Multiprocessor Architectures are Converging

Erik P. DeBenedictis

AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

Hypercubes are one of seveal architectures trying to eliminate the Von Neumann Bottleneck without drastically changing the appearance of computers. Examining these projects reveals that certain common approaches have been successful for all the projects even though they were independently motivated. This paper examines these approaches, which are named *modular* and *protocol* programming. To show the completeness of these methods, the paper shows how to build a system that could be used like a conventional computer but runs faster due to its transparent use of parallelism.

INTRODUCTION

It will be shown that parallel systems have their algorithms expressed in two different ways. The bulk of the code for a parallel program is written using methods optimized for human efficiency in coding. In the code created by these methods, asynchronous events have no bearing on control flow and the result is deterministic. I call these methods *modular parallel programming*, and this code often looks like Pascal programs. Parallel programs also have a portion that exploits asynchrony and is generally non-deterministic. I call this portion *protocol-based parallel programming*, and it is most often found in the programming primitives. Examination of many computing systems reveals algorithms of both types, but perhaps more significantly, that they are isolated from each other.

Conventional computers -- VAX's, for example -- are programmed in Pascal, C, Fortran, Unix (TM), etc. When writing programs, the programmer thinks deterministically; he assumes that the computer will execute exactly the same sequence of statements when run multiple times. The hardware is different, however. The underlying hardware and operating system typically employ a cache, paging hardware, and interrupts which are asynchronous and non-deterministic. While the Pascal statement I:=I+1 will produce the same result run after run, the hardware may access cache, main memory, or disk -- the choice being non-deterministic. When a microcode designer writes the cache management routines, he programs using non-deterministic *protocol programming*, but a major part of his job is to provide a consistant memory abstraction so that Pascal programmers can write deterministic *modular programs*. The concept is to look at an algorithm and ask whether the programmer had to worry about non-determinism, not whether the actual execution, viewed in minute detail, is deterministic.

It is said that the exception proves the rule, so consider the following exception. Operating systems often have 'hooks' that allow a user program to trap certain operating system interrupts. Asynchronous operating system events can thereby invoke a subroutine in the user program -- introducing asynchrony and non-determinism. In using this the user will typically make locks from memory locations to assure that the asynchrony is contained within a small fraction of the code, however. These apparent exceptions are consistent with a loose interpretation of the rule. The fact that a name like 'hooks' is given to the facilities shows that they are a recognized exception to usual practice. Furthermore, the use of locks is a good faith attempt to keep asynchrony out of the conventional code.

Other Examples

Figure 1 gives other examples of modular and protocol programming in existing systems.

Shared memory multiprocessors are generally programmed in Pascal-like languages with a fork-join primitive added. Forked processes may finish asynchronously and thereby make termination order non-deterministic. The behavior of the join primitive to the main program is independent of the termination order, however. Therefore, while the fork-join primitive is exposed to non-determinism,

Idea From	*Modular*	*Protocol*
Conventional Computer	C, Pascal, Unix	Cache Paging Memory
Shared Memory	C, Pascal+ fork-join	Interrupts Kinds of Shared Mem.
Distributed Computer	C, Pascal	NFS rlogin
Hypercube	"loosely synchronous"	message kernel
Parallel Language (Ordered Ports)	process	liason process

Figure 1. Examples of Structured and Unstructured Programming.

it has the effect of insulating higher-level code from non-determinism..

Modern shared-memory hardware presents the user with a simple deterministic abstraction of memory (1) -- often with a gimmick, such as fetch-and-add (2) -- while operating with a non-deterministic protocol internally.

Distributed computers -- a network of SUNs, for example -- are programmed by conventional C programs on each processing element (PE). The operating system uses elaborate protocols such as NFS (Network File System) and rlogin (remote command execution and login) to create filesystem and pipe abstractions.

Hypercubes are commonly programmed with the *loosely synchronous* paradigm by the user, but the message passing kernel is quite asynchronous. A topic of this paper is the development of a more sophisticated set of programming primitives for hypercubes.

