Multiprocessor Programming with Distributed Variables

E. P. DEBENEDICTIS*

Abstract. A distributed variable is a globally accessible object defined by a message passing protocol with read and write operations. As programming primitives, distributed variables have greater abstractive properties than both conventional point-to-point message passing and shared memory.

This paper demonstrates the abstractive flexibility of distributed variables through an implementation of quicksort, while an analysis of this implementation profiles their efficient performance characteristics for multiprocessor hardware.

1. Introduction. The point-to-point message, or pipe, is the most popular programming model for message passing multiprocessors. The pipe abstraction matches some programming abstractions: consider the many ways Unix** commands can be combined by pipes. Not everything can be done as Unix pipeline commands, however, as illustrated by the indispensable C compiler on Unix systems.

Consider a common programming paradigm that cannot easily be represented by a simple pipe, but is easily represented in a distributed system by a variant (albeit major) of the pipe concept. The programming paradigm is the manipulation of sets of data, as illustrated in figure 1. In this paradigm, data objects $x_1 \ldots x_n$ are to be represented as a set, and an operation, say f, is to be applied to every element of the set: $f(x_i)$. In a conventional language $x_1 \ldots x_n$ are stored in an array and an iteration evaluates each $f(x_i)$.

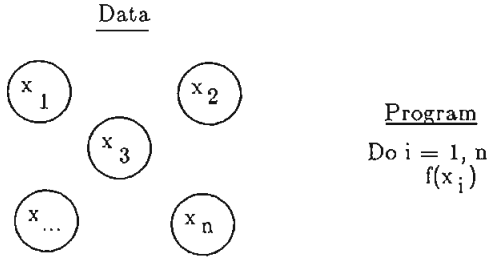

FIG. 1. Sets and Iteration.

The essence of this activity is illustrated in figure 2. Activity enters from the left with the execution of program code before the iteration. The object of the iteration is to

* AT&T Bell Laboratories, Holmdel, New Jersey 07733. ** Unix is a trademark of AT&T.

evaluate the $f(x_i)$'s, which, it is assumed, can be done concurrently. Only after the last x_i is evaluated does program execution continue to the right. The essence of the set concept is to assure that the proper number of $f(x_i)$'s is evaluated even when n varies.

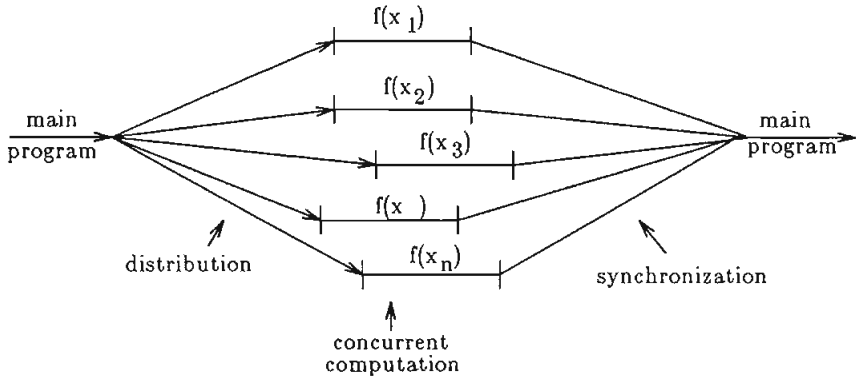

FIG. 2. Behavior of Sets and Iteration.

Figure 3 illustrates a pipelike communication primitive that models this activity, with a simple pipe shown above for clarity. When a sequence of messages, such as 1, 2, and 3 is written into a pipe, the sequence 1, 2, and 3 can later be read. The first variant on this simple pipe concept is shown below with multiple outputs, where each value written into the pipe is duplicated at every output. A reverse, or acknowledge, path is added to the multiple-output pipe as a second variant. Acknowledgements are collected from all the reading ends, combined together, and passed to the writing end.

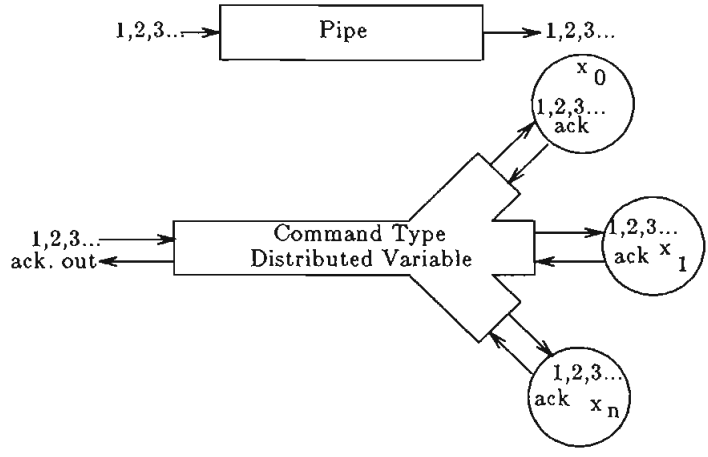

FIG. 3. Set Abstraction.

The complete application of this paradigm to a multiprocessor would have the x_i's distributed among the computational nodes along with the program code for function f.

Each computational node would await the receipt of a message from the set abstraction to start the evaluation of $f(x_i)$ and would acknowledge only when the evaluation is complete. However, this procedure requires the use of distributed variables.

