Distributed Programs and Subroutines For Multiprocessors

Erik P. DeBenedictis

AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

There seems to be a consensus that multiprocessors are harder to program than conventional computers. To change this consensus, this paper develops distributed analogies to important and well known programming techniques found in conventional computers -- namely modular programming and variable scoping. This paper shows that with these techniques, truly parallel versions of some languages are possible -- such as Unix (TM) shell programming. High level parallel languages could also be constructed where the program planning activities would be the same as conventional languages, although lower-level methods would be different. Within the scope of the issues addressed, this can make multiprocessors *as easy* to program as conventional computers.

INTRODUCTION

As a starting point, consider the following prototypical distributed programming system. Unix pipelines are a well known method of representing the concurrent operation of several programs. A prototypical pipeline might be **ls −l | sort +3 >xx**, which lists the files in the current directory (**ls −l** is the Unix directory listing command, long format), directs the listing to the input of the **sort** program, sorts the lines (after skipping 3 fields at the beginning of each line before parsing the **sort** key), and directs the output to file **xx**. Perhaps less well known is that there are distributed Unix systems that can execute the two programs in parallel. Such a system will load the programs onto different CPUs and use physical communication links to transport the data. The combination of two programs is itself a program, however, in the sense that it reads input, writes output, and has internal behavior. In a Unix system, a pipeline can be encoded as a shell script (a file with Unix user interface commands), at which point it becomes indistinguishable from a regular program. Consistent with the previous example, shell script p could contain **ls −l | sort +3**, in which case **p >xx** has the same behavior as the previous pipeline.

The pipeline example uses two program representations -- **ls** and **sort** are written in C, and pipelines are written in the shell language -- and this is a tremendous problem. A module in a high level language (HLL) is often encapsulated as a subroutine. Subroutines can be combined to form more sophisticated subroutines to an unlimited degree. The ability to make a hierarchy of modules is an important feature which contributes to the power of HLLs. The pipeline example is not truely hierarchical: while pipeline programs can be constructed from either C programs or pipeline programs, C programs can be constructed only from other C programs. While Unix systems have 'hooks' for embedding pipeline (shell) programs in C programs, the features that make shell and C programming attractive are absent when these hooks are used. I suggest, therefore, that Unix operating systems give no *support* to the concept of combining pipeline programs into C programs. The trouble is that features of the C language not present in the shell language are unavailable for parallel programming. Since there are many such features; this is a big loss.

This paper develops a single module concept for distributed programs and distributed subroutines and formalizes the interactions between them. The result is the features of a HLL applied to distributed constructs. With these, we can combine distributed modules -- subroutines or programs -- as easily as we can combine subroutines in a HLL. Parallel input and output from programs appear also. Returning to the pipeline example, with these constructs the **ls** and **sort** programs could be distributed programs themselves with a parallel connection between them.

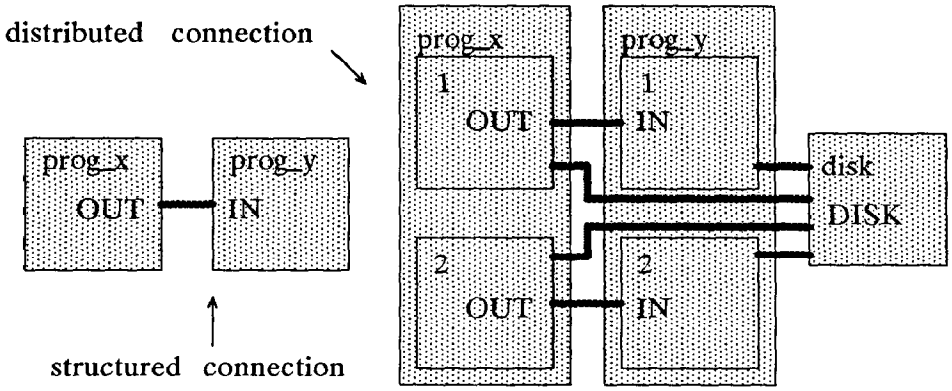

Figure 1. Connection Paradigms for Multiprocessor Programs.