The final example is a parallel programming language (3) apparently developed to allow most convenient coding of message-based parallel programs. This language has two kinds of processes, ordinary and liason, with different syntax for each. Ordinary processes are independent and described as "sequential programs," whereas liason processes have an unusual structure for creating customized communication primitives. It is apparent that the division of algorithms into modular and protocol parts transcends the software-hardware boundary.

Protocol Programming

A protocol for the purposes of this paper is a collection of state machines coupled with messages (4). State machine and protocol design are *ad hoc* in comparison to computer programming, although there is some use of advanced methods. Specifically, simple state machines and protocols can be automatically designed and verified (5). There are high level programming languages, however -- such as Esterel (6). Esterel is much less popular than Pascal -- a popular modular programming langugage.

The top part of figure 2 shows a protocol which has three states on each of the sender and receiver. The two state transition diagrams interact: the transmission of an R message in the left causes the state transition associated with receiving an R message on the right. Also illustrated is a single state diagram representing the combined behavior of the two protocols. When protocols with P and Q states are combined, the resulting protocol may have as many as PQ states.

Any distributed algorithm can be represented as a protocol, making it a complete representation. Describing distributed algorithms as protocols has the advantage that the representation method will not prevent the specification of the optimal algorithm. The disadvantage to protocol representations is that there is a complexity explosion as protocols are combined.

Modular Programming

There are methods that avoid the complexity explosion and thereby assist humans in constructing large, functionally correct, programs in reasonable time. The application of these methods to distributed systems is referred to here as modular programming. For conventional computers, these methods are sometimes called *modular programming*

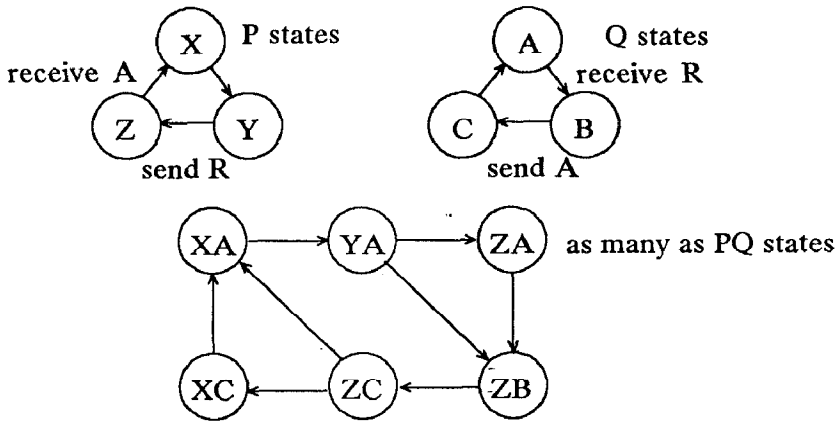

Figure 2. Protocol Programming.

(7), which is closely related to the well known *object–oriented programming* (8). Some people define the phrase *structured programming* as a style of programming devoid of goto statements, while others define it as *that which makes coding efficient in human terms*; the latter definition of structured programming is the same as my definition of modular programming in this paper.

The upper left part of figure 3 shows a programmer coding subroutine X. X is considered to be of the largest size that a programmer can reason with. The programmer can similarly code subroutine Y. Significantly, however, the programmer can also conceptualize a main program that uses calls to both these subroutines along with other code. Even though the entire program is three times as large as the programmer is comfortable reasoning with, the coding is successful because it can be divided into chunks. This compares favorably to protocol programming where the complexity of three programs of complexity x is x^3.

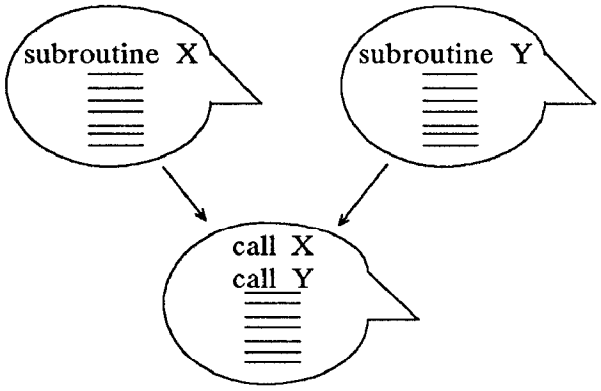

Figure 3. Modular Programming.