2. Definition of Distributed Variables. Distributed variables exist in a naming space that is global to the otherwise independent nodes of a multiprocessor. Although distributed variables have read and write operations associated with them, they do not necessarily obey the semantics of a memory location, and they typically operate on units of data larger than one word.

Addresses in the distributed variable space are used only to establish continuity between distributed variable references on different nodes. Distributed variables are allocated by a programming language function that generates unique 'addresses' in the distributed variable space. Addresses can be communicated to other nodes just like global memory addresses in a shared memory multiprocessor.

Operations on distributed variables are based on the Communicating Sequential Processes (CSP) model of computation [7]: message output and input with guards. A programming language provides the facility to do read and write operations on distributed variables and to allow these operations to be used in guards. Other operations (e. g. setting queue size) may apply to some distributed variables, but they may not be used in guards.

The semantic operation of each type of distributed variable is described by a set of rules. These rules are based on *events*; an event, E, is an action that conveys data, happens on a particular computational node, and on a particular distributed variable in that node, and at a certain time. The two generic operations on distributed variables are designated as events, (e. g., R_a and W_b), with corresponding events being distinguished by $a = b$. The following definitions would be used for pipes:

Definition	
W_n	*write event for message n*
R_n	*read event for message n*
$T(E)$	*time of event E*
$D(E)$	*data associated with event E*

To illustrate, the rules for communication in CSP or Occam are as follows:

Definition	
$T(R_n) = T(W_n)$	*strong synchronization*
$D(R_n) = D(W_n)$	*data is passed without alteration*

These rules would be instantiated in a real system by a protocol, defined in the finite state model [3]. The protocols would be executed either by hardware protocol engines or by software with support from the operating system of the multiprocessor [4].

Formal rules are not given here for intuitive properties, such as the ability of programs to make read or write requests and have them satisfied within a reasonable time (liveness), or for events resulting from a program checking the status of a variable in a guard.

2.1. Relation of Shared Memory to Distributed Variables. Shared memory like communications techniques, including such variations as fetch-and-add, [5] provide such a flexible programming environment that many other scenarios can be easily emulated. Since each access to shared memory invokes the communications network, much more information may be exchanged than would be exchanged if more explicit message control were allowed.

Figure 4 illustrates a basic shared memory system. The single global memory is mapped into address space of every processor, allowing message communications by accesses to common memory locations.

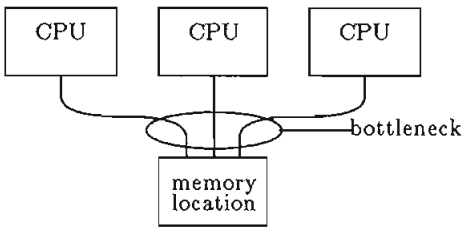

FIG. 4. A Shared Memory Multiprocessor

The naive shared memory scheme illustrated in figure 4 has obvious deficiencies corrected by the cache system illustrated in figure 5 [10]. The machine in figure 5 has special hardware to exploit the inevitable coherence of memory accesses. One type of coherence is that some memory locations are frequently read but never written. The added units in figure 5 can cache a read-only copy of a memory location and hence avoid communication operations for these locations. A second type of coherence is that some memory locations are repeatedly accessed (read and written) from one CPU while being ignored by all others. The cache units support this by designating those memory locations read-write for a particular processor and nonexistent for the others. The shared memory seen by CPUs must, however, support the read-last-value-written semantics of a memory location. Each cache entry in figure 5 includes a small protocol engine that exchanges information with the protocol engines in other computational nodes. These protocols have about a dozen states and are executed by hardware.

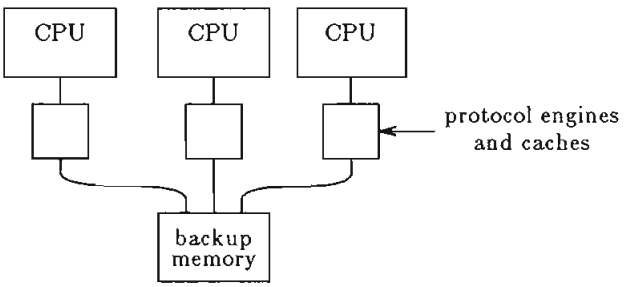

FIG. 5. Cache-Based Shared Memory Multiprocessor

Distributed variables are like an enhanced shared memory cache, as illustrated in figure 6, however the interaction semantics of distributed variables are more flexible than those of memory locations. For example, a distributed variable could be a queue, with first-in-first-out semantics. Also a single program could have distributed variables with different interaction semantics at the same time (such as memory and fifo). Moreover, the data associated with distributed variables need not be of fixed size. For example, reads and writes to a distributed variable could convey text objects. The protocols for distributed variables have 10-1000 states and are partially executed by software.

