International Journal of Parallel Programming, Vol. 16, No. 1, 1987

A Multiprocessor Using Protocol-Based Programming Primitives

Erik P. DeBenedictis[1]

A general strategy is presented for multiprocessing that combines programming technique, machine architecture, and performance estimation. The programmer decomposes an application into manipulations of protocol-based programming primitives (*protocols*) using *Plans* and *scenarios* from software engineering. The programmer may select from generic protocols, which include shared-memory locations and messages, or may build his own. A system architecture that supports efficient emulation of protocols is presented along with a method of estimating program performance based on network characteristics. Results are given from a protocol-based operating system on the 64 processor BTL Hypercube multiprocessor.

KEY WORDS: Multiprocessing; computer architecture; programming technique; circuit simulation; sorting.

1. INTRODUCTION

Multiprocessor research is more in need of a comprehensive strategy than specific instances of systems that run a few applications. That there are successful systems is unquestionable. There is a well established "super-minicomputer" industry that sells shared-memory computers with distributed operating systems. Similarly, there is a "hypercube" industry selling message-passing computers that solve individually large problems, such as PDEs. These examples notwithstanding, the literature has references to the "sorry state" of parallel programming.(1) Users of parallel processing from the "shared-memory" and "message-passing" "camps" will often justify their views in terms akin to "religion." This paper tries to reduce this chaos by suggesting a multiprocessor strategy. The goal is to

[1] AT & T Bell Laboratories, Holmdel, New Jersey 07733.

0885-7458/87/0200-0053$05.00/0

consider a range of issues in enough depth to show that they are compatible while not necessarily exploring every nuance.

One approach to developing a strategy is to adapt existing ideas to multiprocessors. For example, software engineering tells us how to minimize the time for a programmer to write a program. I show later that the uniprocessor bias in software engineering can be distilled from the programmer productivity aspects and the result can be applied to multiprocessors. Deriving a multiprocessor strategy only from first principles seems too difficult. However, considering the compatibility between a candidate multiprocessor strategy and first principles can give a quality measure of the strategy and perhaps give insight into improvements.

An alternate approach is to identify the features used in successful multiprocessor demonstrations and then create a system with all the features. We could then assert that all existing multiprocessors are special cases of the new system. This approach is used in places. With the appropriate protocol, the programming primitives presented here work like memory locations; with a different protocol, they are messages. Both approaches together should produce a good system without excessive featurism.

1.1. The Problem Domain

The problem domain addressed in this paper is parallelism for high performance. Distributed computing and high reliability systems are not considered directly. Furthermore, I address moderately general purpose problems. Canonical examples of general purpose problems are a compiler[2] and a text editor. The target of this strategy is commonly described by the cliche *massive parallelism*, which suggests up to 10K-100K processing elements (PEs). Most of the important primitives degrade in performance as $\log^{-1} n$ or $\log^{-2} n$ for n PEs, which is about all the justification that can be given for suggesting such a machine will work—until somebody builds such a machine and sees.

1.2. Technical Strategy

I expand on a common but underappreciated dichotomy between synchronous-deterministic and asynchronous-nondeterministic computing methods. For example, Pascal program text is clearly synchronous and deterministic. The hardware of a machine that executes a Pascal program often employs caches and virtual memory, however. There is nondeterminism in which medium (cache, main memory, or disk) has a particular Pascal variable during an access. Furthermore, the machine architect

worked hard to assure the consistency of the memory abstraction in the face of any combination of asynchronous interrupts and faults. My multiprocessing approach encourages the programmer to use synchronous-deterministic methods to decompose his program, and to use asynchronous-nondeterministic methods to select the programming primitives.

1.3. Programming Technique

Let us start by considering the way that people think about computer programming from a psychological rather than the usual mathematical viewpoint.[3,4]

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

The code shown here, obtained from Ref. 3, represents programming knowledge, or a *Plan*, called the Sentinel-Controlled Counter-Loop Plan. The Plan reads a series of values until it encounters a particular sentinel value indicating the end. The Plan tallies the number of values encountered before the sentinel.

The Sentinel-Controlled Counter-Loop Plan is an example of something that an experienced programmer has used many times, but usually through variants and in combination with other activities. Here, the counted values are obtained by reading input, whereas in a variant they might come from an array or a linked-list. A similar Plan that adds a series of values can be imagined by adding the input to a running total, instead of incrementing the count variable, each time through the loop. If the counting and totaling Plans are merged, by retaining the same reading and looping structure, and followed by a division of the total by the count, a composite Plan is obtained that computes the average of a series of values.

Figure 1 is a Plan that is relevant to multiprocessor programming. The Plan is called the Master-and-Slaves Plan (MS) and the action in it starts with the master, who picks a task and makes the slaves work on the task. When the slaves are all done, the master is notified and can do whatever action follows this Plan. Note that the activities performed by the slaves are generally asynchronous and different, thereby distinguishing this Plan from the way a SIMD computer operates.

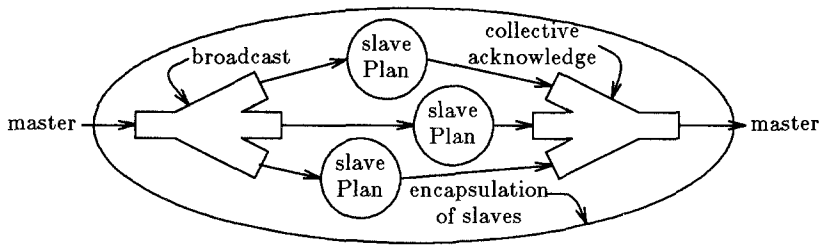

Fig. 1. Master-and-Slaves Plan.

An example of this Plan is a person running a multiprocessor program interactively. The person is the master and uses the program by repeatedly typing a command to the program and observing the output. The slaves are the PEs of the multiprocessor, and they repeatedly input commands from the master, compute something with the other PEs, and collectively report completion to the master. Of course, a multiprocessor is not restricted to having one master; some parts of the quicksort program described later are master for other parts of the program.

MS could be implemented by the master broadcasting commands to the slaves, and the slaves participating in some sort of collective acknowledgment protocol with other slaves and the master to indicate completion. A MS primitive is discussed later.