The language must have a particular property for this type of abstraction to work. It must be possible to write and debug subroutines in isolation and yet be assured that they will work properly when combined with others. Chunks of code should be axiomatically independent except for certain carefully designed interfaces. The most common such interfaces are subroutine arguments, and local and global variables.

The point of all this is that we are using divide–and–conquer to simplify the design of a multiprocessor system. The protocol–like, asynchronous, and non–deterministic parts will be relegated to being programming primitives. The bulk of the code constituting an application will be structured conventionally. Since humans cannot write large unstructured programs, there is no need for a machine that can execute them.

Programming With Both Methods

Investigators who have studied how people program conventional computers from a psychological standpoint suggest that mini programming scenarios, or Plans, are important. Figure 4 illustrates the Sentinel–Controlled Counter–Loop Plan (9). This Plan reads a series of values until it encounters a particular value (SENTINEL) indicating the end. The Plan tallies the number of values read. The code illustrated is claimed to be an instance of an item of abstract knowledge known by Pascal programmers; a skilled programmer might know a hundred Plans of this type and composition rules for them. The statement "count := count + 1;" is understood by the programmer to be a place where any statement –– indeed another Plan –– can be inserted. The Plan shown in figure 4 is sequential and therefore probably inefficient on a multiprocessor. However, the concept of using Plans to conceptualize programming is effective for multiprocessors, if the right toolbox of parallel Plans could be devised.

```
count := 0;
read(x);
while x <> SENTINEL do begin
   count := count+1;
   read(x);
end
```

Figure 4. The Sentinel–Controlled Counter–Loop Plan.

Now consider a multiprocessor Plan. Figure 5 illustrates the Integer Dispatch Plan, which is a multiprocessor Plan inspired by Ultracomputer programming (2). The action is the evaluation of another Plan, coded as *f(x)* for the integers *1..k*. The operation involves a distributed programming primitive that models the *take a number* mechanism often found in a bakery. The customers in a bakery represent the PEs of the multiprocessor which do work. When a customer enters the store he *takes a number i* from the take a number mechanism. When a number exceeds *k*, the Plan is complete and the customer leaves never to return. Otherwise, the customer computes *f(x)* and takes another number. Figure 5 illustrates this algorithm in terms of *take a number* –– a programming primitive –– the behavior of which is discussed next.

```
PE1: while (i = take_a_number <= n)
        f(i)
PE2: while (i = take_a_number <= n)
        f(i)
...
```

Figure 5. Integer Dispatch Plan.

Take a number is easily implemented with the Ultracomputer fetch–and–add (f&a) primitive. F&a(L, V) on memory location L causes its value to be incremented by value V, and the original value of V to be returned as the value of the f&a. The Ultracomputer has sophisticated hardware to assure fast operation of this primitive even when many PEs are operating on the same location simultaneously. A moment of thought reveals that f&a(T, 1) is

appropriate for *take a number*, where T is a temporary memory location, initially 1, belonging to the Plan.

The Integer Dispatch Plan is straightforward to understand when coded with f&a, but would be impenetrable if coded with messages. The code on each PE is a simple *while* loop with an interaction with a distributed primitive: it is not asynchronous. The protocol for the f&a primitive -- with a combining network -- includes the receipt and retransmission of messages from nodes which are computing *f*. To code this behavior with messages would require periodic polling or asynchronous interrupts. Therefore, this example illustrates a programming task easily coded by separating the modular and protocol portions, but which would be difficult to code otherwise.

PROGRAMMING PRIMITIVES

Let us consider distributed programming primitives by first considering the ideas of others. Recent shared memory multiprocessors employ snooping caches and multiple distributed primitives. The snooping cache projects a memory abstraction to the programmer when in fact, the underlying hardware works differently.

Many recent shared memory machines have both a regular memory and a second, special memory. The second memory in superminicomputers support software locking directly, and is fetch-and-add for IBM's RP3. Because the behavior of fetch-and-add memory is so complicated, its design could be considered 'programming' rather than 'logic design.'

Distributed computing systems have an elaborate message-based protocol -- NFS, for example -- that makes a disk equally accessible on all the machines on the network. This is equivalent to saying that a user can run a program on an arbitrarily selected CPU with no change in behavior.