An Example

The left half of figure 1 illustrates two programs connected. The output of the first program becomes the input of the other. This type of composition is well accepted. Modularity is one reason for acceptance: each program is written without worrying about the other one, thereby reducing programmer effort. A second reason is that programs can be efficiently composed in many ways to solve a wide variety of problems. The technical facilities that makes this kind of composition possible are connections in the roles of *local variables* and *subroutine arguments*. The names OUT and IN in the two programs are analogous to formal parameters in a subroutine: their actual meaning is established at run time by their callers. The combination of these two programs forms a de facto program which has a connection in the role of a local variable. This local variable connection is what OUT and IN are bound to at run time. If two instances of the connected programs were executed, there would be two connections -- which is why they are local variables. While program composition and local variables are useful, they are not currently exploited on multiprocessors.

The right half of figure 1 suggests how such a connection might be formed on a multiprocessor. Both prog_x and prog_y are illustrated as running on two processing elements (PEs). To maintain bandwidth between the programs commensurate with the number of PEs involved, the OUT–IN connection must involve multiple independent physical connections. Such a connection is illustrated by a connection between corresponding PEs in each program. The connection is created like a local variable, although it is a communication path.

Figure 1 shows the programs accessing disk files -- a well accepted activity even though it is currently unavailable on multiprocessors. File access must be through messages and must involve independent physical connections to realize multiprocessor performance potentials. Files can be placed in the same universe as local variables. Files are global, instead of local, since every program in the computer can access the same instance of a file. Also, the binding between a file name and data on the disk is made at run time, rather than at compile time -- as is the case for other variables.

Discussion

Why don't uniprocessor programming language and operating system techniques work? Conventional techniques are based on a *many to one* mapping between variables and memory addresses. We need a *one to one* mapping. When we access a variable -- say a subroutine -- on a conventional computer, the CPU goes away until the subroutine is finished. There is no reason to keep running us because there is only one CPU and it is just as well off running the subroutine as us. Although lots of programs know about the variable, only one accesses it at a time, so a stack can hold return addresses. These assumptions are not true for a multiprocessor. A subroutine can get called from every PE in the system at once, and for each call it has to send an answer back to the proper place. I claim that when a message is received, it has to *come* from a place that uniquely identifies the sender, so a response can be sent back. Furthermore, the return address must retain its validity even if the caller got moved to a different PE while we were running.

Here is the approach: Programs send and receive messages through *named ports*. Ports are either named by the programmer, such as OUT and IN, or are program identifiers (pids) of subprograms. A program hooks the ports of its subprograms together by *asserting rules.* A rule is an equation stating that particular types of messages on particular ports are to be connected. Rules are bidirectional, and a port can only be mentioned in one rule -- this assures a one–to–one mapping. Finally, there is a way of *bidirectionally multiplexing* several ports into one by encoding a port name into a message header, and vice versa. Examples of these are presented later.

There is direct analogy to variables in uniprocessor programming: Programs interact only through global variables. Variables are either programmer defined, such as subroutine names, or

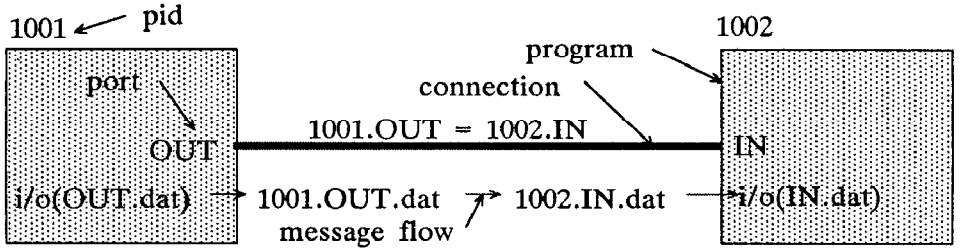

Figure 2. A Connection.

are established by the operating system (file descriptors or pids), or are constants like system call numbers. A linker hooks the variables of its subprograms together by eliminating one variable definition and several accesses to the same variable

SOME EXAMPLES

A Simple Connection

Figure 2 illustrates two connected programs. Here, the left program with port OUT sends message OUT.dat, which is the concatenation of header OUT and data called dat. Ports OUT and IN are connected. The right program gets message IN.dat. The system is axiomatically bidirectional, meaning that the right program could send IN.dat causing the left program to get OUT.dat.