2.2. Relation of Distributed Variables to Communication in CSP and Occam. Channel-like communication techniques, including CSP [7], Unix pipes [9], and 'datagrams' [2] model linear processing of data (pipelines) in similar ways. CSP communication operations are

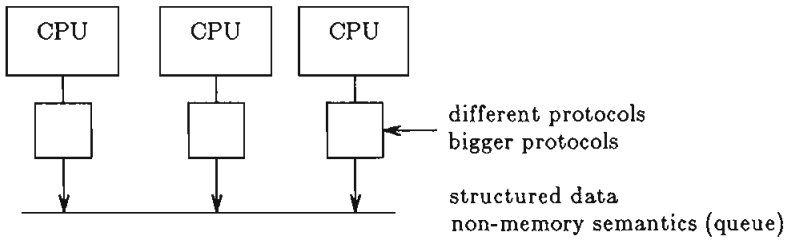

FIG. 6. Distributed Variable System

analogous to (and sometimes described as) multiple processor assignment statements. The rhetorical question is: where are the multiprocessor analogies to IF statements, WHILE statements, and procedures?

CSP, a concurrent programming language that has become a standard for describing message passing programs, has processes that can be on physically separate CPUs and that can send messages to each other. Figure 7 shows two processes, P1 and P2, exchanging a message. The statement P2!X sends a message consisting of the value of variable X to process P2. Similarly, the statement P1?Y receives a message from process P1 and puts the value into variable Y. However, in the CSP model both the sending and receiving process must specify the other, and moreover CSP is strongly reliant upon synchronization, since the input and output operations occur at the same instant, with one process blocking until the other is ready.

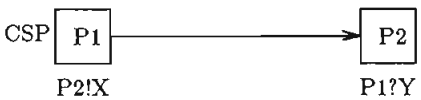

FIG. 7. CSP Communication Model

A new language called Occam [8] offers some communications enhancements over CSP, but is similar in many ways. In Occam, input and output operations are directed toward entities called CHANs instead of other processes. For this reason a process does not need to know the identity of its companion process in a message passing operation. A second enhancement is that a CHAN may be either a data structure in shared memory, implying the communicating processes are in the same physical CPU, or the name of a physical communication channel, implying the processes are in separate processors.

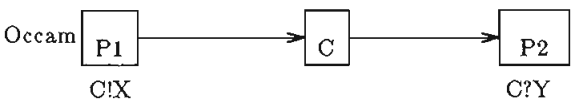

FIG. 8. Occam Communication Model

Figure 8 illustrates processes P1 and P2 communicating in Occam. Processes P1 and P2 declare a variable, C, of type CHAN, which is initialized in both processes to refer to the same CHAN by methods not shown. The statement C!X in P1 sends a message with value

equal to X to CHAN C. The statement C?Y in P2 receives the message from C and puts the value into Y. Occam is strongly synchronized, like CSP, implying that a CHAN has no queuing capacity.

Distributed variables are like Occam CHANs but possess two decided enhancements. A distributed variable, unlike a CHAN, may have more than one reader or writer at the same time. Also, distributed variables have changeable semantics; they may duplicate data (i.e., broadcast) or process it in other ways (i. e., numerical addition).

Figure 9 illustrates a distributed variable with two writing processes, P1 and P2, and three reading processes, P3, P4, and P5. The processes all have a variable declared as *int Mbox C*, a distributed variable, called C, communicating variables of type *int*. A mailbox type distributed variable, declared with the word *Mbox*, has semantics similar to an Occam CHAN, but with queuing. The statements C!X and C?Y have the same meaning as in CSP and Occam.

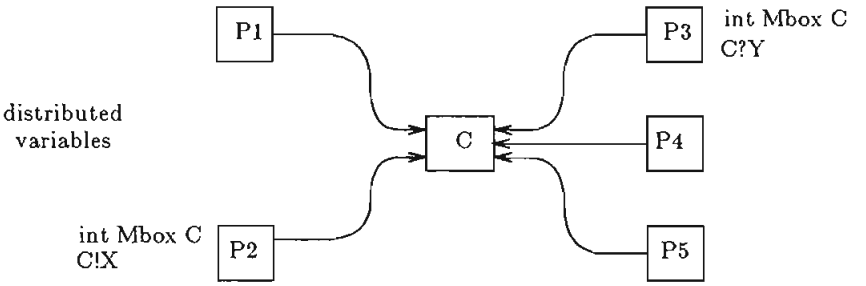

FIG. 9. Distributed Variable Communication Model

2.3. Comparison with Remote Procedure Calls. The distinction between distributed variables and object oriented programming with remote procedure calls [1] is also significant, since objects in these systems, although present on the various nodes, are so on only one node at a time, and procedure calls are communicated via messages to the node containing the appropriate object. A distributed variable, however, may be present and running simultaneously on multiple nodes, resulting in faster execution.

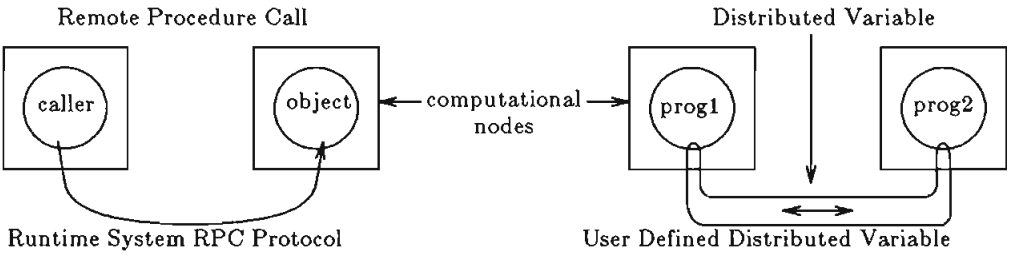

FIG. 10. Communications Scenarios

Figure 10 illustrates the distinction between remote procedure calls and distributed variables. With the remote procedure call, the user writes conventional (sequential)

programs for *caller* and *object*. *Caller* can invoke a function on *object* by requesting that the run-time system communicate the procedure name and arguments to the appropriate node. With distributed variables, however, the user supplies the behavior of the distributed variable as a message passing protocol, as well as conventional programs *prog1* and *prog2*. *Prog1* can invoke a function on the distributed variable directly and it, whose operation is under control of the user, sends the messages.

