MULTI: A shared memory approach to cooperative molecular modeling

Tom Darden, Pam Johnson and Howard Smith

National Institute of Environmental Health Sciences, RTP, NC, USA

A general purpose molecular modeling system, MULTI, based on the UNIX shared memory and semaphore facilities for interprocess communication is described. In addition to the normal querying or monitoring of geometric data, MULTI also provides processes for manipulating conformations, and for displaying peptide or nucleic acid ribbons, Connolly surfaces, close nonbonded contacts, crystal-symmetry related images, least-squares superpositions, and so forth. This paper outlines the basic techniques used in MULTI to ensure cooperation among these specialized processes, and then describes how they can work together to provide a flexible modeling environment.

Keywords: shared memory, parallel processing, molecular modeling

INTRODUCTION

The increasing complexity of molecular modeling needs has caused some researchers to abandon the monolithic approach, wherein one large executable process performs all processing of user requests, in favor of a system of cooperating specialized processes. One driving force in this development has been the emergence of sophisticated desktop multiprocessing operating systems, such as UNIX, which have made interprocess communication mechanisms like pipes, sockets, semaphores and shared memory available to the scientific programmer. Concomitant with this, the universal acceptance of the multiwindowed desktop metaphor has made the average user aware of the advantages of having available a collection of specialized processes that can exchange information.

In our environment, another important consideration is the need to integrate several existing molecular modeling programs. A major goal of our work was to provide a graphical front end for these programs, many of which are batch oriented. Modification of these programs for direct incorporation as subroutines in a monolithic modeling program seemed to be a project that would eventually overwhelm us, particularly since we had no control over the direction in which these programs might evolve. Indeed, there are often several variants of these programs in existence, which differ from each other in subtle ways. We felt that our modeling system should work equally well with any of the variants, so that researchers could choose their favorite from their own experiences. This effectively rules out the possibility of a single monolithic modeling program. Finally it should be noted that licensing restrictions would often prevent us from releasing a program that contained code written by other groups, whereas we could give away our code plus the "glue" to hook in other existing programs.

These considerations seemed to suggest the development of a mechanism which would allow separate executables to cooperate. One method of communication that has been used often is to pass information through files using a common protocol. This technique is straightforward to implement and is therefore preferable in a batch-oriented system. However, other than being slow it must be supplemented by some method of preventing read/write access collisions, to be used successfully by a group of processes. Another approach would be to use the pipes mechanism provided with Unix. This is considerably faster than files and is a safe and reliable method of communication. However, it requires that one of two communicating processes be the parent of the other. One goal of this project was to allow a user to connect to a pre-existing batch process that may have been started by another user, perhaps on a different processor. This effectively restricted our choice to mechanisms based on sockets or on shared memory and semaphores.

A message-based system using sockets or UNIX system V message queues seemed easier to implement than a shared memory-based system, and we had experience using VMS mail boxes in an early version of the system. We decided to use shared memory and semaphores to implement interprocess communications. Message-based systems seem to be ideal for systems having only a few processes, such as Quanta/Charmm,[1] or for visualization environments like ConMan, apE and AVS,[2] which create displays using a series of filters connected in a unidirectional dataflow graph. However, we envisioned a system having many specialized processes with completely symmetric access to the common data. (See Figure 1.) These processes should be able to come alive, connect to the system, contribute to the common data, and then die, at any time during the system life-span. As the amount of data and the number of processes grew, a message-based approach seemed to become increasingly difficult to design. In the first place, less memory space is required to share common data than to have private copies

Address reprint requests to Dr. Darden at the National Institute of Environmental Health Sciences, RTP, NC 27709, USA.
Received 10 October 1990; accepted 15 October 1990