1.4. The Take-a-Number Plan

Here is a simple Plan that applies to a multiprocessor. The Fortran code shown here represents an activity of frequent occurrence in real problems; namely doing something (evaluating function $f(i)$) for all i between $1 \cdots k$. Consider the case where the evaluations of $f(i)$ are independent and each done on a single PE of a multiprocessor. Let us further allow the time to compute an $f(i)$ to vary unpredictably and strive for an implementation that achieves a good load balance. The result will be called the Take-A-Number Plan.

```
      do 10 i = 1, k
10    f(i)
```

To develop a good algorithm, consider something often found in a bakery. Bakeries often have a *take-a-number* mechanism which is a dispenser with a roll of paper tags numbered sequentially. Customers tear off a number when they enter the store and wait until their number is called. For the algorithm, the computing is done by people, and people represent the

PEs. When a person has nothing to do he gets the next number (call it i) from the dispenser, if $i > k$ the person quits, otherwise he evaluates $f(i)$. The multiprocessor code is shown here in terms of *magic_get_next_number*, which represents the operation of the take-a-number mechanism.

PE1: while ($i = magic\_get\_next\_number \leqslant k$)
  $f(i)$
  ...
PEn: while ($i = magic\_get\_next\_number \leqslant k$)
  $f(i)$

The fetch-and-add ($f \& a$) primitive proposed as part of the Ultracomputer project at NYU[5] provides the functionality of *magic_get_next_number* easily. The $f \& a$ primitive is applied to locations in memory shared between PEs and containing integers. In an indivisible operation, $f \& a$ adds a value to the integer and returns the original value. Let us say $f \& a(x, v)$ adds value v to memory location x and returns the original value of x.

A moment of thought reveals that $f \& a(x, 1)$ is suitable for *magic_get_next_number*, where x is a variable, initially 1, belonging to the Plan.

Plans like Take-a-Number are fundamental to human visualization of problems and should be exploited. Indeed, the take-a-number analogy appears independently in the synchronization literature.[6] I use Plans as the basis for a multiprocessor strategy, and not merely a way of describing algorithms in english. For example, the system architecture is contorted to make Plans execute rapidly. Likewise, the primitives underlying the execution of Plans are independent to assure that Plans can be composed reliably.

1.5. Programming Primitives and Protocols

I propose to represent distributed programming Plans, or techniques, as manipulations of programming primitives. This is already done to some extent. Multiprocessor algorithms are typically represented as sequences of message passing operations or accesses to shared memory. Currently, however, two implementations of one Plan on machines with different distributed programming primitives are considered to be independent pieces of knowledge. I suggest that if a Plan is most understandably represented as, say, $f \& a$'s, that representation should prevail even if the target multiprocessor does not have $f \& a$ hardware.

When viewed as a sequence of data transfers over wires, programming primitives are nothing more than protocols. Again, this is done now to

some extent. Protocol diagrams can be seen explicitly in the descriptions of shared memory[7] and RPC.[8] The combining elements in the f & a-based Ultracomputer[9] use a protocol to remember which f & a locations have outstanding requests.

Not only can protocols be a tool to aid understanding of parallel programming, but the architecture of a multiprocessor that supports protocols directly is discussed later. Such a multiprocessor would have more uniform performance characteristics when executing programs written with a variety of distributed programming primitives, although a modest increase in complexity would be required.

1.6. Elapsed Time Estimation

Elapsed time estimates can be attached to programming Plans. Consider the Take-a-Number Plan as applied to a uniprocessor by the Fortran code shown earlier. Clearly there are two components to the elapsed time: the loop overhead, and the cumulative elapsed time used by the embedded Plan (evaluating function f). The elapsed time of an entire program can be estimated from the composition of all its Plans.

A different elapsed time estimate can similarly be associated with the multiprocessor Take-a-Number Plan. A possible way to formulate the elapsed time is $k/n(O+P)+$corrections, where O is the overhead associated with the f & a primitive, and P is the time for the embedded Plan. This assumes constant overhead O for the f & a primitive and linear speedup otherwise. A correction term would have to be added to account for uneven finishing times of the PEs unless $k \gg n$. By composing Plans, the elapsed time of a program would become an expression in the elapsed times of *primitives*.

Being able to associate an elapsed time estimate with distributed primitives is a feature of the protocol-based system architecture that is not universally present in other multiprocessors. Protocol-based primitives are qualitatively independent by design; this means that a primitive will operate according to its rules regardless of the activity of any other primitives. This is different from a hypercube, where deadlocks can result from sending messages between improperly selected combinations of PEs. Given qualitative independence of primitives, we should be able to model the elapsed time of primitives. Given a good architecture, the model will be simple and accurate.

The elapsed time for an operation on a protocol is composed of message transmission, message latency, and queuing delays. The bandwidth and latency of a communication network can be analyzed and values determined for the elapsed time to send or receive a one message (I call this

value M for message) and for a message to pass through the network (L for latency). I model the behavior of a protocol to an interaction by a PE by a data flow graph where the arcs are labeled with M's and L's. The elapsed time is then total weight between the stimulus and a measurement point.

Figure 2 illustrates the two tests that define the M and L parameters. Time flows from top to bottom, and the "time line" for each PE is a vertical column. Vertical lines are tagged with weight M and represent receiving or sending a messages; diagonal lines represent network latency and have weight L. Elapsed time is the sum of weights on arcs. On the left, PE 1 is sending a series of messages to PE 2, without waiting for PE 2 to get one message before transmitting another. The time to send each message is designated M. On the right, two PEs are sending a message back and forth; each complete cycle is designated as $4M + 2L$.

The choice of the proper protocol for a given application may be neither obvious nor trivial, but may be important. There are often several approaches to implementing a protocol for a given primitive; *f* & *a* is such an example, as will be discussed later. Different approaches may result in dataflow graphs for elapsed time that differ by factors of $O(n)$ (where n is the number of PEs in the multiprocessor) when used in different applications. For n between 10K-100K, it would be too big a factor to ignore. I provide a path for considering this: the elapsed time values for primitives appear in the programming Plans, and the programmers are known to reason effectively at the programming Plan level.