Three precedents are established by this examination of distributed primitives. First, global naming conventions seem popular -- shared memory machines have global addresses for accessing primitives, and distributed computing systems have global file names. Second, there is precedent for presenting the user with an abstraction different from what the electronics provides directly -- memory, fetch-and-add, and disk files are implemented with messages. Finally, distributed primitives are developed with attention to programmers' needs. Shared memory systems with different kinds of primitives give the programmer a choice.

BTL Hypercube Software System

The BTL Hypercube has software that provides the programmer with an enhanced set of primitives. The BTL Hypercube is a experimental hypercube constructed at Bell Labs in 1984 with a similar design to commercial hypercubes. Experimental system software and applications were developed for this machine, which are reported in the literature (10-15). There is another project at Bell Labs to develop hardware with similar functionality -- which is described later.

The operating system for the BTL Hypercube supports the distributed primitives illustrated in figure 6 as first-class objects. In addition, a programmer can develop primitives for inclusion in a single application, or as an addition to the operating system. The programmer can therefore choose the most appropriate primitives for a given application.

Types of Distributed Primitives
Queues
Master-Slave Objects
(control distribution+collective acknowledge)
Synchronizers
Distributed Addition Objects
(fetch and add)
+ single application primitives

Figure 6. Primitives on BTL Hypercube.

BTL Hypercube software supports an tremendous number of instances of these primitives, as illustrated in figure 7. The PE has access via subroutine calls to primitives of up to 256 different types, and up to 4 billion independent instances of each type. The operating system has a table with an *input function, output function,* and *state vectors* for up to 256 different protocol behaviors. Protocols are installed at program initialization time by presenting to the operating system functions for input and output state transitions (protocol behavior) and a chunk of memory for demand allocation of state vectors (protocol instances). The software has a *virtual distributed primitive facility* where memory is allocated for each of the 4 billion protocol instances on its first use. Each protocol instance is uniquely identified in the entire multiprocessors by a global names consisting of its type and instance numbers.

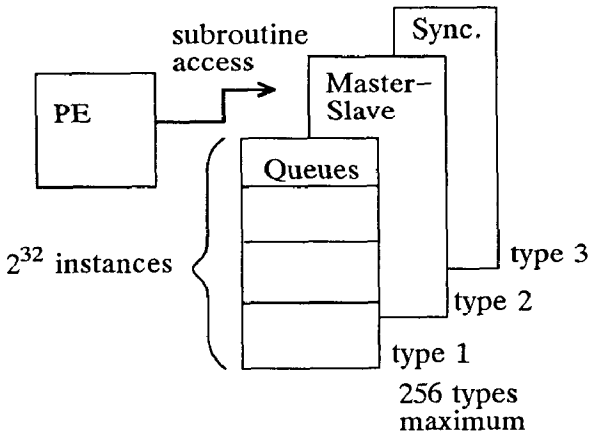

Figure 7. Arbitrary Number of Primitives.

All primitives are independent, which prevents spurious system crashes when modules are combined -- as may happen with some commercial hypercubes. When two subroutines are combined on some

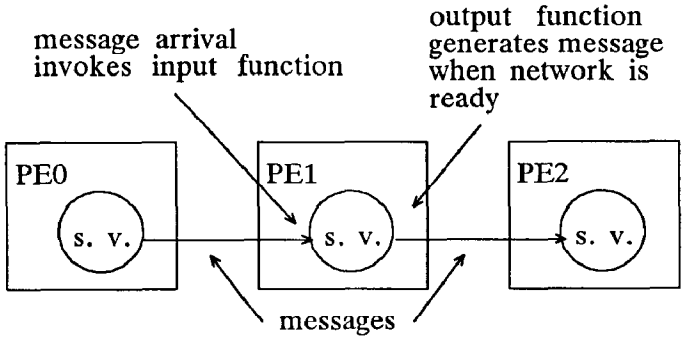

Figure 8. BTL Hypercube System Operation.

commercial hypercubes, there is a possibility that the buffer memory in some PE might overflow, causing deadlock and a system crash.