3. Types of Distributed Variables. The concept of variables in regular programming includes giving the programmer the choice of the most appropriate data type of variable from a set which includes integers, reals, and so on. Similarly, there are several semantic types of distributed variables, such as Mailbox and Command. For conciseness in presentation, only these two are described here.

3.1. Mailbox Type Distributed Variables. A Mailbox type distributed variable is a queued version of CSP messages, Occam CHAN's, or pipes. Much of the operation of a Mailbox variable is described by the following two rules:

Definition	
$T(W_i^n) < T(R_i^n)$	*causality*
$D(W_i^n) = D(R_i^n)$	*data values preserved*
$T(W_{j+N}^n) \geq T(R_j^n)$	*queue size limited to N*

The first rule states that the time, measured in the normal physical sense, of a write event is less than the time of the corresponding read event. The second rule states that the data associated with corresponding read and write events is the same. The third rule establishes queuing capacity.

3.2. Command Type Distributed Variables. Command variables model the concept of sets of data objects and operations on those sets - multiprocessor analogies to arrays and iteration - but operate like multicast. A Command variable has one writer and many readers; values written into a Command variable are broadcast to all readers, and additionally a read acknowledge is provided to the writer.

The notation used for Command variables is shown below:

Definition	
W	*write event*
W^*	*write acknowledge event*
R_i^n	*read event*
$R^*{}_i^n$	*read acknowledge event*
$T(E_i^n)$	*time of an event*
D(E)	*data associated with an event*

The rules for Command variables are described below:

Definition	
$T(W) < T(R_i^n)$	*causality*
$D(W) = D(R_i^n)$	*data values preserved*
$T(W^*) > T(R^*{}_n^n)$	*collective acknowledgement*

Command variables have *connect* and *disconnect* operations to add and subtract elements from the set. Any object can do a *write* operation, sending a message to all objects in the set, that is later read by *read* operations. Reads are optionally acknowledged by *read acknowledge* operations from set members which are coalesced into a *wait for acknowledge* operation.

The Command abstraction may be viewed as set representations instead of multicast channels, by considering the following more mnemonic phrases and names: *enter set, leave set, apply to set, get command, acknowledge command, operation done,* and *send to one.*

4. Programming Example - Quicksort. The following example illustrates the style and type of programming possible with distributed variables. For the sake of brevity, only a simple algorithm - quicksort - is presented, which sorts an unordered set of data, presumed to be in memory, and leaves the list in memory as a tree in non-descending order.

4.1. Quicksort. The conventional quicksort algorithm [6] is first presented with emphasis on those aspects important to the distributed variable algorithm. Figure 11 illustrates quicksort. At the start of the algorithm there is a bag of elements, with no particular ordering. The first phase of the algorithm is to select one element and designate it as the decision element. The second phase partitions the remaining elements into two new bags with the property that all elements smaller than the decision element go into one bag, those larger in the other. The result at this stage is three bags of elements: the original bag, with only the decision element, the *smaller* bag, with elements smaller (or equal to) the decision element, and the *larger* bag, with the rest. The algorithm is then applied recursively to the smaller and larger bags.

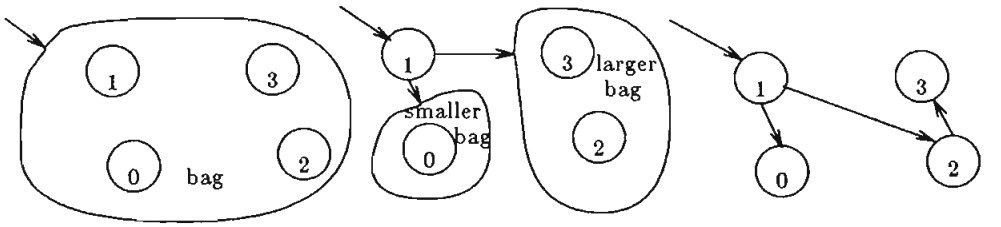

FIG. 11. Quicksort

4.2. Quicksort with Distributed Variables. In the distributed variable algorithm the objects to be sorted are processes in the multiprocessor and the original bag is a data structure containing two distributed variables (see figure 11). The elements manifest their presence in the bag by trying to read from the distributed variable structure representing the bag.

