffer, nbufs)
PIPE_HANDLE *handle;
char *buffer;
int nbufs;
```

DESCRIPTION

`Pipe_expand` adds `nbufs` buffers to the already existing pipe specified by `handle`. The argument `buffer` is either NULL or a pointer to a user-supplied buffer. Once a pipe has been created, the buffer size is fixed. If the system is unable to allocate enough buffer space or the pipe does not exist, -1 is returned. Otherwise, NULL is returned.

B.6 PIPE_SHRINK

pipe_shrink – remove buffers previously added to an existing pipe

C SYNOPSIS

```
int pipe_shrink(handle, buffer, nbufs)
PIPE_HANDLE *handle;
char *buffer;
int nbufs;
```

DESCRIPTION

`Pipe_shrink` removes `nbufs` buffers to the already existing pipe specified by `handle`. The argument `buffer` is either NULL or a pointer to a user-supplied buffer. The user can only remove user-supplied buffers in the same quantity as were added by a matching `pipe_expand` call. If there are not enough buffers left to deallocate, or the pipe does not exist, -1 is returned. Otherwise, NULL is returned.

C File System Calls

The following system calls are used to open and close files, establishing a communication link between the node that has direct access to the file and another node in the cube. A system call to seek within an open file is also defined. Once a file has been opened, the user can use the various input and output routines using the file handle returned by `file_open` as the destination or source.

C.1 FILE_OPEN

file_open – open a file

C SYNOPSIS

```
FILE_HANDLE *file_open(node, type, path, arg1, arg2, buffer, size, nbufs);
int node, type;
char *path;
int arg1, arg2;
char *buffer;
int size, nbufs;
```

DESCRIPTION

`File_open` opens a file denoted by `path` controlled by node `node`. The `type` argument indicates whether to open the file read, write, or read/write. The two arguments `arg1` and `arg2` are passed directly to the operating system controlling the file system. If the file is opened read or write, type can also include the `sequential` option. The user is not allowed to seek in a file opened sequentially because the kernel creates a pipe to use to speed up the access of data. In the sequential case, the three `pipe_open` arguments `buffer`, `size`, and `nbufs` must be provided.

If the file is successfully opened, a file descriptor is returned to the user. This file descriptor can be used as the source or destination in message passing system calls. A NULL is returned if the file could not be opened.

C.2 FILE_CLOSE

file_close – close a file

C SYNOPSIS

```
int file_close(file_descr);
FILE_HANDLE *file_descr;
```

DESCRIPTION

The `file_close` system call closes the file associated with the specified file descriptor.

C.3 FILE_SEEK

file_seek – seek in an open file

C SYNOPSIS

```
int file_seek(file_descr, position, whence);
FILE_HANDLE *file_descr;
int position, whence;
```

DESCRIPTION

The current location of the seek pointer of the open file described by `file_descr` is adjusted. If `whence` is 0, `position` is the absolute address to seek within the file. If `whence` is 1, seek to the current position plus the offset `position`. If `whence` is 2, `position` is an offset from the end of the file. This system call is illegal for any file that has been opened with the `sequential` option.

If the call is successful, the current absolute position in the file is returned. Otherwise, a -1 is returned if the file handle is invalid or refers to a file that had been opened sequentially.

D Handle Creation Routines

The following routines can be used to build handle structures for using the message passing routines for regular messages.

D.1 HANDLE_FULL

handle_full – make a handle for use in the message passing system calls

C SYNOPSIS

```
MESS_HANDLE *handle_full(loc, node, type, proc, mask)
MESS_HANDLE *loc;
short node, type, proc, mask;
```

DESCRIPTION

This routine creates a full handle with all possible fields used. The first argument `loc` is either NULL or the address of a user supplied structure. If this argument is NULL, a static buffer will be created and returned to the user. Any other call to this routine or other handle creation routines will overwrite this buffer. If the user wishes to use multiple handles, then a buffer should be supplied as the first argument. The `node` argument is the source or destination node. The `mask` argument is used for broadcasting a message. The `proc` argument specifies a particular process, and the `type` argument specifies a particular type.

D.2 HANDLE_NODE

handle_node – make a simple handle with only node and type

C SYNOPSIS