The operation of the BTL Hypercube's software system is illustrated in figure 8. Each primitive is an arbitrary finite state protocol between all the PEs in the multiprocessor, addressed by a global name, and defined by input–output functions and a state vector. When a message arrives, the operating system decodes the protocol type and instance to identify the primitive's input function and state vector. If the particular primitive has not been previously accessed on that PE, a state vector is allocated from a freelist of state vectors. The operating system then invokes the input function, which modifies the state vector. Sometimes this is the end of the story, at other times the appropriate response to the input message is for another message to be sent in reply or to another PE. If a message is to be sent, the operating system invokes the output function on the state vector -- when the network is ready to accept input.

Joe Koszarek has a project which generally incorporates some of these features into hardware, and this project is described in detail in the paper *Hardware Support for Distributed Objects in a Hypercube*, elsewhere in this volume. In summary, Joe's machine uses special purpose hardware to execute the protocols associated with primitives. State vectors are in special memory, and the input and output functions are in microcode. Finally, the interface between the structured portion of an application and the primitives are by memory accesses, rather than subroutine calls. Each primitive is assigned several memory addresses in the address map of the main CPU. To access a primitive, the CPU does a memory read or write from the appropriate address. The special purpose hardware may delay the memory access by at most a few cycles while processing the access. The hardware version is much faster than the software version.

CODING–EFFICIENT PARALLEL PROGRAMMING

The way people think determines computer programming and not vice–versa. To illustrate this point, consider the following tongue–in–cheek example:

potential hypercube user: I want to solve a problem on a hypercube which I describe in such–and–such a way.

hypercube guru: Your conceptualization of the problem is wrong. Come back when you can formulate it in terms of bees flying near a beehive under the influence of artificial forces.

The potential user seldom comes back. This is a clear example of hyperbole because hypercubes are not programmed with bees, beehives, and artificial forces. However, the methods of hypercube programming -- multiple computers, loosely synchronous programming, and messages -- may be equally ridiculous to a user accustomed to programming with mathematical operations, statements, and subroutines.

To avoid scenarios like the one above, there must be compatibility between the abilities of the user and the machine. There is some consensus that multiprocessors will fail to be compatible in some ways -- dusty decks of code will not run, and new languages will have to be developed. I will show here that the thought processes that programmers use to translate problems into programs can be similar, however. I will also show that operating systems can be exactly the same -- which is important since most people access computers only through operating system commands.

Conventional computers have different programming languages with different features. Every computer language is based on a small set of language features upon which a tremendous amount of programming experience can be developed. Sometimes a small change in the features will result in a tremendous change in programming experience -- a person can be a good LISP programmer but a poor C programmer. There is a major class of languages that are similar -- Fortran, Basic, Algol, Pascal, and PL1 (FBAPP[1]) -- to name a few. These languages have many of the same features -- and perhaps not surprisingly, programming skill in one language often can be applied to another. My suggestion is to apply features from this class of languages to parallel

[1] I got this term from Alan Perlis -- before the popularity of C -- which is now one of FBAPP's most popular members.

machines.

There are four additional language features which, in my judgement, would make multiprocessors FBAPP-like. There many other features that already apply to hypercubes -- because hypercubes use Fortran and C as base languages, a good set of primitive data types are available, for example. It can only be conjectured that there are *only* four more features needed and that these are the correct ones: to prove the conjecture requires building a hypercube based on these features, waiting a decade, and seeing what people think about it. The four features are listed in figure 9, along with feature "etc." to illustrate tentative nature of the list.

Subroutines + Hierarchical Composition
Subroutine Arguments
Variable Scoping
Separate Compilation/Dynamic Linking
etc.

Figure 9. Language Features.

The Work of Others

The common programming paradigm for hypercubes -- named *loosely synchronous* by Caltech -- provides a particular kind of program structuring. In this paradigm, the hypercube PEs all enter a computational phase at the same time. When done computing, the PEs communicate and synchronize by sending and receiving a fixed number of messages. There is some *master* code which is invoked during the synchronization phase: this code may be on one PE or may be redundantly executed with the same input data on every PE, or both. The cycle repeats until the master code decides the program is done, at which point all PEs exit together.