1.7. A Review of Experimental Results

This paper is a report on the testing of these ideas on a hypercube-style multiprocessor at Bell Labs (BTL hypercube). This multiprocessor,

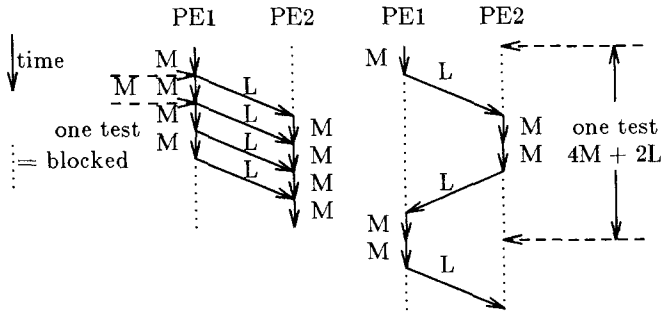

Fig. 2. Network parameters.

similar in its hardware and network architecture to other hypercube multiprocessors, has an experimental operating system designed for protocol support. In spite of the added functionality, the system appears to be a viable hypercube operating system. But the system can also be viewed as an emulator and software development tool for a yet undesigned system which would have sophisticated hardware protocol support.

The distributed programming primitives tested as of this writing are queues,[10] zero-length queues,[11] shared memory with both a combining network[9] and a cache,[7] distributed sets,[12] Linda tuples,[5] distributed addition, and broadcast.

A variety of programming Plans have been explored with this environment: dynamic programming algorithms using message broadcast, quicksorting with distributed sets, circuit simulation with queues and message broadcast, and circuit simulator variants using distributed addition and Linda tuples.

2. SYSTEM DESIGN

2.1. Network Operation

A prototypical network is illustrated in Fig. 3.[13] The PEs are represented by horizontal slices and have output and input buffers (*A-H*, *B-I*, *C-J*). Communication paths are represented by diagonal boxes and can store at least one data unit. Data units move only between buffers connected by a line, and only in a left-to-right direction (including diagonally). Data movements are indivisible and reliable, specifically, data cannot move into a full buffer. To be a network, there must be a path using only left-to-right motion from each input to every output. This type of network is not the same as a trivially connected set of nodes. Note in Fig. 3 that there are two logical buffers (*D* and *G*) between nodes 0 and 1. The BTL Hypercube uses hypercube connections to make a Banyan network, which has the required properties.

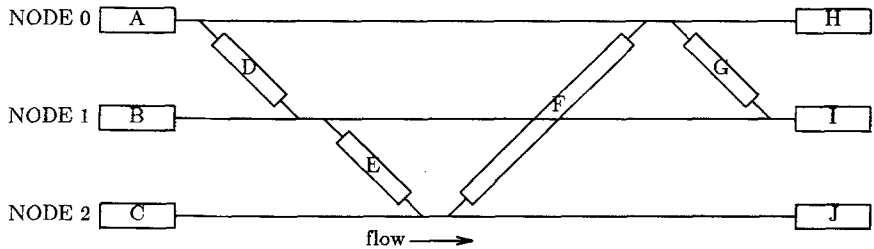

Fig. 3. Buffer graph for a simple network.

This type of network has some important properties. The network is completely reliable, simplifying protocol design and hopefully reducing life-cycle costs. The network is inherently self-throttling, meaning a PE is not constrained to process input at any particular rate—because the network will make outputting PEs slow down if necessary. This type of network was proposed by Sullivan and Brashkow[14] before its properties were appreciated [see Ref. 13] although the properties have been known for some time[15] for a different application.

2.2. Protocol Emulation

The protocol-based multiprocessor executes protocols from a representation similar to the finite state representation.[16] The system, therefore, includes a scheduler that executes the *input function* when a message arrives, and executes the *output function* when the network can accommodate a message and *action* is specified by a *state vector.* [The term *state vector* is used in the sense of a state machine, and can be thought of as a vector of bits (a binary number) or a data structure.] The input function operates on an input message and a state vector, altering the state vector. The output function operates on a state vector, altering the vector, and perhaps sending a message. Whenever a state vector is changed the scheduler is informed if the new state vector specifies action.

Although protocol emulation in this way is compatible with the network properties of reliability and deadlock resistance, direct access to the network through conventional blocking I/O statements is not, and the latter is therefore excluded.

2.3. Protocol Multiplexing

Every communication in the protocol-based multiprocessor is part of a protocol that is interpreted by the system. Furthermore, the multiprocessor

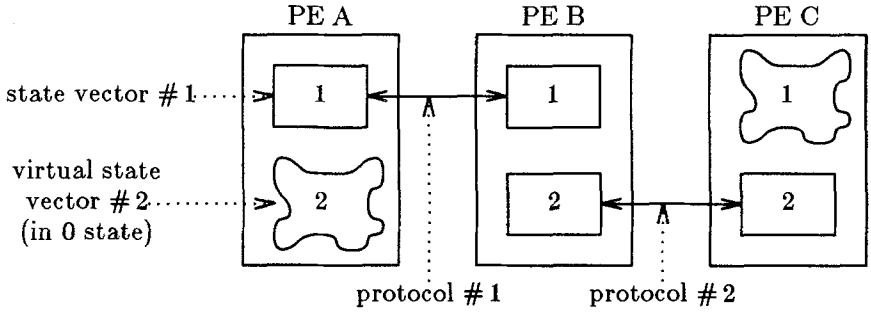

Fig. 4. Illustration of protocol multiplexing.

supports many independently operating protocols. This facility requires two things: every message must be tagged with a *protocol number* to identify with which protocol the message is associated, and there must be a state vector to record data and state information about a protocol, for each protocol interacting with a particular PE.

A matter of practical concern arises here. Experience, and analogy to virtual memory on a conventional computer, suggest a virtual space of protocols that is larger than the number used at any one time. Also, most protocols interact with only a few PEs, leaving their state vectors on other PEs in the 0 state (initialization state). The circuit simulator example, presented later, uses a protocol for each wire in the input circuit. Net-numbers of wires become protocol numbers. Only the two processors that have the ends of a wire use a protocol, although all must have the protocol number in their protocol space. Protocol numbers of 32 bits seem appropriate and so are programs where only a half dozen protocols ever leave the 0 state-implying that 6 out of 2^{32} state vectors are in a nonzero state. This suggests that the system should allocate state vectors on demand and deallocate them when no longer necessary. A system managed heap, or some similar structure, is used.

Figure 4 illustrates protocol multiplexing. The PEs labeled *A* and *B* are interacting via protocol number 1, and *B* and *C* via number 2. These two instances of the protocol are functionally independent. While PE *B* must have a state vector allocated for each of the two instances of the protocol operating on that PE, *A* and *C* require only one each. Here, the virtual protocol state facility avoids allocating memory for these state vectors until they are accessed, either by message receipt or user program access, and when accessed it appears in the 0 state. A protocol may interact with more than two PEs, although this is not illustrated.