The *rule* 1001.OUT = 1002.IN makes the connection. Numbers 1000 and up refer to pids which are unique integers assigned to a program when loaded for execution. When a message, such as OUT.msg, flows out of a program and into its environment, the pid is appended to its left end, i.e. 1001.OUT.msg. The *pattern* 1001.OUT will match any message beginning with 1001.OUT, such as 1001.OUT.msg. When a message matches one pattern of a rule, the matching part is replaced with the other pattern. This changes 1001.OUT.msg to 1002.IN.msg. If the pid is a subprogram, a message with a pid as its leftmost part will flow into that pid, deleting the pid from the message. For example, 1002.IN.msg flows from the environment to 1002, changing the message to IN.msg.

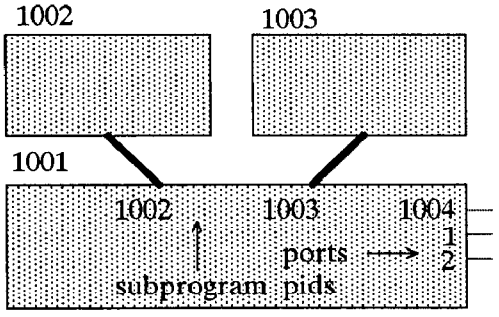

Figure 3. Subprogram Connections.

Subprograms

Figure 3 illustrates subprograms. Here, pid 1001 is a program with pids 1002 and 1003 as subprograms. Pid 1001 can send messages to the destinations 1002, 1003, 1004, 1, and 2. Since 1002 and 1003 are connected subprograms, messages to these destinations would go to the subprogram and the destination would be removed from the left part of the message. All other destinations are treated as ports with the message flowing into the environment. Port numbers should be less than 1000 to avoid the possibility of a clash with a pid at execution time because of the notation used here. On the other hand, port numbers greater than 1000 are useful for emulating subprograms.

Multiplexing

There are a variety of reasons that a system should be able to emulate itself. There is elegance in completeness, for example. More pragmatically, no matter how carefully we design hardware, it will not be perfect. Inevitably, we will run out of some resource -- physical communication links, for example. If we can emulate several logical communication links with one physical

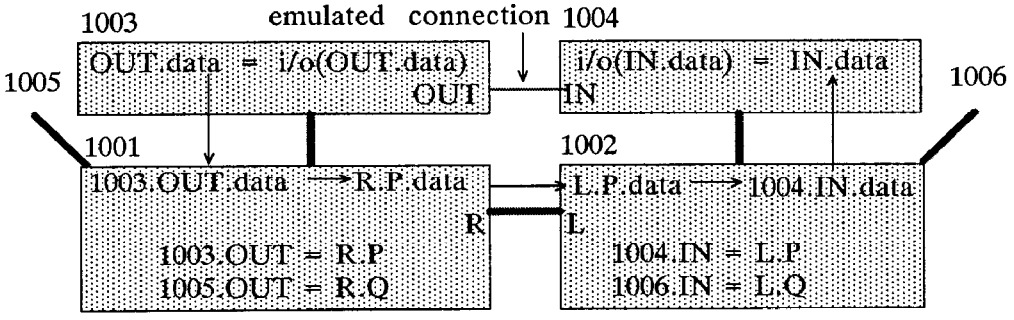

Figure 4. Link Multiplexing.

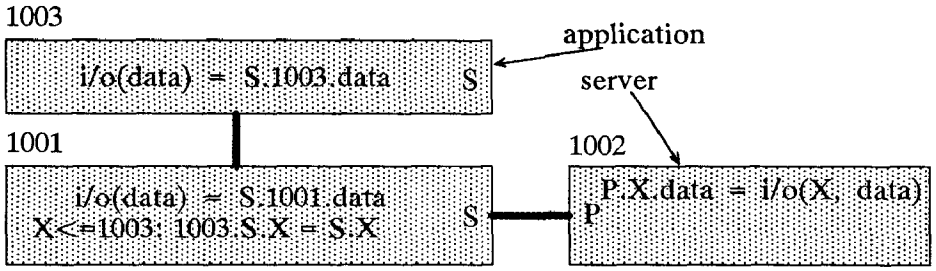

Figure 5. A Server.

communication link, the severity of the problem is reduced. This section gives an example of how one communication link can emulate two links with no change in system operation.