This paradigm has one level hierarchical composition. PE programs (the subprograms), one per PE, can be combined into a master program (the calling program). The method is limited because master programs cannot be combined into other master programs, except by running them sequentially.

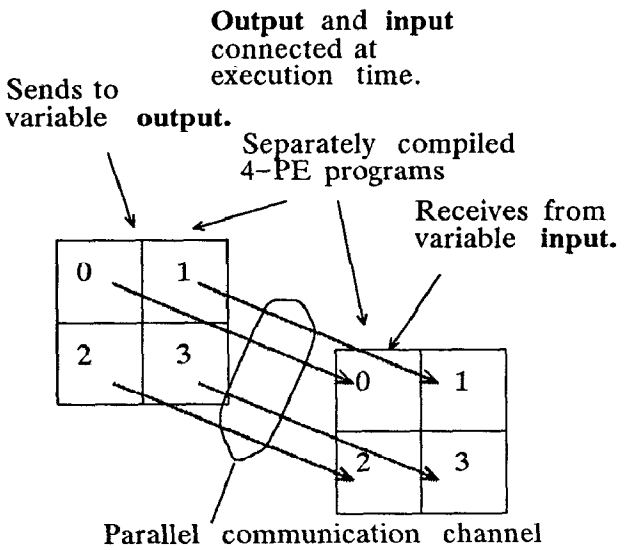

Figure 10. Parallel Communication Channel.

Distributed computer systems offer a form of control parallelism through the piping facility of the Unix rlogin command. If a network has machines named *m1*, *m2*, and *m3*, then the Unix pipeline command *m1 eqn | m2 tbl | m3 troff* will run programs *eqn*, *tbl*, and *troff* on the three machines in parallel. The rlogin facility will route the output of each program to be the input of the next program in the sequence.

This paradigm also has a one level structure, but also has scoped variables and separate compilation. C programs can be combined into pipeline programs which are executed in parallel. The method is restricted however, by the fact that the pipeline language is much weaker than C. While it is possible to combine pipeline programs into larger pipeline programs, the pipeline language cannot be considered general purpose in any sense of the word. Pipes are a communication facility represented by variables (file descriptors) local to a program and its subprograms. Finally, the method is completely dynamic: no compilation or linking is required to put programs together.

The Ncube Axis hypercube operating system has an interesting combination of these two methods, but it is also limited. A Ncube hypercube can run several programs simultaneously through *space sharing*. It is possible for a parallel main program to run two parallel subroutines. This would appear to be the desired behavior, but on the Ncube system, programs can send messages only to other PEs in their subcube, and to the host. Interactions between a parallel program and a parallel subroutine would therefore have to go through the host. Of the four language features discussed, Axis has general hierarchical composition *on the host*, subroutine arguments, and local scoping for message destinations.

In summary, the various multiprocessor projects have addressed all four of the language features listed, but never all at once. Since programmers use each of these features many times in each application, absence of a single feature will cripple a programming system. For this reason, I suggest putting all the features into a single system.

Parallel Communication Channels

My suggestions on parallel program structuring are summarized in the next few paragraphs, and are described in more detail in the paper *Distributed Programs and Subroutines for Multiprocessors*, elsewhere in this volume. Extending the Axis model, figure 10 illustrates two parallel programs which are separately compiled yet able to communicate in parallel. In the illustration, one program does output to a *distributed variable* named output (distributed variables are written in boldface). A distributed variable is a symbolic

Conventional Computer	*Multiprocessor*
memory for comm.	parallel comm. channel
memory for storage	memory
variable	distributed variable
local variable	
global variable	what are these?
subroutine argument	

Figure 11. Analogies.

representation of a parallel communication channel. At run time the operating system binds the output variable of one program to the input variable of another program. Messages directed to output are sent directly to the appropriate PE of the other program.