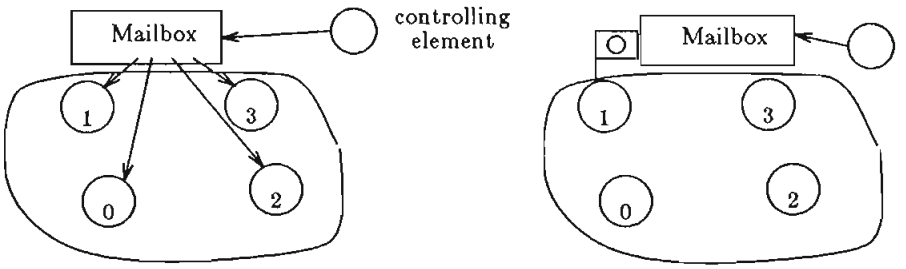

FIG. 12. Decision Element Selection

The first phase, decision element selection, is illustrated in figure 12. Decision element selection starts with the controlling element (initially the main program) outside the bag. The controlling element writes a dummy value into a Mailbox variable that is part of a

distributed variable structure representing the bag. According to the semantics of Mailbox variables each value written into a Mailbox variable is read only once, hence the read of exactly one object succeeds. The decision element selection phase ends with one object knowing that it is the decision element (illustrated by the flag on object 1 in figure 12).

The second phase, bag partitioning, is illustrated in figure 13. A bag is represented by one Mailbox and one Command variable, as shown below. Bag partitioning starts with the decision element just selected. The decision element creates two new bags by creating two initially empty instances of the distributed variable structure that represents a bag. The decision element then formats a message consisting of its key value and the two new bags and writes it into the Command variable representing the original bag. All the objects read this message.

struct bag - for sorting

attribute of structure	*data type of attribute*	miscellaneous comments
m	int Mbox	*decision element selection*
c	struct msg Cmd	*sorting message*

struct msg - formatted message

attribute of structure	*data type of attribute*	miscellaneous comments
key	integer	*comparison key*
small	struct bag	*bag for smaller objects*
large	struct bag	*bag for larger objects*

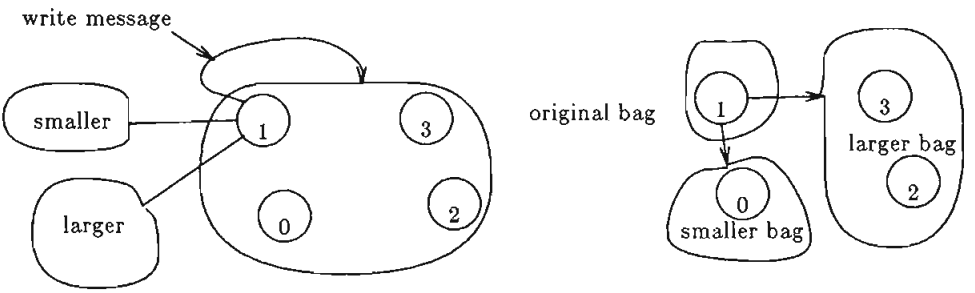

FIG. 13. Bag Partitioning

When the other objects get this message, they simply compare the key in the message with their key value and change their membership to one of the two new bags. They then acknowledge reading the message from the Command variable.

The bag partitioning phase ends at the original decision element when its Command write is acknowledged. At this point, there are two bags of exactly the same form as at the beginning of the algorithm, but with the decision elements taking the place of the old controlling element. Recursion begins when the decision element writes a dummy value into the two mailbox variables of the new bags.

4.3. Characteristics of the Quicksort Algorithm. Concurrency in the algorithm comes from two places; all key comparisons are done in parallel, and the quantity of simultaneously active decision element grows exponentially.

This algorithm uses communication channels very dynamically. This characteristic runs counter to many current trends in hardware for multiprocessors. About 20 bytes are transferred over a command variable between setup and abandonment.

The expected value of the number of sequential recursions is of $O(log\ n)$. (It is a well known characteristic of quicksort that the recursion depth has logarithmic expected value, but is linear in the worst case.) In the next section the runtime cost of distributed variable operations is analyzed, and determined to be logarithmic. The overall runtime of this quicksort algorithm becomes $\log^2 n$, which is competitive with sorting networks.

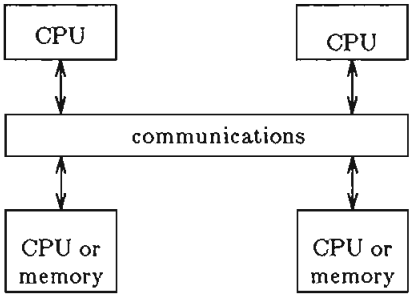

FIG. 14. Multiprocessor Model for Runtime Estimation

5. Runtime Cost of Distributed Variables. Figure 14 suggests a multiprocessor model for runtime estimation. Conventional complexity theory and knowledge of the MIP rate of a processor describe adequately the activity that occurs within a CPU or memory module. Interactions between these modules, which are usually non-computational data transfers, have different characteristics. The runtime model should account for the fact that relative values of MIP rate, communication speed, and latency vary widely between different multiprocessors.

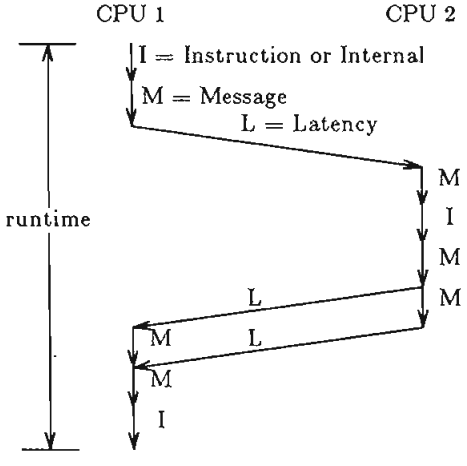

FIG. 15. Contributions to Runtime

5.1. A Metric for Measuring Runtime. Figure 15 is a data flow graph for representing contributions to runtime. In the example CPU1 computes, then sends a message to CPU2, then blocks awaiting two messages in reply, then computes again. CPU2 receives the

original message, computes, and sends two messages in reply. The arcs in each of the two columns represent operations performed by the CPU of the two computational nodes in the multiprocessor. The arcs spanning the space between the columns represent communication operations. The arcs are labeled with the letters I, M, and L, each being the amount of time taken by the corresponding operation. Runtime is the length of the longest path from any input to any output.