In figure 4, pids 1001 and 1002 are multiplexing the virtual OUT–IN connection of children programs 1003 and 1004 (and separately 1005 and 1006) over their R–L connection. Messages traversing the R–L connection have a type specification in their header. P identifies the type associated with multiplexing the OUT–IN connection. Messages with a different type, say Q or R, could multiplex other connections. The important thing going on here is that the messages traversing the *real* physical link are one word longer than those being emulated. This extra word provides the necessary extra information.

A Server

Figure 5 illustrates a server accessible via normal scoping rules. The server has one input that connects simultaneously with all programs that have the server in their scope. Messages from other programs arrive at the server with the pid of the originating program appended to the data. This allows the server to respond properly. Each program with access to the server can create requests originating from itself, or it can export server access to its subprograms.

Messages to port S, the server port, have the pid of the originator added as the second field during initial formatting. A program can export access to the server to its subprograms by relaying messages to the server without altering the pid field. The critical rule is *X<=1003: 1003.S.X = S.X*. The rule states that any message coming directly from child 1003 directed at the server -- and originally coming from a pid that matches wildcard *X* -- is to be relayed to the local server port. Wildcard *X* is qualified by *X<= 1003* which is defined to match *1003* or any *subprogram* of *1003*.

The simple expression *X <= 1003* hides considerable complexity. The capabilities introduced here are a parallel version of Algol–type scoping rules. A stack and a display are required to implement Algol–type scoping; and supporting parallelism requires that the linear stack be changed to a tree structured *hyperstack*. Evaluating *X <= 1003* requires that the run time system maintain information about which programs are subprograms of other programs. This information is the hyperstack. Executing a parallel system with variable scoping requires elaborate data structures -- and here they are, but it is fortunate that the elaborate data structures do not make this paper complicated.

Virtual Subprogram Facility

Figure 6 shows how a program can simulate subprograms of its subprograms. The straightforward method of executing a parallel program on a multiprocessor is to put one program on each processing element. Doing this assures maximum parallelism. Subprograms could not be done in the obvious way because a program and its subprograms would have to be on different PEs. The suggested solution is for each PE to have an operating system program and one user program. The operating

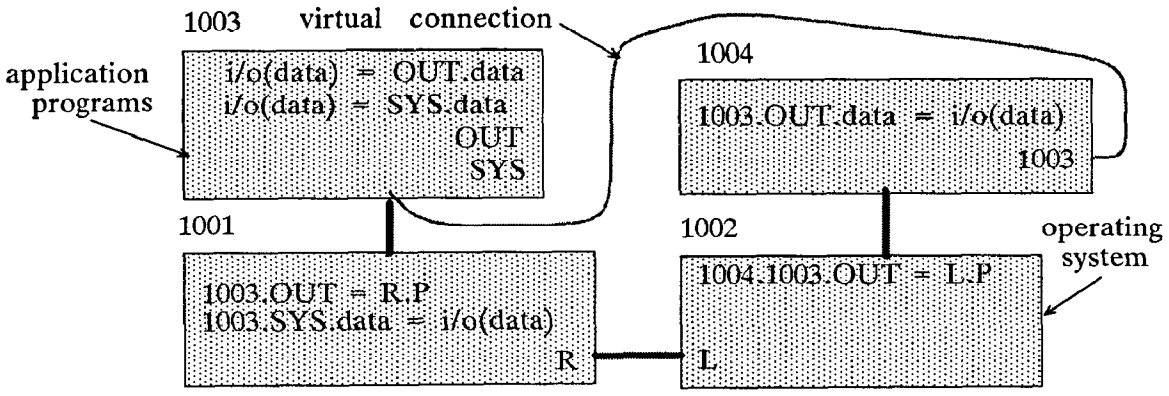

Figure 6. Virtual Subprogram Facility.

system would emulate connections between programs via the multiplexing method described earlier. Interactions between a user program and its subprograms would have to be emulated also. Fortunately, a mechanism for emulating subprograms is available. Interactions with a subprogram are via messages directed at the pid of the subprogram; but there is no difference to the user program between interacting with a subprogram with a particular pid and communicating over a like numbered port.