When allocating instances of protocols, it is also necessary to associate input and output functions with the state vector. This suggests that a *protocol type* should be included with each protocol number. Protocols of the same type would have state vectors of the same size and the same input and output functions, whereas protocols of different types would not. A new protocol could therefore be added to a running system by having the operating system bind a new type to information about state vector size and input and output functions.

Figure 5 illustrates facilities for the dynamic allocation of protocols. Figure 5 illustrates a PE with two different types of protocols declared, identified by squares and triangles. For each protocol with a different behavior the PE maintains a heap for state vectors that have left the initialization state, and a freelist for state vectors awaiting demand allocation. In addition, an input and output function is maintained for

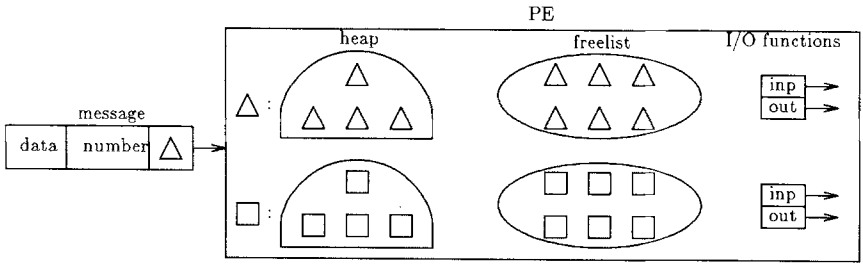

Fig. 5. Dynamic protocol allocation.

each protocol type. When a message arrives, the PE looks for the state vector corresponding to the number field in the message in the heap corresponding to the type specified in the message. If such a state vector is not found, a state vector is moved from the freelist to the heap and set to the 0 state. The input function is then executed with the message and selected state vector as arguments. An application program can access a state vector, possibly invoking demand-allocation, by specifying a type and number in a system call. Finally, a program can define a new protocol behavior by specifying input and output functions and a pool of state vectors to become the freelist.

2.4. Interactions with the User

One method whereby the user interacts with protocols is by directly examining and changing state vectors in an indivisible operation. In a typical case the user would enter a critical region, change the state vector, notify the system that the state is active, and then leave the critical region. The user could then poll the state part in a busy-wait until the protocol enters an idle state. Additionally, a task queue could be added to each PE and protocol definitions could be extended to allow protocols to enter and leave the queue. The task queue would be able to interrupt the CPU. These features have been tested in software on the BTL Hypercube.

3. AN EXAMPLE PROTOCOL

This section illustrates protocol design with a detailed example. The example chosen is a simple message-passing implementation of shared memory. The example was chosen because the semantics of shared memory are well known and this implementation is simple if not efficient.

Figure 6 illustrates the chosen approach to shared memory. A protocol and a *home PE* are associated with each memory word. Within

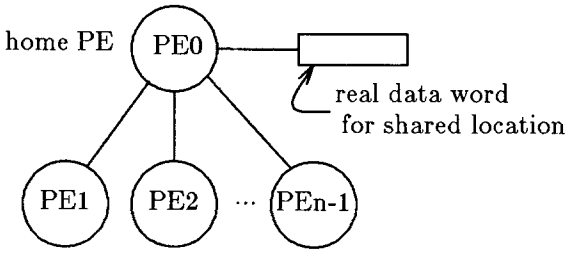

Fig. 6. Simple implementation of shared memory.

the home PE is a memory word (shown as a rectangle) which has the real value of the shared memory location. Accesses to this word from within the home PE are made by conventional accesses to this word. Accesses from other PEs are done by sending a message of type *Read* or *Write* to the home node and waiting for an *Ack* message. The protocol is consistent with memory semantics where the read or write occurs at an unspecified time during the period between the *Read* or *Write* and the *Ack*.

A state transition diagram for the protocol executed by nonhome PEs is illustrated in Fig. 7. The protocol is normally in the *IDLE* state. To start a read or write operation the application program changes the state from *IDLE* to *READ* or *WRITE*, moving the argument to the data portion of the state vector on writes. If necessary, these operations are done in a critical region to assure they are atomic. The system is informed that the state vector is in an *active* state, indicating that the output function will generate a message.

When the network is ready to accept an output message, which may be immediately or after a finite delay, the protocol scheduler will invoke the output function. The output function will send a *Read* or *Write* message and change the state to *WAIT*. The data word is sent in the data portion of a *Write* message.

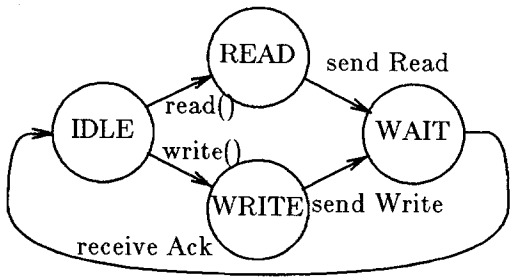

Fig. 7. Non home node protocol.

When an *Ack* message is eventually received, the input function changes the state to *IDLE* and moves the data portion from the *Ack* message to the state vector. The return values for reads is taken from the data portion of the state vector.

The input and output functions and the code to do a write (on the nonhome PE) are illustrated by Fig. 8 in C. Both the input and output functions accept a pointer to the state vector as an argument; the state vector is a structure with attributes *state* and *data*. The input function takes a pointer to a message as an argument; the message is a structure containing *type*, *origin*, and *data* fields. Both the input and output functions return a nonzero value when the state vector returned is *active*, meaning it will generate an output message. The write function instead uses the statement *activate(s)* to inform the system that the state vector is in a condition where it will generate an output message. The identifiers *IDLE*, *READ*, *WRITE*, and *WAIT* are enumerated constants representing the different

```
struct state_vector {
   enum { IDLE, READ, WRITE, WAIT } state;
   int data;                              /* data-to-write or read data */
   enum { NOT, WAITING } bits[n]; } ; /* only used on home node */

struct message {
   enum { Read, Write, Ack } type;
   int origin;                            /* originating PE */
   int data; } ;

input_function(s, m) state_vector *s; message *m; {
   s->state = IDLE;                       /* state part of state vector */
   s->data = m->data;                     /* data part of state vector */
   return(0); }                           /* indicates no output message */

output_function(s) state_vector *s; {
   if (s->state == READ)                  /* CPU requested read */
     /* send Read message */
   else if (s->state == WRITE)            /* CPU requested write */
     /* send Write message with s->data */
   else return(0);                        /* indicates no output message */
   s->state = WAIT;                       /* change state */
   return(0); }                           /* indicates no output message */

write(x, s) state_vector *s; {
   /* enter critical region */
   s->data = x;                           /* data part of state vector */
   s->state = WRITE;                      /* request write */
   activate(s);                           /* put on activity queue */
   /* leave critical region */
   while (s->state != IDLE) ; }           /* busy wait until done */
```