An arc labeled with I represents an Instruction or Internal operation, typically the execution of one instruction by the CPU. The value of I is therefore the time to execute an average instruction.

An M labeling an arc represents the Message time, or the CPU time consumed in sending or receiving information. M is in the model to limit the bandwidth of the communication network and may correspond to several physical effects. The greatest communication bandwidth occurs when a CPU is sending or receiving continuously, resulting in a message every M seconds. In a message passing multiprocessor, M would probably represent the CPU time required to queue the message and initialize the IO hardware (start-up time). In a shared memory system, M more likely represents the capacity of the communications network.

Arcs labeled with L do not consume CPU time, but instead represent latency, or the time between when a message enters the network and when it is available for reading. The sending and receiving CPUs are free to do something else during the latency delay. A message transmission consists of a M arc on the sending node, a L arc to the receiving node and another M arc on the receiving node.

Important system design decisions can be made if the I, M, and L characteristics of programming primitives, and consequently programs, are well understood. The best algorithm for a particular machine can be selected once the IML parameters of the machine are known. Similarly, a machine can be designed with IML characteristics to optimize the running of an algorithm or class of algorithms. The next section examines the runtime of some parts of the quicksort algorithm.

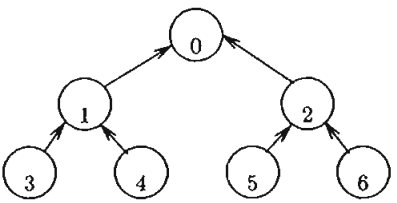

FIG. 16. Parent Function and a Binary Tree

5.2. Expanded Trees. The function $p(x) = (x\text{-}1)/2$ defines a binary tree if node $p(x)$ represents the parent of node x, as illustrated in figure 16. Computer division, where the remainder is discarded, is implied.

To define a tree rooted on an arbitrarily selected node j on an n node multiprocessor, a simple modulo shift can be used: $p'(x) = (p(x-j \mod n)+j) \mod n$.

A suggested implementation of distributed variables involves hashing the global name of the distributed variable to a number in the range $0...n-1$ and using this value to define the root of a tree. A system with many distributed variables would therefore have many trees defined on it with roots and other parts distributed with statistical evenness.

A distributed variable defined on an n node multiprocessor frequently interacts directly with only m, $m << n$, nodes. An efficient operation of the branching of the tree described above is shown in figure 17 where it stops at the first level with at least m nodes and includes all the remaining nodes.

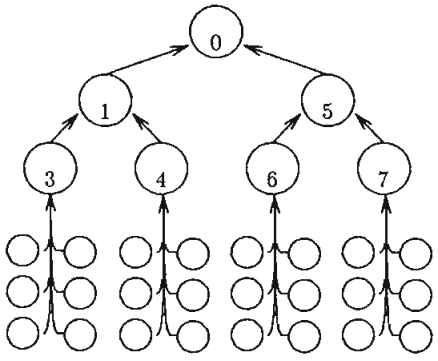

FIG. 17. A Truncated Binary Tree

Pruning nonessential nodes from the truncated binary tree becomes useful for doing operations on sets of nodes. A nonessential node is one which does not interact with the distributed variable directly, nor do any of the node's children. Essential nodes must allocate state information to indicate which of their children are essential and whether or not they themselves interact directly with the distributed variable. No resources are used by non-essential nodes.

The pruned, truncated binary tree has some fairly nice properties. It obviously has logarithmic depth because of the truncation criterion, and the fanout at all levels except the last is 0, 1, or 2. The expected value of fanout at the last level is $O(1)$ (the precise value depending on how rounding was done in computing the truncation depth), although statistical variance is significant.

What has been presented is a method of constructing trees that include a subset of the nodes of a multiprocessor based only on information readily available. The trees have logarithmic depth, and a fanout with an expected value that is constant. The hashing mentioned earlier will cause multiple trees to root themselves in different places, tending to use resources evenly. Undesirable properties of these trees are that the proper truncation depth for the tree depends on the number of interacting nodes (which may be difficult to determine), and some of the properties (fanout and even resource usage) are statistical in nature.

5.3. Tree Setup. When a node determines that it is directly or indirectly in the tree, it sends a *connect* message to its parent. Receipt of a *connect* message informs the node that it is involved in the tree, if it is not already. After a node receives an *acknowledge* message (unless it is the root) it relays one *acknowledge* message to each child having sent a *connect* message. Receipt of an *acknowledge* message by a directly connected node means that that node is properly connected to the tree and can begin tree operations.

The left part of figure 18 illustrates the logarithmic latency of a tree setup, when all nodes start at the same time. When the tree is partly setup already an additional node can be added to the tree more quickly. The right part of figure 18 shows that the true cost of a tree setup has both M and L contributions due to the inevitable sequential processing of

messages from the children.