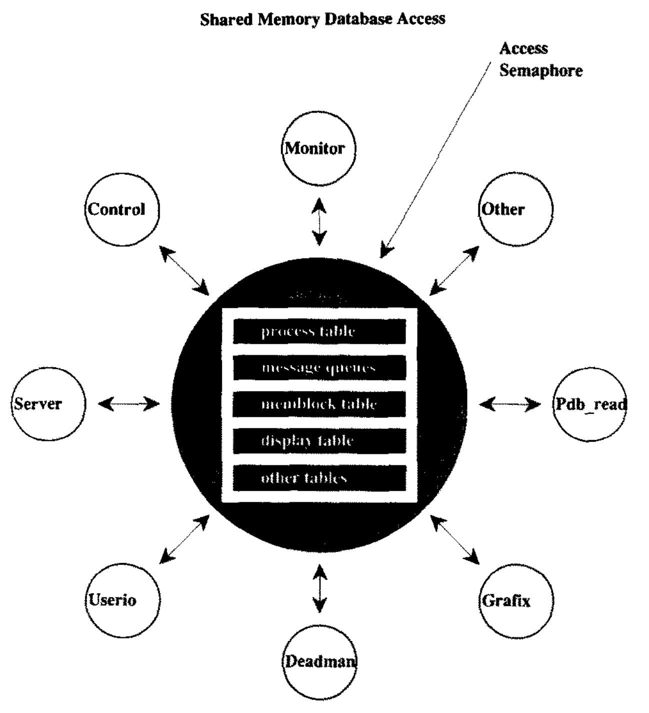

Figure 1. Central shared memory database and sample processes. The central database contains tables that are accessed symmetrically by the processes making up the system; one process at a time is allowed in the database, with access controlled by a semaphore

for each process. Second, it is inefficient to constantly transmit large amounts of data to many processes, whereas this potential bottleneck is eliminated if the data is shared. Third, maintaining the common data in a shared memory database that exists independently of those processes living at a given time, seems inherently more reliable than any method based on messages between the processes. Finally, using shared memory we could implement our own message passing scheme, which allows us to monitor and debug all traffic between processes because the necessary information is in the address space of the processes, rather than in system space (as happens with sockets). Indeed, the advantages of shared memory to the programmer are so compelling that many researchers are actively pursuing methods for extending shared memory across networks.[3]

Although sockets are currently necessary to communicate with foreign hosts, binary data usually need to be translated first to take care of incompatibilities; we decided to combine the translation and transmission tasks into one gateway process that communicates with local processes using shared memory. Another interesting approach to implementing global data sharing is the Linda tuple space protocol.[4]

USING SHARED MEMORY AND SEMAPHORES

The system V interprocess communication (IPC) facility consists of message queues, shared memory and semaphores. Shared memory comes in arrays of contiguous blocks called "regions." A process can create a new region by using the SHMGET system call. The region is given a unique identifier by the operating system. After that it exists as a system resource until some process deletes it. Any process can access this region (unless read/write permissions have been restricted) by attaching to it using the SHMAT system call. This routine returns an address, namely that of the beginning of the region, where it has been attached to the address space of the process. Its use is analogous to that of the MALLOC routine familiar to C programmers. Indeed, the operating system treats requests for shared memory identically to other dynamic memory requests.[5] As an example, if one stored the atomic coordinates of a molecule having N atoms in a shared memory region having identifier m, a program may access these coordinates using a C language assignment such as

```
PTR = (float*)SHMAT(m)
```

where *ptr* is a pointer to *float,* and m is an integer. After that, the coordinates of the ith atom are available for reading or writing using statements such as

```
X = PTR[3*i]
```

or

```
PTR[3*i+1] = Y
```

in other words, exactly as for any other array. We have found that reads or writes to shared memory regions happen fully as fast as those to local memory in the systems we have tested (IRIX, ULTRIX and SUNOS).

Of course, if several processes can simultaneously access the region, some mechanism must be used to synchronize activity. For this we have used the UNIX system V semaphore facility. These semaphores seem to have a reputation for being confusing.[6] However, in the simplest case a programmer can think of a semaphore as a token granting access to a resource. Typically, a program issues a system call to obtain the token at the beginning of a section of code where shared memory is to be accessed, and another to release the token at the end of the section. The process will be blocked ("sleep") at the first system call until the token is made available to it. More complex options are available. For example, one can use sets of semaphores to block a process until several resources are all available, or as counting variables to register the number of unread messages in a process's message queue. Semaphore operations are moderately fast. In the simplest case of two tiny processes doing nothing but accessing a critical section as described above, each can make about 1 000 passes per second through the section in the systems we have tested. Other timing information is given in Reference 7.