```
MESS_HANDLE *handle_node(loc, node, type)
MESS_HANDLE *loc;
short node, type;
```

DESCRIPTION

This routine is the same as `handle_full` with only the node and type arguments.

D.3 HANDLE_PROC

handle_proc – make a handle with node, type, and process

C SYNOPSIS

```
MESS_HANDLE *handle_proc(loc, node, type, proc)
MESS_HANDLE *loc;
short node, type, proc;
```

DESCRIPTION

This routine is the same as `handle_full` with node, type, and process arguments.

D.4 HANDLE_BRDCST

handle_brdcst – make a handle with node, type, and mask for broadcast

C SYNOPSIS

```
MESS_HANDLE *handle_brdcst(loc, node, type, mask)
MESS_HANDLE *loc;
short node, type, mask;
```

DESCRIPTION

This routine is the same as `handle_full` with node, type, and mask arguments.

E Implementation Notes

While designing the user interface of the new functionality, the authors have attempted to insure that the features described can be implemented in some reasonable way. Implementation details have been thought out when it was not obvious how to provide the functionality required. The following sections describe those implementation details.

E.1 Underlying Mechanisms

Several underlying, kernel mechanisms have been envisioned to support the Multifaceted Communication System. These mechanisms are required as building blocks for the proposed I/O features as well as future features and paradigms.

The first of the features is a new notion of ACTION. Actions can be thought of as lightweight processes that operate in the context of the kernel. These actions either occur immediately, in which case the data structure describing the action is deallocated, or they occur at some later point in time. In the latter case, the action "sleeps" on some event as a regular process would.

Remote actions are built from local actions. A node can send a remote action to another node. The kernel must guarantee that any node can send an action to any other node at all times. If the remote action does not require any resources (e.g. memory or message buffers), of the receiving side (excluding the resource of the message itself), the recipient must perform the action as soon as is possible. If the remote action does require some resource (such as the allocation of a local action with associated data structures), the recipient node can either acknowledge the remote action guaranteeing that it will be performed at a later time when or if the necessary resource becomes available, or it may ignore the remote action, discarding the message completely. Upon receipt of an acknowledgement, the sending node can discard its record of the needed remote action.

Otherwise, the sending node must be prepared for the possibility that the remote action will be discarded. Note that if both positive and negative acknowledgements are desired, they can be guaranteed since they do not require resources.

It is important to make remote actions that do not require resources fast as the kernel must guarantee that it will process them.

One future use of these local and remote actions is signal delivery and receipt.

Next, a basic mechanism of unbuffered, synchronous I/O for communication from neighbor to neighbor is needed. The current message passing paradigm already needs this mechanism. On top of the existing mechanism, unbuffered, asynchronous I/O is needed. There should be a clean, kernel interface that both I/O paradigms use at the lowest level as well as any other communication paradigm.

E.2 New Data Structures

Many abstract data structures have been mentioned in this document. Included in this section are the specifics of those data structures.

E.2.1 Handles for Message Passing

```
typedef struct mess_handle {
        short addr;                     /* Node Address */
        short mask;                     /* Mask for Broadcast */
        short process;                  /* Process ID */
        short type;                     /* Type of Message */
} MESS_HANDLE;


typedef union handle {
        MESS_HANDLE *m_handle;  /* Message Passing Handle */
        FILE_HANDLE *f_handle;  /* File Handle */
        PIPE_HANDLE *p_handle;  /* Pipe Handle */
} HANDLE;
```

E.2.2 Asynchronous Action Structure

```
typedef struct action {
        int type;                       /* Type of Action */
        long arg1;                      /* Optional Arg 1 */
        long arg2;                      /* Optional Arg 2 */
} ACTION;
```

E.2.3 I/O Vector Structure