It seems apparent that parallel communications channels are analogous to memory locations in conventional computer languages. Figure 11 illustrates some derived analogies. Memory in a conventional computer is sometimes used for communication between parts of the hierarchical control structure. Since different parts of the hierarchy may appear on different PEs, the analogy of memory for this purpose must involve distributed programming primitives. This is not meant to imply that all memory on a conventional computer is to be replaced by distributed programming primitives: memory on a conventional computer is sometimes used for storage of intermediate results, and multiprocessors have memory for the same purpose. A variable is a symbolic abstraction of a memory location in a conventional computer, and I use the term *distributed variable* to name the symbolic abstraction of a parallel communication channel on a multiprocessor. Conventional computer languages have scoping rules for variables -- distinguishing between local and global variables and subroutine arguments. Proper variable scoping is known to be important in language abstractions. The next two sections illustrate distributed variables in local and global contexts.

Local Variables

Figure 12 shows a use of a parallel communication channel represented by a distributed variable. The left part shows a shell program (a sequence of commands for the Unix user interface) which is running a pipeline subprogram. The pipeline runs programs s1 and s2, connecting output to input. The shell program creates a variable called **pipe** and passes a reference to it to the two subprograms. The symbolic representation of **pipe** is interpreted at run time to allow messages to be sent from s1 to s2. A second shell program, illustrated on the right, runs a second pipeline program. This shell creates a different **pipe**. The result is that the two pipelines operate independently, as desired.

Global Variables

Figure 13 shows a global distributed variable acting as an access point to a disk subsystem. At boot time, the operating system creates a distributed variable for disk accesses and passes reference to this to the parallel disk subsystem. The shell on the left and the ls command (Unix directory listing) are given access to the variable **disk**. The shell on the right and its ls command likewise inherit the same variable. The two ls programs therefore access the same disk system, which is the desired behavior for a timesharing system.

In this example, the distributed variables have three functions: they represent the multiple communication channels that result from connecting parallel programs; they channel the data directly from source to destination, avoiding a bottleneck by following the inheritance path to the operating system; and they must convey enough information to the disk subsystem in order to indntify where the reply should be sent.

The previous example made it easy for distributed programs to send messages to parallel disks. However, messages may not be the most convenient interface. Standards for user interfaces to operating systems require specific semantics for input and output -- such as a common read/write pointer. It would be possible to implement these semantics in the parallel communication channel if protocol-based distributed programming primitives were used instead of messages.

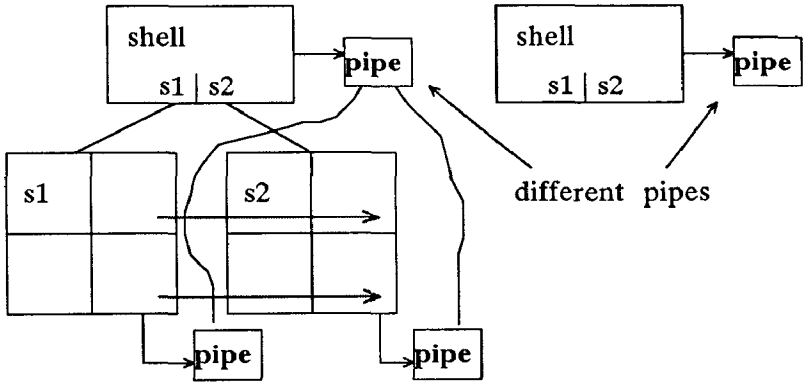

Figure 12. Parallel Local Variable.

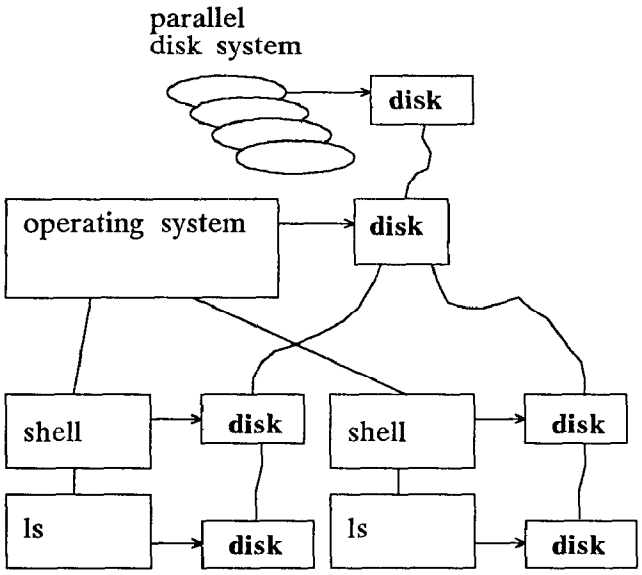

Figure 13. Parallel Global Variable.