We should mention some important limitations to these IPC facilities. First, when more than one process attaches to a region of shared memory, the base address returned by SHMAT is typically different for the different processes. Therefore, it is not generally useful to have pointer variables in shared memory that store the addresses of other variables in shared memory, because these addresses are not generally valid for all such processes. This makes data structures such as trees and linked lists difficult to implement in shared memory, and we have tended to limit ourselves to arrays and structure arrays. Second, as with all long-lived resources, there are built-in limits to a program's use of IPC facilities. Typically, at most 100 regions can exist in a system at one time.[7] In many systems there are size limitations to regions, although this is not a problem on SGI 4D machines. Only a limited number of semaphores can exist in a system at one time: Sixty is a typical number.[7] Given additional IPC resources, it would be more efficient to subdivide the data in shared memory into many logically separate sections, each separately semaphored. The IPC limits can be relaxed by reconfiguring the kernel, although we have chosen to avoid doing this. Finally, it is up to the programmer to ensure that these resources are deleted when the last process exists.

BASIC COMMUNICATION STRUCTURES

The basic elements of the communication mechanism are a pair of semaphore arrays, a shared memory database and a collection of shared memory data blocks. (See Figure 1.) The shared memory database contains a process table having pertinent information about each cooperating process, a message queue table containing a message queue for each process in the process table, a memory table containing status information about generic shared memory data blocks, a table of all the display lists currently defined, and a memory block called "string space" that can be used to pass short messages. Additionally it contains a small (but growing) set of miscellaneous global tables. All interprocess communication uses this database mechanism. The semaphore arrays are used to control access to the shared memory. For example, a master semaphore controls access to the shared memory database. A process attempting to enter the database waits for this semaphore; that is, the process sleeps until the semaphore becomes available. Therefore, while one process is inside the database, no other interprocess communication can take place. In this way, even complex transactions are guaranteed to be atomic. This arrangement also allows a process to examine the state of the whole system, knowing that nothing is changing. The second array of semaphores implements a form of cooperative process blocking. That is, one process can block another by sending it a message to wait at a blocking semaphore. The recipient then sleeps, waiting for the sender to increment the recipient's blocking semaphore, which frees it to continue working.

The monitor process is responsible for the integrity of the central database while the MULTI system is alive, and for cleanup of all resources upon system shutdown. It sleeps in the background, awakening periodically to check for processes that have died, clearing the process table entries for any such processes and freeing up any resources they did not release before exiting. In addition it checks to see if any resources have been held for too long and kills the offending process. The monitor process is the only process that is allowed to override the normal blocking mechanisms. In this way bugs in individual processes do not result in system deadlock.

The control process provides a periodic graphical display of the contents of the central database, allowing the user to keep track of resource utilization by the system. (See Color Plate 1.) In addition the control process has a convenient menu interface for spawning and killing the many specialized processes making up the system, although all such processes can be started up independently from the command line.

MESSAGE PASSING MECHANISM

Each process has a message queue in the central database. Messages consist of a set of informational fields followed by a pointer to a linked list of shared memory data blocks. A process wishing to write a message to another process first acquires one or more data blocks by entering the database and finding some free blocks in the memory table, subsequently marking the data block's entries in the table to reserve exclusive access to them for itself. If no blocks are available, more can be created using SHMGET, and added to the memory table. Once it has reserved these blocks, no access collisions can occur, so it can exit the database, making it available to other processes. After filling the data blocks, the process links these blocks together into a message, re-enters the database and adds the message to the recipient's queue, marking the data block's entries in the memory table to assign exclusive access to the recipient. The process also increments a semaphore associated with the recipient's message queue, whose value gives the number of outstanding messages in the queue. Using this semaphore and the master semaphore, the recipient can poll its queue, returning if it is empty, or sleep until a message arrives.

The recipient reads a message by entering the database, where it reads the informational part of the message and acquires the identifiers of the shared data blocks. Depending on the message type, the recipient's dispatcher routine branches to an appropriate subroutine that "knows" how to read the data blocks. The necessary information describing the amount and format of the data in a data block is present in a header structure at the beginning of the block. Using this mechanism processes are able to implement a fairly generic data structure transfer, rather than having to write a new data handling subroutine for every new message. The header structure also automates data translation for transmission to a foreign host. After reading the message, the message recipient re-enters the database, marks the data blocks as free, and decrements the queue semaphore. A small variation of this method would allow several processes to share read access to messages.