Fig. 8. Non home node protocol functions.

```
input_function(s, m) state_vector *s; message *m; {
   if (m->type == Write) s->data = m->data;/* do the real write */
   s->bits[m->origin] = WAITING;     /* note originating PE */
   return(1); }                      /* will send A msg */

output_function(s) state_vector *s; {
   int pid = find(s->bits);          /* bit position of a 1 bit */
   if (pid < 0) return(0);           /* no output message this time */
   s->bits[pid] = NOT;               /* turn off bit */
   /* send Ack message to pid with m->data */
   if (find(s->bits) < 0) return(0); /* no output message next time */
   return(1); }                      /* output message next time */
```

Fig. 9. Home PE protocol functions.

states of the protocol, and *Read*, *Write*, and *Ack* similarly represent message types.

Figure 9 illustrates the input and output functions for the home PE. These are somewhat less elegant because the unbounded fanout at the root node in Fig. 6. The home PE must record in its state vector whether or not each other $n-1$ PEs is awaiting an acknowledgement—using n bits (one unused). The size of the state vector limits the utility of this protocol; a 100K PE multiprocessor would require a state vector of 12.5K bytes, which would be clumsy. In practice, I use a tree with fanout f and $O(\log_f n)$ height.

The input function copies the data from the message to the state vector, for write messages only. It then sets the kth bit, where k represents the number of the originating PE (the *origin* field of the message). The output function simply sends *Ack* messages to any PE whose bit is set. This is shown by use of the function *find* which returns the index of the any set bit, or a negative value if there are none.

4. TOPOLOGY AND TREES

Trees are important in a variety of contexts. Adding more levels to the one level tree with illustrated in Fig. 6 saves memory and improves performance under most conditions. A later section uses trees for broadcasting and synchronization. Indeed, the concept of tree-like topologies appears to be of sufficient generality to warrant direct support by the operating system or hardware.

4.1. Tree-Based Protocols

Figure 10 illustrates a binary tree generated by the *parent function* $p(x) = (x-1)/2$. The resultant tree has some desirable properties: the num-

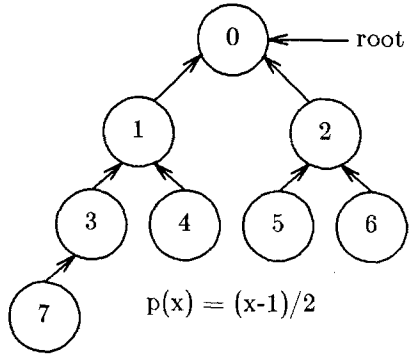

Fig. 10. A tree generated by a parent function.

ber of relatives (parent plus children) at each node is three or less, it is of logarithmic depth, and it is defined by a simple function.

When tree topologies are defined by functions, topology independence of protocols is obtained. As a first step, assume the root of a tree is designated node 0. For each tree node, its parent is designated as relative 0, and its children are numbered sequentially starting at 1. The following two functions define a tree:

$R(p, n)$ is the tree node that is the nth relative of tree node p.

$N(p, c)$ is the relative number of tree node c when viewed from tree node p.

Protocols can be described entirely in terms of the R and N functions. Input messages are considered only in terms of which relative they came from and output messages are sent to specific relatives rather than PE numbers.

4.2. Hashing of Tree Roots

If tree node n were to go on PE n, a bottleneck would develop because PE 0 has all the roots. A solution is to not apply the tree functions directly to PE numbers, but to apply them to an isomorphic mapping of the PE numbers. Let the function designated M_r: $P \to P$ be a one-to-one mapping of PE numbers to PE numbers, with the additional property that $M_r \cdot 0 = r$. Further define

$$R'(p, n, r) = M_r \cdot R(M_r^{-1} \cdot p, n) \quad \text{and} \quad N'(p, c, r) = N(M_r^{-1} \cdot p, M_r^{-1} \cdot c)$$

R' and N' thereby form definitions of a tree which is rooted at r. The protocol number can be hashed to generate r.

4.3. Topology Quality

A protocol correctly parameterized in terms of the functions R and N operates correctly regardless of the definitions of these functions. The performance may vary considerably, however.

The number of children at each node must be large to minimize the number of sequential messages necessary for one signal to span a tree. A large fanout increases speed. On the other hand, the number of bits in the state vector in a node is frequently dependent on the fanout of that node. Recall that the home node in the shared memory protocol required a n bit state vector, where the fanout of the node was $n-1$ (one bit was unused). Therefore, a small fanout decreases memory use. Topologies trade off memory in the state vector versus speed.

A second consideration is the match between the network topology and the communications paths used by the tree topologies. Consider, for example, a hypercube viewed as a Banyan network. The set of paths from any node to all other nodes forms an easily computed tree where every parent-child connection corresponds to a single physical communication path. The resulting "hypercube tree" on an n PE hypercube has a maximum fanout of $\log_2 n$. By contrast, the tree generated by the parent function $p(x) = (x-1)/2$ has a maximum fanout of only 2, but parent-child connections may require costly message store-and-forwarding. Trees which both can be embedded into hypercubes, and are binary, are known [see Ref. 17], but the functions defining the trees are complex. Here, topology trades off complexity for speed.

With my multiprocessor strategy, topology can be the last thing to worry about. Furthermore, if topology is an issue, it can be dealt with without rewiring the machine. Topological considerations are irrelevant when the programmer is trying to write a functionally correct program, because the qualitative behavior of protocols is independent of topology. Furthermore, if protocols are specified by generalized tree functions, most of the effort in specifying a protocol can be reused with a different topology. Topology is reduced to rough performance measures for primitives when a programmer is formulating an approach to an application. A library of important primitives coded with different topologies and costs so that all a programmer would have to do is pick the right primitive.