Figure 6 illustrates the virtual subprogram facility. Programs 1001 and 1002 on the bottom are the operating system and 1003 and 1004 are the application programs. Program 1004 requests of the operating system that a subprogram be allocated for it. The value 1003 is returned as the pid of the subprogram -- even though a subprogram is not allocated to the caller. What happens is that parts of the operating system communicate and locate a PE with no application program. The operating system on this PE then creates a subprogram, pid 1003 in figure 6. The operating system then sets up multiplexed connections for messages between pid 1003 and *port* 1003 on the calling program (pid 1004). Pid 1003 is the subprogram of 1004 as far as either can determine, but the connection is *virtual* as indicated by the curved line in the figure.

Most messages from a virtual subprogram should be relayed to its parent. The obvious exception is messages that cause creation of subprograms; uniprocessor system utilities such as memory allocation are another exception. A portion of the operating system on each PE is a server for certain activities originating on that PE. These services are labeled by messages of type SYS in the figure. Messages of type SYS never go to another PE.

A Distributed Server

The concept of a server being accessible anywhere is powerful, but a single server would be overburdened in a large multiprocessor. This section discusses a server accessible as before, but with a distributed implementation.

Pid 1004 in figure 7 is an application with access to a server through port S. Pids 1005 and 1006 are collectively a server. Instead of one server getting requests from many application programs, there are many servers, each getting requests from its single LOCAL port. Operating system 1002 connects the S port of 1004 to the LOCAL port of server 1005 with the rule 1004.S = 1005.LOCAL. The servers, such as 1005 and 1006 can communicate with their colleagues through the *server connection* ports -- SC.n. Port SC.n communicates with the n'th server. Pid 1001 is the single program that controls the entire distributed program. It allocates the n subprograms and specifies their connections. For example, the rule X < 1001, Y<1001: X.SC.Y = Y.SC.X connects the SC ports of all the subprograms together.

The OUT port on pid 1005 is the subject of the next section. Each part of the server has an OUT port that is relayed by rules 1005.OUT.X = OUT.X and X.OUT.Y = OUT.Y to the output of entire program.

A Distributed Pipe

Distributed servers can be connected by

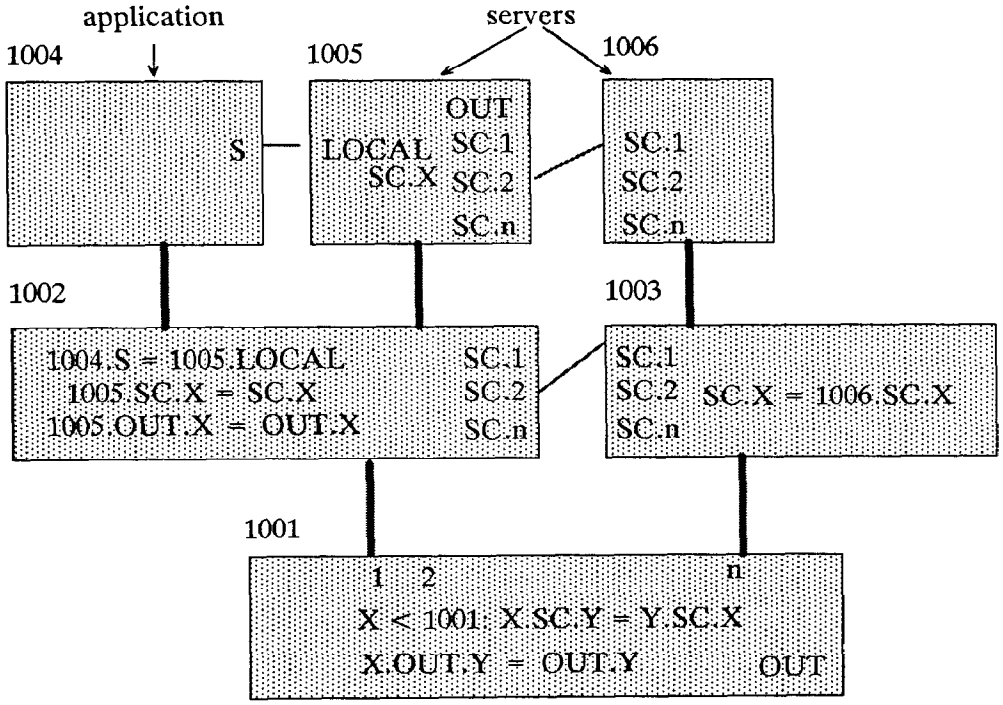

Figure 7. A Distributed Server.