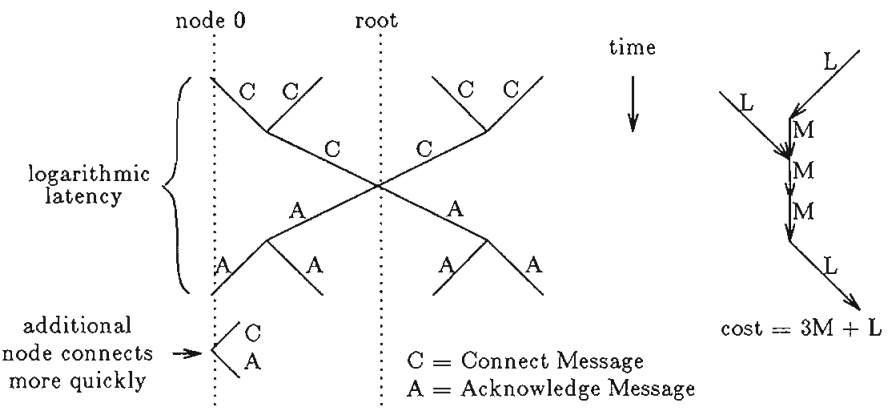

FIG. 18. Runtime Cost of Tree Setup

5.4. Mailbox Variables. Mailbox variables require a more sophisticated implementation than pipes because they may have an arbitrary number of readers and writers. The straightforward approach of putting the queue of messages on one node, in this case the root node, of a truncated tree, is adequate, however. Using a truncated tree to collect read and write requests is necessary to avoid large storage requirements on the root node.

A sufficient approach is to set up two truncated trees, one including all readers and one all writers. Each node in the tree is given a few bits to record read and write attempts generated by its children. A protocol relays one such message to a node's parent and then waits for a message indicating the parent has propogated the message to its parent.

The root node, the same for the read and write trees, matches write requests with empty buffers and read requests with messages, and sends messages directly to the requesting node.

This protocol results in a logarithmic cost for message read and write operations, although read and write requests that are satisfied are handled sequentially on one node. Decision element selection in the quicksort algorithm involves many read requests, but only one write request and only one message is transferred. Decision element selection therefore has a logarithmic cost.

5.5. Command Variables. Command variable writes, which involve each node in the tree relaying data to each of its children, have latency logarithmic in the size of the multicast set, and involve sending as many messages as there are links in the truncated tree. Acknowledge operations on Command variables have similar cost to writes, but involve each node sending an acknowledgement to its parent after receiving acknowledgements from all its children and a local acknowledgement.

6. Conclusions. It is generally recognized that hardware is ahead of software in parallel processing. In comparison to current mainframes, current hypercube hardware has potential cost-performance figures more favorable by several orders of magnitude. However, only a limited number of applications have been demonstrated to be effective for multiprocessors.

Distributed variables attempt to remedy this situation by providing the programmer with a self-consistent set of programming primitives with good abstractive potential, which are compatible with multiprocessor architectures. An analogy to Fortran and APL illustrates the possible relevance. Both Fortran and APL present the programmer with programming primitives with similar abstractive capabilities, but those primitives are used in very different ways. Fortran's primitives match the architecture of conventional computers very well (for historical reasons) and Fortran is regarded as an efficient language. APL's primitives were not developed with any machine in mind, and APL is less popular.

Distributed variables have several implications for multiprocessor hardware and software. Special purpose hardware (protocol engines) for efficient implementation of distributed variables is not particularly complex in terms of integrated circuit technology, but is not present in current designs. Current programming styles for multiprocessors tend to use communication channels as static objects. Distributed variables can also be used very dynamically - recall the dynamic use of Mailbox and Command variables in the quicksort algorithm. Distributed variable system design technique, therefore, may release the potential of multiprocessor hardware by creating an efficient programming language design appropriate for general purpose, high performance multiprocessors.

7. Appendix - Quicksort Program. The quicksort program below is written in C extended with distributed variable data types and CSP-style IO commands. The declaration of the semantic type of a distributed variable is syntactically similar to declaration of a pointer, except * is replaced by the words *Mbox* and *Cmd*. For example: X is declared as an integer by *int X*, a pointer to an integer by *int *X*, and as a Mailbox type distributed variable communicating integers by *int Mbox X*.

IO command syntax is similar to CSP's, although output guards are allowed. An *IO-command* has the syntax *chan ? lvalue*, or *chan ! expr*, optionally followed by a controlled statement, as in *chan ! expr --> statement*. *IO-commands* appear either singly, as */ IO-command /* or in alternative lists, as */ IO-command / IO-command ... /*. *Read acknowledge* operations for Command variables are inserted automatically by the compiler following *statement* in a IO-command of the form *chan ? expr --> statement*.