5. PERFORMANCE ANALYSIS

The performance of a primitive can be abstracted into simple expressions even though it may depend on both topology and functional

aspects of the protocol. The diagram in Fig. 11 compares the elapsed times of two $f \& a$ protocols. The "time line" for each PE, combining unit (C) and memory (Mem) is illustrated as a vertical column. Figure 11 shows the activity if all PEs were to attempt a $f \& a$ simultaneously.

The left part of Fig. 11 illustrates a protocol with one adder colocated with the memory cells (column Mem). Figure 11 illustrates PEs doing $f \& a(x, 1)$, where x is zero initially. The PEs send $f \& a$ messages directly to Mem and block awaiting a reply. Mem, the root of a one-level tree like Fig. 6, receives values, adds them to memory, and sends the answer back. If there are many simultaneous requests, Mem must process them sequentially. If there is only one $f \& a$ request, the elapsed time will be $4M + 2L$; if there are n simultaneous requests, the maximum elapsed time will be $(2n + 4)M + 2L$. This is a lousy protocol for large n.

The right part of Fig. 11 illustrates a protocol with a combining network[9] (column C) to condense data before the memory cells (column Mem). The single combining unit represents a logarithmic depth tree structured network that might appear in a larger machine. The PEs send $f \& a$ messages to their parents in the combining tree. C combines simultaneous requests $f \& a(x, a)$ and $f \& a(x, b)$ into the request $t = f \& a(x, a + b)$, t representing the returned value. C then replies with values t and $t + a$, respectively. The values 0 and 1 are returned to PE1 and PE2, which is consistent with the properties of $f \& a$ (although different from values 1 and 0 as illustrated on the left—which are also consistent). If there are

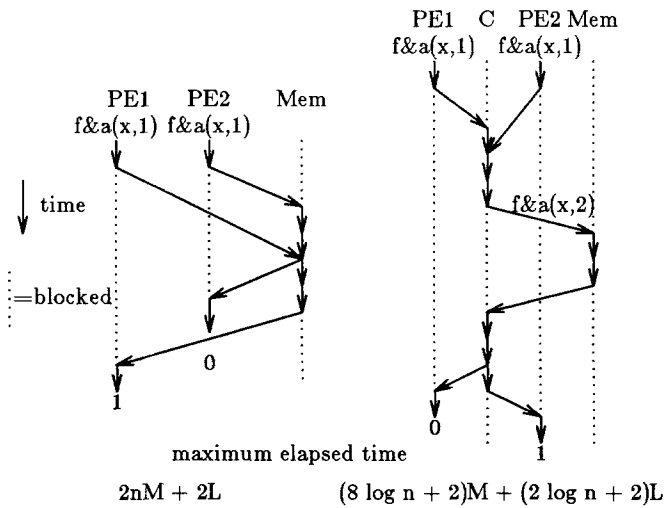

Fig. 11. Two $F \& a$ protocols.

many simultaneous requests, the combining network can reduce them to one request and disperse the proper answers in logarithmic time. Each combining stage introduces a worst-case elapsed time of $8M + 2L$, and the *Mem* unit introduces $2M + 2L$. The worst-case time for simultaneous accesses on a n node multiprocessor is $(8 \log_2 n + 2)M + (2 \log_2 n + 2)L$. A programmer would probably abstract this elapsed time information into "$O(\log n)$ *cost for everything.*"

The difference between n and $\log n$ costs should be large in absolute terms, and might make the difference between a multiprocessor system being effective or not. It is therefore crucially important that coarse timing measures (say to within a factor of two) be considered even when the primary objective is improving programmer productivity.

6. THE MASTER-AND-SLAVES PRIMITIVE

It seems axiomatic that programming primitives should be simple enough to be fast, but sophisticated enough to do what a programmer really wants. Programmers can use shared memory protocols[7,9] freely since their access rate is nearly as high as the instruction rate of the CPU. In a prototypical use of shared memory, however, multiple accesses are used to set a semaphore, do something, and then release the semaphore. The need for multiple uses counters the speed advantage. By contrast, this section presents a protocol for the MS primitive that is complex and has nonconstant time ($O(\log n)$) operations. In a single usage, however, the MS primitive directly implements a major portion of the Master-and-Slaves Plan. A well designed MS primitive might be more effective than simpler and faster primitives, such as shared memory.

The MS primitive can be viewed in terms of how it works (a multicast pipe) or in terms of how it is used by the programmer (an abstraction of sets). As a multicast pipe, it consists of objects in a globally accessible space with several defined operations. The objects have *connect* and *disconnect* operations which add and subtract slave connections. A *write_to_all* operation can be done by any PE and the associated data satisfies *read* operations by all connected PEs. Reads are optionally acknowledged by a *read acknowledge* operation which a writer can wait for by a *wait for acknowledge* operation. Data can be sent to one reader with a *write to one* operation.

To view the primitive as a set representation instead of multicast channels, the names *enter set*, *leave set*, *send command*, *get command*, *acknowledge command*, *operation done*, and *send to one* are more mnemonic.

The goal of the protocol design for the MS primitives is to be fast even

when the number of connected PEs in a set can vary between zero and n, the size of the multiprocessor. The approach is to base all the operations on trees, including all the PEs that are in the set, but as few others as possible. The structure and algorithm for setting up the tree are interesting. Algorithms to send data and collect acknowledgments are obvious once a tree exists.

Figure 12 illustrates the tree connection protocol. Assume PEs 2, 4, and 5 enter the tree in that order. PE 2 enters the tree by sending a *Connect* message to its parent and receiving an *Ack* in reply, indicating that it is a member of a tree rooted at PE 0. At this point, the tree has only the double-line edge shown in Fig. 12. PE 4 enters the tree by sending a *Connect* message to PE 1, which relays the message to PE 0; the root replies with an *Ack* message to PE 1, which is relayed to PE 4. PE 1 becomes a member of the tree even though it is otherwise uninvolved. PE 5 enters the tree by sending a *Connect* message to PE 2, which is already a member of the tree. PE 2 notes that its left child is a member of the tree and replies immediately with an *Ack* message. The *Ack* signifies membership in a tree rooted at PE 0, even though no message went to PE 0. The resulting tree is the part of Fig. 12 with solid and double-lines. Since *Connect* messages propogate only until they reach an already connected PE, no PE will receive more than two *Connect* messages. No connection involves more than $2 \log_2 n$ messages, which makes this a good algorithm.

```
struct state {
enum {NO, YES} local_connect;
enum {NO, WAITING, YES} i_am_connected;
enum {NO, UNACKNOWLEDGED, YES} got_connect_from_child[f];}
```