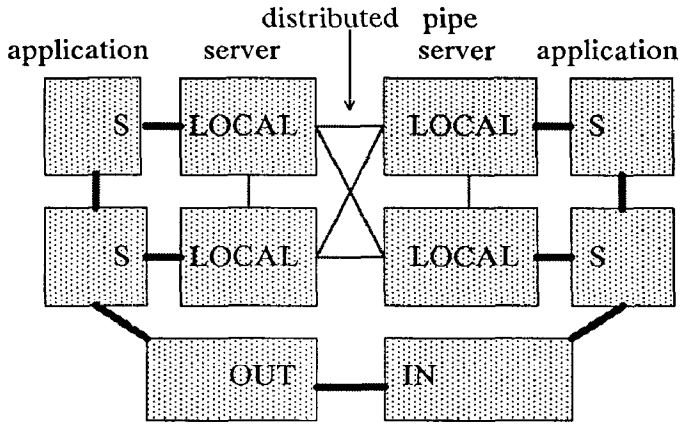

Figure 8. A Distributed Pipe.

distributed connections. The OUT ports of the distributed server in figure 7 are all connected to the OUT port of the main program. Figure 8 illustrates two main programs with distributed OUT and IN ports connected. The two programs at the bottom each have four subprograms -- two applications and one server for each application. Within each program the servers can interact, as illustrated by the vertically drawn emulated connections. Because of the OUT-IN connection between the main programs, each server can also communicate with all the servers of the *other* program. This *distributed pipe* is illustrated by the horizontal and diagonal emulated connections. While we can *describe* complicated connections effectively, making the data follow the most direct route is a subject for later.

AN ALGEBRA FOR THE RULES

Rules can be combined. This permits a message to be sent directly from source to destination even though a long sequence of rules applies.

The method of combining rules involves two steps. It must first be decided if a rule refers to a port or a subprogram. In the first step, each rule is translated from its context in a program to the global context. In performing this translation, messages to subprograms are unchanged, messages to ports have the pid of the sender added, and function specifications have the pid of relevant program added. In the second step, pairs of rules are combined using transitivity. Only minor algebraic manipulations are required: the same thing can be added to both sides of a rule (A = B implies A.C = B.C) and binding of variables may have to be manipulated.

Examine figure 5 and consider the communication between 1001 and 1002. Three rules are involved: in 1001 there is i/o(data) = S.1001.data, in 1002 there is P.X.data = i/o(X, data). Also, the link between 1001 and 1002 is equivalent to the rule 1001.S = 1002.P. Figure 9 illustrates the combining of these three rules. The final combination would be expressed in words as follows: *an i/o statement in 1001 with a simple message is tied to an i/o statement in 1002 with the value 1001 appended to the head of the simple message.*

Figure 10 illustrates a program in a syntax like the C language. Lines 1-12 and line 3 declare nested programs with a syntax similar to data structures. Line 1 says parent is the name of a program and its contents appear in brackets. Line 3 declares a subprogram of type child and variable C to represent its pid at execution time. The body of program child must be declared elsewhere. Line 10 creates a program at run time and stores the pid in a variable.