SHARING DISPLAY LISTS

Despite the last comment, normally messages are sent by one process and consumed by another. By contrast, display lists are shared by multiple processes, each having read or write access to them. Display lists are structured as linked lists of shared memory blocks, each holding an array of display list instructions. As with other shared memory data blocks, the format of the display instructions is given as a header structure for every block of the list. Each display instruction has a part that is common to all display lists, and a part that varies with different lists. The common part has several fields, including instruction type (move, draw, and so forth), color, linewidth and atomic coordinates. There is also a utility field that can hold a floating point number written by one program to be read by others. For example, a molecular mechanics program could write the force magnitude for an atom in its utility field, where it would be read by a plotting program or a display editor that could modify the atom's color field according to a color ramp. The optional variable part can hold data like atomic names or residue numbers that are valid in some contexts but not in others.

The basic display lists in MULTI are the simple wireframe molecular displays, built by a molecular editor called the server process. The server maintains one or more buffers, each associated with a molecular display list and containing one or more molecules. Other processes can access the molecular editor features of the server using "macros," which are programs in a simple stack oriented assembler-type language. These macros are sent to the server and processed by the server's interpreter. For example, the PDB_READ process reads protein databank files, loading the data into an array of atom structures and deciding connectivity on the basis of distances. It then sends a macro to the server that initiates and controls a conversation between the processes. It first requests that a new buffer be made and that the necessary atoms be created, using the server's dynamic memory manager. Certain data common to all atoms, such as atomic coordinates, are sent over at this time. Next, the connectivity information is sent, from which the server builds a tree structure for each molecule. Next, PDB_READ requests the server to create a run time data structure that will contain variables specific to protein databank molecules, such as atom names and numbers and residue names and residue numbers, and to give a copy of it to each new atom. The data to fill these structures is sent next. Finally, the macro tells the server to build a display of the buffer. This display will hold the run time PDB data structure in its variable portion, so that it is available to any other process. User input to the server is handled by the userio process, which also journals all commands and output from other programs in the MULTI system.

Other than atoms, the server can work with vectors, which are geometric objects allowing transformations of trees and subtrees within buffers. The docker process makes editing requests of the server regarding vectors and manipulates them interactively under user control to modify inter- and intramolecular conformations in a controlled fashion.

DISPLAY LIST ACCESS

Access to a display list is controlled by a data structure in the central database called the "display token." Acquiring the token allows a process to make one traversal of the display list. Each process has a priority rating that determines the order in which access is granted to the list. For example, processes that do not need to read the display lists are given high priority so that they traverse early in line, while processes like grafix, which do not edit the list, are given low priority so that they traverse last or nearly last. Prioritizing processes is a simplistic approach to ensuring the correct display of data in comparison with the dataflow execution graphs used by ConMan and the other visualiza-

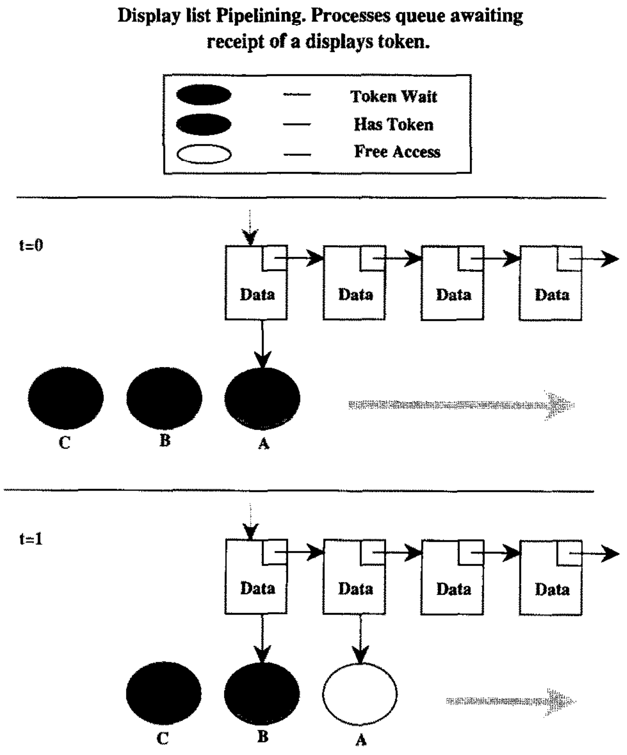
Display list Pipelining. Processes queue awaiting receipt of a displays token.
Token Wait
Has Token
Free Access
t=0
Data
Data
Data
Data
C
B
A
t=1
Data
Data
Data
Data
C
B
A