CONCLUSIONS

With the methods outlined, we can construct a parallel system that has the same behavior as Unix pipelines. Pipelines consist of programs and communications channels, and we have parallelized both. Methods have been presented for parallel access to disks which also share disks appropriately for a timesharing environment. The execution time of such pipelines could be fundamentally faster than current uniprocessor Unix systems. Specifically, the runtime of a pipeline on a conventional Unix system is proportional to the number of i/o bytes and instruction executions. In a parallel system as outlined here, both i/o and instructions can be done concurrently. This demonstrates my view of how hypercubes will be used like conventional computers, but will run faster due to their transparent use of parallelism.

Unix pipelines, even when the connections are parallel, are less general than subroutines in conventional languages. This paper outlined how distributed variables could be used to implement variable scoping in a hierarchical distributed program in a way similar to several popular languages.

Three immediate suggestions for improvements to hypercubes were given. Enhanced primitives can be coded in software. These are compatible with the current 'loosely synchronous' programming paradigm, but may make some programs easier to code because the programmer has a richer set of communication primitives available. This is not a trivial addition because the behavior of enhanced primitives cannot generally be added to 'loosely synchronous' programs by subroutines. Second, hypercube hardware can be enhanced to assure independence of communication operations, and to improve the speed of programming primitives. Finally, program structuring methods can be used to allow composition of parallel programs. As a first step, 'loosely synchronous' programs can be combined with others and with system software -- disk subsystems for example. As a second step, multiple user programs can be combined.

References

1. L. Rudolph and Z. Segall, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pp. 340–347 (June 1985).
2. A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer, *IEEE Transactions on Computers*, pp. 175–189 (February 1983).
3. J. Basu, L. Patnaik, A. Goswami, Ordered Ports -- a Language Concept for High-level Distributed Programming, *The Computer Journal*, pp. 487–497 (1987).
4. A. Danthine, Protocol Representation with Finite-State Models, *IEEE Transactions on Communications*, pp. 632–643 (April 1980).
5. G. Micheli, R. Brayton, A. Sangiovanni-Vincentelli, Optimal State Assignment for Finite State Machines, *IEEE Transactions on Computer-Aided Design*, pp. 269–285 (July 1985).
6. G. Berry, L. Cosserat, The ESTEREL Synchronous Programming Language and its Mathematical Semantics, Seminar on Concurrency, (ed.), S. Brookes, A. Roscoe, G. Winskel, *Springer-Verlag*, pp. 389–448.
7. D. Parnas, On the criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, pp. 1053–1058 (December 1972).
8. G. Birtwhistle, *et al*, Simula Begin, *Auerbach* (1973).
9. Learning to Program = Learning to Construct Mechanisms and Explanations, *Communications of the ACM*, pp. 850–858 (September 1986).
10. B. Ackland, A. Ahuja, E. DeBenedictis, T. London, S. Lucco, and D. Romero, MOS Timing Simulation on a Message Based Multiprocessor, *Proceedings of the IEEE International Conference on Computer Design*, pp. 446–450 (October 1986).
11. E. DeBenedictis, Multiprocessor Programing with Distributed Variables, Hypercube Multiprocessors 1986, (ed.), M. Heath, *SIAM*, pp. 70–86 (1986).
12. E. DeBenedictis, Protocol-Based Multiprocessors, Hypercube Multiprocessors 1987, (ed.), M. Heath, *SIAM*, pp. 10–16 (1987).
13. E. DeBenedictis, A Multiprocessor Using Protocol-Based Programming Primitives, *International Journal of Parallel Programming*, pp. 53–84 (February 1987).
14. S. Ghosh, An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors, submitted to the *1988 International Converence on Parallel Processing*.
15. S. Lucco, A Heuristic Linda Kernel for Hypercube Multiprocessors, Hypercube Multiprocessors 1987, (ed.), M. Heath, *SIAM*, pp. 32–37 (1987).