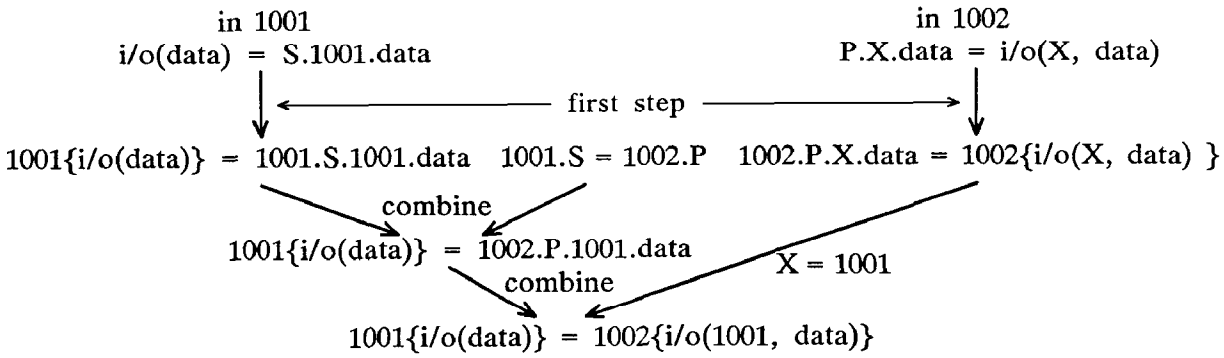

Figure 9. Combining Rules.

```
1  program parent {
2     port S;
3     program child C;       /* instance_of_child integer with pid value of child */
4     rule { i/o(data) = S.1001.data } ;
5     rule { X<=C: C.S.X = S.X } ;
6     rule_handler(pid, rule) { }    /* receives rule from child */
7     i(data) { }            /* message input handler */
8     o(3.14);           /* message output */
9     assert_rule(rule);
10    int pid = create_program();
11    terminate();
```

Figure 10. Structured Program Syntax.

The interfaces between a program and its environment are via ports, as declared in line 2. The compiler assigns small integers to port identifiers when a program is compiled, like the assignment of offsets to elements of data structures. When a subprogram is declared, as in line 3, a skeleton of the subprogram that has port assignments should be available -- like .h files in C.

Lines 4 and 5 declare rules for message connections. Line 4 binds two functions of program parent to messages. Function i() is a function, declared on line 7, which will be invoked when input messages arrive. Calls to function o(), declared on line 8, generates output messages. Statically declared rules can be processed at compile time for optimum efficiency. Line 9, however, asserts a rule dynamically.

Static and dynamic rule assertion -- and also subprogram declaration -- have a philosophy similar to memory allocation on conventional computers. Variables and arrays can be declared at compile time or created at run time. Compile time declaration is usually slightly more efficient, but is generally less flexible.

Line 6 is a function invoked by the operating system when a subprogram asserts a rule. Most of the time a rule assertion by a program should update a single data base of rules in the operating system. Sometimes more elaborate behavior is appropriate. Two examples are debuggers, and running an operating system as an application program, which both require omnipotent control over their object programs. Usually a rule handler will simply reassert a subprogram's rule in the parent's context.

Line 11 terminates the program -- but not its subprograms. This paper has been written assuming that programs, pids, and rules are allocated and later abandoned. There is no explicit deallocation. A terminated program does not generate output messages and declines to receive input messages. Rules remain effective until the pids they refer to terminate.

CONCLUSIONS

A *distributed* program encapsulation mechanism has been presented. Examples of hierarchical abstraction and Algol-like scoping rules have been shown as an encapsulation mechanism. Since the basic language features are present, it should be possible to code programs similarly to other Algol-like HLLs. In addition, examples are given of dynamic variable binding in the context of file and pipe i/o in an operating system. With these features, it should be possible to construct a parallel operating system with Unix-like features. It is significant that both the programming and operating system methods use the same mechanism; this makes the programming features available at every level.

The encapsulation mechanism is based on bidirectional bindings of names. The mechanism can be considered as the bidirectional analogy to the unidirectional name-to-memory bindings found in most computer systems. The bidirectional bindings are described by algebraic equations which can be interpreted in ways analogous to those in conventional computer systems: transitive closure can be applied to the equations by a compiler or linker before execution time, or by the operating system when it dynamically binds programs and files together. The equations can also be emulated -- which is its most dynamic interpretation.

The purpose of this effort was to develop distributed analogies to well-known programming methods. Disruption of accepted programming techniques is minimized because no new and exotic methods were introduced. Rather, a (hopefully complete) set of tried-and-tested programming features were considered.