This data structure shows the parts of the state vector required for the connection protocol. The occurrence of a local connection operation is recorded by the *local_connect* attribute. The three states of a PEs interaction with its parent are represented by the *i_am_connected* attribute. Initially there is no interest in a connection—value *NO*. This state is

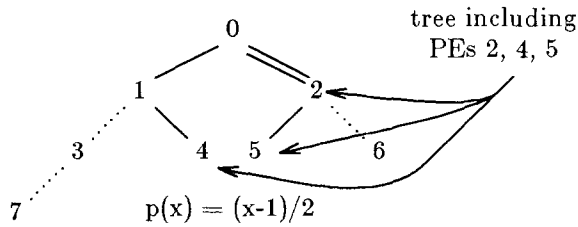

Fig. 12. Operation of the tree connect protocol.

changed to *WAITING* when a *Connect* message is sent to the parent, and later changed to *YES* when the *Ack* message is received. Since a *Connect* message is only sent on the transition of the *i_am_connected* attribute from *NO* to *WAITING*, no more than one *Connect* message can be sent. The subprotocol with each child has three states, which are stored in the *got_connect_from_child* attribute. There is initially *NO* interest in connection. The receipt of a *Connect* message is noted by changing the attribute to *UNACKNOWLEDGED*—which also indicates that an *Ack* message is expected by the child. When an *Ack* is sent, the attribute is changed to *YES*. Designing protocols is like specifying things with petri nets or programming in APL—you get the hang of it after awhile.

6.1. Observed Results

Table I compares various elapsed times for the MS primitive. The program, that runs on the BTL Hypercube, has one master (at a time) and 63 slaves. The master sends commands to the slaves via a tree structured MS primitive. A program run averages 1800 command-acknowledge cycles originating from nine different master PEs. The tree is generated by the parent function $p(x) = (x-1)/f$, f is the fanout. Table I gives the total elapsed time for both dispersion and collective acknowledgment of a command to 63 PEs. Note a broad minimum. The time increases for small fanouts because the tree height is greater, causing many message relays and associated L delays. For larger fanouts, the time increases because a node communicates with its children sequentially, incurring many M delays. The BTL Hypercube has performance values $M = .45mS$ and $L = .3mS$ for random communication. The measured figures are slightly better than might be predicted, possibly because the parent function generates many short hypercube paths.

7. PROGRAMMING EXAMPLE–A CIRCUIT SIMULATOR

Integrated circuit simulation is an important computationally intensive application. Circuit simulations that use exact transistor models and accurately model the analog functional and timing behavior of integrated

Table I. Timings for MS Primitive with Different Fanouts

fanout	1	2	3	4	5	6	8	12	15	30
mS/cycle	170	22	19	18	21	21	21	29	36	62

circuits are currently applied to portions of integrated circuits with around 100 transistors. It is important to industry, however, that whole integrated circuits, containing perhaps 1,000,000 transistors, be simulated. Whole integrated circuits can currently be simulated only by abstracting the analog and timing behavior of many small portions of the circuit and then functionally simulating the entire circuit with these abstractions. Functional simulation is inaccurate at modeling timing and analog properties. This section discusses a distributed algorithm for a simulator midway between circuit and functional simulators. By bringing more computational power to bear on a simulation task, this simulator[18] permits more extensive simulation of chips during the design cycle, and might therefore speed progress in the IC industry.

7.1. Uniprocessor Circuit Simulation

The type of simulator discussed here divides the simulation into intervals (Δt) and repeatedly computes the voltage on each wire at time $t + \Delta t$ based on voltages at time t.

```
for each timestep
  for each element
    read V(t) from inputs, simulate, and write V(t + Δt) to outputs
```

The Plan shown here must be merged with (what I call here) the Simultaneous Update Plan. This Plan assures that the value computed for a wire at time $t + \Delta t$ is really based on voltages at the input of the circuit element at time t. Simply associating a variable with each wire to hold its voltage does not work. When a wire goes from the output of one element to the input of another, and the first element happens to be updated first, then the second element is updated using the new voltage value. A common uniprocessor version of the Simultaneous Update Plan, shown below, associates two variables with each wire, one for an *old* value and one for a *new* value. When each circuit element is updated values from the *old* variables are used to compute values for the *new* variables. A second phase iterates over each circuit element a second time moving the *new* variable to the *old* variable.

```
for each element
  new voltage = update(old voltage)
for each element
  old voltage = new voltage
```

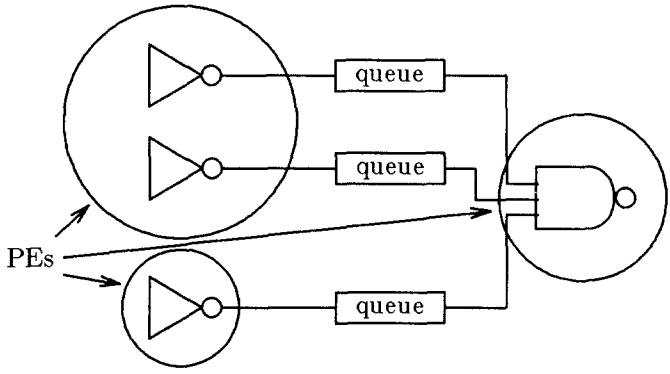

Fig. 13. Multiprocessor simulator with queues.

7.2. Multiprocessor Circuit Simulation

Figure 13 illustrates a multiprocessor Plan for circuit simulation. The Simultaneous Update Plan is managed by queues that are written by circuit elements with outputs and read by circuit elements with inputs. During initialization, one voltage sample is put into each queue. In a one-step simulation, the number of voltages in some queues would follow the sequence 1-2-1, and some 1-0-1. In an asynchronous simulation, a quickly simulating region might encounter an empty input queue and have to wait.