Figure 2. Sample processes A, B, *and* C *are shown accessing a display list, which is made up of a linked list of data blocks. Access to the display list is through receipt of a display token. Processes proceed sequentially through the data blocks of a display list. One process at a time is allowed in a block. Data access in block 2 by process* A *can proceed in parallel with access in block 1 by process* B, *allowing for speedup on multiprocessors*

tion environments mentioned above. However, it has served our needs so far. In addition, it does not seem difficult to extend the model to a more dynamic user-controlled execution ordering.

A process wishing to traverse the display list first looks to see if it has the token. If it does, it attempts to access the first data block of the display list. If the process does not have the token, it sleeps by decrementing its blocking semaphore. After acquiring the first block, it gives the token to the highest priority process that has not yet received it, waking that process if necessary by incrementing its blocking semaphore. The last process to acquire the token passes it to the highest priority process, which ensures continuous display traversal. Only one process at a time is allowed into each data block of the display list. Processes sleep, if necessary, using the blocking semaphore mechanism while waiting for the process ahead of them in line to awaken them and give them the data block. In addition to preventing read/write conflicts, this restriction eliminates the complex deadlock situations we encountered in earlier designs. Finally, in shared memory multiprocessors this method of pipelined display access, illustrated in Figure 2, allows for efficient parallelism.

The deadman process is responsible for preventing deadlock among the set of processes engaged in display traversal.

Deadlock cannot happen if a set of processes is already traversing one or more display lists, using the token-passing mechanism explained above. However, complex deadlock states can occur while trying to alter the pattern of traffic, such as when a process joins or leaves the group of processes accessing some display list. To prevent this, processes change their display list attachments by making requests of the deadman process. The deadman forces all processes currently accessing display lists to leave them and sleep while it sets up the new traffic pattern. Afterwards it starts the traversal off correctly by waking the processes, passing the token for each display list to the highest priority process attached to it. This technique for changing the attachments is fairly rapid.

SPECIALIZED PROCESSES

There are several other specialized processes that can read or modify display lists. One of the fields in the constant part of a display instruction is a bit field, which allows one to group atoms into sets. The set-builder process can group atoms into sets by user specified tests on the values of various fields in the display instructions, such as PDB data types if they have been defined. Because the variable names and types for each field in the display instruction are given in the header information of each block of the display list, the set builder does not need to have *a priori* knowledge about them. Complex sets can be built up using the set operations union, intersection and complement. The set-builder uses a bit stack for each atom to implement these operations. Therefore, operations that are used to define a set can be journaled automatically into postfix-ordered scripts by the set builder. These scripts can be replayed later in analogous situations. Alternatively, they can be written by hand and simply read in. For example, the script to define a peptide backbone reads

ATOMNAME CA = ATOMNAME C
= ATOMNAME N = ||

Once the sets are defined, the display editing process can be used to modify colors or linewidths for the atoms in a set. One can select more complex and expensive displays, such as the lighted solid mode, for a subset of the atoms. One can also assign labels to sets of atoms; complex, user defined labels can be built from one or more fields in a display list instruction using C language PRINTF formatting rules. Finally, the display editor can be used to mask a set of atoms out of a display list. This is especially useful for large datasets, such as proteins in water baths, where the water molecules can be put into a set and temporarily left out of the display to speed up display traversal.

Other types of display lists are created by a variety of specialized processes. For example, the *p*-ribbon process generates ribbons showing protein secondary structure using an algorithm described by Carson and Bugg.[8] The *n*-ribbon process generates a smooth rigorously determined global axis for curved nucleic acids.[9] The Connolly surface program,[10] produces a set of points showing solvent contacts. The bump-checker produces a set of dashed lines showing close interatomic contacts during docking or contacts between two sets of atoms, such as hydrogen bond donors and acceptors. Finally, there is a process that reads generic user defined display instructions described in an ASCII file and loads a display list.

The rms and crystal processes produce transformation matrices that are applied to a display list. The rms process produces optimal rigid superpositions of two atom sets using the algorithm of McLachlan.[11] The resulting distances between corresponding atoms are loaded into the utility field of their move instructions, for use by other programs. The crystal process produces symmetry related images of a molecule in neighboring unit cells.