```
typedef struct iovec {
        struct iovec *next;             /* Next Pointer */
        char *location;                 /* Start Address */
        int size;                       /* Size of Entity
        int stride;                     /* Distance Between Entities */
        int repcount;                   /* Repetition Count */
} IOVEC;
```

E.2.4 Kernel Control Block Structure

```
typedef struct controlblk {
        int flags;              /* I/O Flags and Options */
        IOVEC *iovec;           /* Pointer to I/O Vector */
        HANDLE addr;            /* Source or Destination Handle */
        ACTION *async;          /* Action Structure */
} CNTRLBLK;
```

E.3 User Requested Actions

The user may instruct the kernel to perform an actions when some type of asynchronous I/O completes. The simplest action is a type STORE which will set the location specified by arg1 with the value specified by arg2. A type of INDIRECT_STORE would cause the location pointed at by arg1 to be set to the value pointed at by arg2. Similar constructions could evoke counting up or down in units of one or of the number of bytes of data transmitted. A type of SEND_SIG would send a signal of type arg2 to the process arg1. A type of RETRIEVE would store the value that would normally be the return value in the location specified by arg1.

Ideally the user should be able to request more than one action; however this poses implementation problems since the user-supplied action block is most safely handled by coping it into the kernel before the call returns, and a fixed-size block can be accommodated most easily. (Note that if the block is a stack-allocated structure in the user process, then the original action block might have been deallocated by the time it is needed in the kernel.) An action block of fixed size will impose strict limits on the number of actions than can be specified. The sending of a signal provides a workable alternative that allows the user process to react in arbitrary ways.

E.4 Pipe Implementation

Some pipe_create calls will allocate buffers to be used to send or receive data for a particular pipe. These buffers will be linked together through one field of the pipe buffer structure. A second field will allow buffers to be linked together for transmitting or receiving of data. For this reason, messages are guaranteed to be contiguous only if they fit into one buffer. The mixing of data caused by multiple create calls and multiple fillers can only occur on a buffer-by-buffer basis. Messages will be intermixed within one buffer if the user allows a particular pipe handle to be shared among several processes on a node. If each process issues its own pipe_create call, then the data will not be mixed within buffers.

Buffers are linked together on a free list of unused buffers or a busy list of data to be sent or received. If the user requests the *unbuffered* option, the buffers will be supplied with the read or write calls. These user supplied buffers will be linked into the queue of buffers available for the specified pipe handle. They will be marked so that when the data is sent or received, these temporary buffers will be returned to the user rather than put on the pipe handle's free list.

A pipe_create call will only allocate buffer space on the local node. The kernel does not attempt to establish an I/O channel to the destination node until the user actually fills the first buffer (or flushes the first buffer) or requests data from a pipe. When the sending and receiving sides establish a connection, the receiving side informs the sending side how many outstanding buffers it has. The sending side guarantees not to send any more buffers than the receiving side can handle until the receiving side acknowledges the buffers have been read.

The originator of the I/O bridge attempts to establish a connection by sending a remote action to the other side. Either the source or destination node of a pipe can be the originator of the I/O connection. The kernel does not guarantee that it will accept the remote action. If it does not, it is the responsibility of the originator of the bridge to poll at reasonable intervals. If, however, the responding

side can accept the remote action, it will inform the originator that it will queue the remote action. The originator then is not required to poll. It can simply wait until the responding side is prepared to create the bridge.

Since the receiving side pre-approves a certain number of buffers to be sent, special care must be taken when destroying a pipe. A pipe cannot be immediately destroyed until the sending side knows that it can no longer send those already approved buffers.

References

[Krumme90] D. W. Krumme, A. L. Couch, B. R. House, and J. Cox, "The Triplex Tool Set for the NCUBE Multiprocessor, V.2.4," Tufts University (Jan. 1990).

[Maggio89] M. D. Maggio, "An Analysis and Modification of Quinn's Hypercube Quickmerge Algorithm," (Unpublished), Tufts University (May 1989).

[Quinn88] M. J. Quinn, "Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge," technical report PCL-88-13, Department of Computer Science, University of New Hampshire (March 1988).