New is used in two ways; *New(abstract_type)* creates a distributed variable of type *abstract_type*, and *New stacksize f(args)* creates a new object with *stacksize* bytes of stack space and starts it executing *f(args)*. Connection and disconnection from a Command type distributed variable is done by the procedure calls *CCon* and *CDis*.

```
#include "../lib/mpc.h"

struct bag {                        /* sorting message */
 int Mbox m;                        /* mailbox for decision element */
 struct Smsg Cmd c;                 /* command variable for my_bag */
};

union msg {                         /* system message - 3 formats */
 struct {                           /* command to initialize a sort */
  char type;                        /* character with 's' */
  struct bag s;                     /* sorting structure */
 } sort;
 struct {                           /* command to do a sort */
  char type;                        /* character with 'S' */
 } Sort;
 struct {                           /* command to write out the tree */
  char type;                        /* character with 'f' */
 } flat;
};

struct Smsg {                       /* message for use in sorting */
 int key;                           /* comparison key */
 struct bag less;                   /* my_bag for lesser elements */
 struct bag more;                   /* my_bag for greater elements */
};

object(data,command)
int data;
union msg Cmd command;
{
 int nullint,flag;
 union msg m;
 struct bag my_bag,nullmsg;
 struct Smsg kids;
 CCon(command);
 Detach();
 for (;;) [ command ? m -->
  switch (m.sort.type) {
  case 's':                         /* command to init for sort */
   my_bag=m.sort.s;
   CCon(my_bag.c);
   break;
  case 'S':                         /* command to sort */
   for (flag=1;flag != 0;)
    [ my_bag.m ? nullint -->

     /* next 5 lines format a message */
     kids.key=data;
     kids.less.m=New(int Mbox,1);
     kids.less.c=New(struct Smsg Cmd,0);
     kids.more.m=New(int Mbox,1);
     kids.more.c=New(struct Smsg Cmd,0);

     /* send message to whole bag, including self */
     [ my_bag.c ! kids --> [ my_bag.c ? kids ] ]

     /* cause selection of decision elements */
     [ kids.less.m ! 0]
```

```
     [ kids.more.m ! 0]

     flag=0;                        /* exit from loop */

    | my_bag.c ? kids -->

     CDis(my_bag.c);                /* disconnect from old bag */
     my_bag=                        /* change bags */
      kids.key>data ? kids.less : kids.more;
     CCon(my_bag.c);                /* connect to new bag */

    ]
   break;
  case 'f':                         /* command to flatten tree */
   [ my_bag.c ? nullmsg -->
    [ kids.less.c ! nullmsg ]       /* 'left' subtree */
    printf("%12d/n",data);          /* current element */
    [ kids.more.c ! nullmsg ]       /* 'right' subtree */
   ]
   break;
  }
 ]
}

main() {
 int i;
 union msg Cmd command=New(union msg Cmd,0),m;
 struct Smsg nullmsg;
 Detach();
 srand(time(0l));
 for (i=0;i<16;i++)
  New 4096 object(rand(),command);

 m.sort.type='s';                   /* start sort message */
 [ m.sort.s.m=New(int Mbox,1) ! 0 ] /* make Mbox + send nullmsg */
 m.sort.s.c=New(struct Smsg Cmd,1); /* create Cmd part */
 [ command ! m ]

 m.Sort.type='S';                   /* do sort message */
 [ command ! m ]

 m.flat.type='f';                   /* flatten tree message */
 [ command ! m --> [ m.sort.s.c ! nullmsg ] ]
 Terminate();
}
```

8. References. -

[1] A. BIRRELL, B. NELSON, Implementing remote procedure calls. *ACM Transactions on Computer Systems,* February 1984. Pages 39-59.

[2] D. R. CHERITON and W. ZWAENEPOEL, The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* , October, 1983, pages 129-140.

[3] A. DANTHINE, Protocol Representation with Finite-State Models. *IEEE Transactions on Communications,* April 1980. Pages 632-643.

[4] E. DEBENEDICTIS, Operating System Principles for Hypercubes. 1985. Publication imminent.

[5] A. GOTTLIEB, R. GRISHMAN, C. KRUSKAL, K. MCAULIFFE, L. RUDOLPH, M. SNIR, The NYU ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, February 1983. Pages 175-189.

[6] C. A. R. HOARE, Quicksort. *Computer Journal*, April 1962. Pages 10-15.

[7] C. A. R. HOARE, Communicating sequential processes. *Communications of the ACM*, August 1978. Pages 666-677.

[8] INMOS CORPORATION, *Occam*. Inmos Corporation P. O. Box 1600, Colorado Springs, Colorado, 80935, 1984. Promotional literature distributed by Inmos.

[9] D. RITCHIE, K. THOMPSON, The UNIX timesharing system. *Bell System Technical Journal*, August 1978. Pages 1905-1929.

[10] L. RUDOLPH, Z. SEGALL, Dynamic decentralized cache schemes for MIMD parallel processors. *Proceedings of the 11th Annual International Symposium on Computer Architecture*, June 1984. Pages 340-347.

SOCIETY for INDUSTRIAL and APPLIED MATHEMATICS

1400 ARCHITECTS BUILDING • 117 SOUTH 17th STREET • PHILADELPHIA, PA 19103-5052 • (215) 564-2929

October 29, 1986

Dear Author:

SIAM is delighted to announce the publication of HYPERCUBE MULTIPROCESSORS 1986, edited by Michael T. Heath. Enclosed is a copy for your personal use. We are happy to offer you a 40% discount on any additional copies. The volume has been priced at $37.50 ($30.00 for SIAM members).

We appreciate your participation in this volume and congratulate you on a fine publication.

Best wishes,

Catherine Rininger

Catherine Rininger
Production Editor

Enclosure