Another process that can modify molecular display lists is the movie process, which reads in molecular dynamics trajectories. It advances time continuously, forward or backward, under mouse control, linearly interpolating atomic coordinates between successive discrete frames of the trajectory file, and writes the resulting coordinates into the display list. This provides a smoothly changing movie display. Alternatively the movie process can advance time in discrete steps. Most of the specialized processes in MULTI can be made to operate in a loop controlled by this time-stepper using the protocol for display list access. A recorder process can then save the successive frames in a binary file for playback. Because the display list format is given in the header structure for each block, this record/playback mechanism is generic.

The grafix process reads and executes the display lists discussed above. It opens a window in which a user selectable subset of the available display lists can be viewed. Querying and monitoring of geometric data is also done by the grafix process. Display of molecules and other kinds of display lists can be restricted using a display mask that is compared against the display list bit field for each instruction. This mask can be rapidly altered under user control by toggling function keys. In this way, for example, the set of backbone atoms in a protein and their complement, the sidechain atoms, can be toggled on and off. The water molecules in the water bath can also be handled in this way.

More than one instance of the grafix process can exist in an active system. Each such process has its own independent subset of attached display lists and its own viewing parameters, so that several simultaneous views of one or more display lists are possible. Other attributes, such as RGB color definitions of the color index field in the display instructions, can be modified for one window relative to another. (See Color Plate 1.) It is also possible for the grafix processes to share these data so that, for example, the relationship between their viewing parameters remains constant. In this way a display list can be looked at from more than one view while it is being rotated under user control. Side-by-side stereo views can also be implemented in this way. The screendump process reads display lists as well as the viewing parameters of a grafix process and does the necessary transformations and clipping in software to produce high resolution color postscript output of the grafix process's window contents.

DISCUSSION

A commonly expressed wish within the molecular modeling community is for a means to integrate the many programs available. Some researchers have stressed the need for visualization tools. We feel that the problem is more extensive than this. The most compelling reason to combine multiple programs into one large executable is to share data structures, such as those data describing the atoms in a molecule. Shared memory provides a means to accomplish this while retaining the enhanced user flexibility and natural parallelism offered by the cooperating processes model of computing.

The source code for the programs described above is available at no cost from the authors. The MULTI system currently runs on SGI 4D systems. The basic shared memory communication model should, however, work on any system V version 3.2 or later. Most of the remaining portability issues relate to the user interface.

ACKNOWLEDGEMENTS

We thank our early users, in particular Ken Lipkowitz, Paul Charifson, Lee Pedersen, Charles Foley, Lee Kuyper, Lisa Balbos and Donald Doyle, for enduring less than perfect software, as well as for many useful suggestions, some of which played a major role in the evolution of the MULTI system. In addition we would like to thank all those researchers in the community who have made their source code available to us.

REFERENCES

1 CHARMm is a product of Harvard University. Quanta, together with the CHARMm interface, is available from Polygen Inc.

2 Upson, C. Tools for creating visions. *UNIX Rev.* 1990, **8**, 39–47

3 Stumann, M., and Zhou, S. Algorithms implementing distributed shared memory. *COMPUTER* 1990, **23**, 54–64

4 Ahuja, S., Carriero, N., and Gelernter, D. Linda and friends. *COMPUTER* 1986, **8**, 26–34

5 Bach, M.J. *The Design of the UNIX Operating System.* Prentice Hall Software series. New York (1986)

6 Rochkind, M.J. *Advanced UNIX Programming.* Prentice Hall Software Series. New York (1985)

7 Stevens, W.R. *UNIX Network Programming.* Prentice Hall Software Series. New York (1990)

8 Carson, M., and Bugg, C. Algorithm for ribbon models of proteins. *J. Mol. Graphics* 1986, **4**, 121–2

9 Darden, T. A method for fitting a smooth ribbon to curved DNA *J. Comp. Chem.* 1989, **10**, 529–51

10 Connolly, M. Solvent-accessible surfaces of proteins and nucleic acids *Science* 1983, **22**, 709–13

11 McLachlan, A.D. Gene duplications in the structural evolution of chymotrypsin *J. Mol. Biol.* 1979, **128**, 49–79