Figure 14 illustrates the MS Plan in the context of the circuit simulator. The definition of the circuit simulation problem requires that there be a person running the program issuing commands such as *simulate*

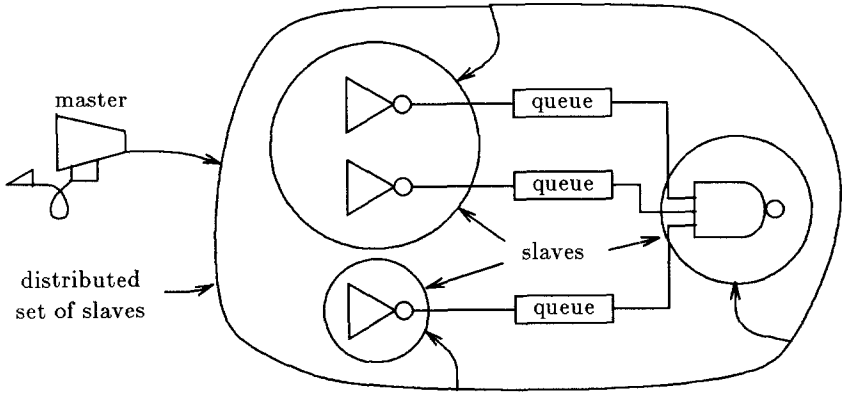

Fig. 14. Simulator control plan.

for 100 ns. Such a command must be delivered to every slave with circuit elements, which simulate until done, and then participate in a collective acknowledgment directed toward the master. The master then decides if more simulation is in order or if the answer is to be printed.

7.3. Variants of the Simulator

Several different versions of this circuit simulator have been studied at Bell Labs and are summarized in Table II. The algorithm described earlier suggests that each PE synchronize after each simulation time step to avoid unbounded filling of the queues. The synchronization necessary to separate timesteps is less general that provided by MS; specifically, no data needs to flow for this synchronization. A special synchronization protocol was developed that has higher performance than MS, and this version is called *synchronous.* A version of the simulator was tried where each region asynchronously updates voltages when a new set of input voltages are ready at the input queues and all the output queues have at least one empty location for a new voltage. The asynchrony inherent in this version improves load balancing by allowing temporarily compute-bound elements to fall behind the rest of the simulation without incurring idle time on some PEs. This version is called *asynchronous.* A kernel for the Linda language was developed by Lucco[19] and a simulator version, called *linda* was tested. The speedup factors are summarized in Refs. 18 and 19 for an 6K transistor fuzzy logic chip.

A timing analysis based on the program decomposition and characteristics of the fuzzy logic chip is instructive. Let T_{inner} be the time to read input, simulate one region, and write output. We have $T_{\text{inner}} = T_{\text{sim}} + 2\alpha F_g M$, where $T_{\text{sim}} \approx 1.0mS$ represents the time for a simulation step and $F_g \approx 7.6$ is the average number of wires per region. The previously stated value $M = .45mS$ is valid for the synchronous simulator, but our asynchronous queue protocol is poorly coded, resulting in $M = .85mS$ for

Table II. Distributed Objects in Various Simulation Versions

	synchronous	asynchronous	linda
MS's	1	1	none
queues	3700 w/flow control	3700 datagram	none
synchronizers	1	none	none
tuple spaces	none	none	1
error message paths	1	1	1
speedup on 64 PEs	18	6	*n/a* (17)

the asynchronous version. Factor α represents the number of messages generated for each voltage sample. For the asynchronous version, $\alpha \approx 1.2$ because each voltage sample generates a message, and 20% of the time an acknowledge message travels in the reverse direction. If the voltage on a wire is stable, there is no need to resend the voltage; and the synchronous version does not need to send a synchronization message either. An evaluation with the synchronous version sends messages only when signals are changing, corresponding to $\alpha = .06$. Each simulation time step is modeled by $T_{\text{timestep}} = L_b(mT_{\text{inner}} + L + T_{\text{sync}})$, m is the average number of regions per PE. All the messages in the inner loop are sent concurrently (see left part of Fig. 2), so only one L delay applies per time step. The synchronous version explicitly synchronizes at this point, adding a cost of $T_{\text{sync}} = \log_2 n(2M + L)$. Furthermore, a load balance factor L_b should be applied that multiplies the elapsed time accumulated so far. The fuzzy logic chip has $L_b = 2.0$ for synchronous and $L_b = 1.25$ for asynchronous. Time in the MS primitive is insignificant. Evaluating these expressions, we get a speedup of 17 for synchronous and 3 for asynchronous, which is as close as we expect to 18 and 6.

A speedup factor of 20 is respectable in absolute terms, but some explanation is in order. Table III illustrates measured and projected speedup figures for the BTL Hypercube, a nominally faster (ficticious) machine, and the machine proposed in a later section. The asynchronous simulator has high communication overhead. With better hardware, it would be very good; currently it is not viable. The synchronous simulator incurs about a 50% overhead because of imperfect load balance. Analysis and experience indicate that this figure is *smaller* for larger chips. Overhead increases as the number of PEs increases, *unless* the size of the simulated chip increases proportionately. Fortunately, the interest in circuit simulation centers on being able to simulate large chips on large machines. Existing hardware (1024 PE Ncube) would give an unprecedented speedup of 280 if the 26% efficiency could be maintained. Testing this prediction is future work.

Table III. Projected Simulator Performance

machine	M	L	synchronous efficiency	asynchronous efficiency
BTL Hypercube	.45/.85 mS	.3mS	26%	5%
slightly faster	100 uS	100 uS	41%	30%
proposed later	100 nS	1 uS	50%	80%

8. PROGRAMMING EXAMPLE–QUICKSORT

While the previous example illustrated the use of protocols, the example was not complicated enough to require them. A quicksort algorithm is presented here that uses the MS abstraction in an easy-to-understand way,[20] but in a way that would be difficult to implement directly. For the sake of brevity, only the inner loop of the algorithm is presented. The algorithm sorts an unordered set of data presuming the data is initially in memory and leaves the list in memory as a tree in nondescending order.

The conventional quicksort algorithm[21] is first presented with emphasis on those aspects important to the set based algorithm. Figures 15–17 illustrate quicksort. At the start of the algorithm there is a bag of elements (1–5), with no particular ordering. The first phase of the algorithm arbitrarily selects one element and designates it as the decision element. The second phase, Fig. 16, 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 the decision element and all other elements with the same key value, the *smaller* bag, with elements smaller than the decision element, and the *larger* bag, with the rest. The smaller and larger bags, Fig. 17, have the same form as the original bag, and recursion can be applied.

8.1. Quicksort with the Master-and-Slaves Primitive

The MS-based algorithm is also introduced with Fig. 15. The objects to be sorted are processes in the multiprocessor and the original bag is an MS primitive. The elements manifest their presence in the bag by trying to read from the MS primitive representing the bag.

The first phase, selecting a decision element, is illustrated in Fig. 15. Selecting a decision element starts with the master element (initially the

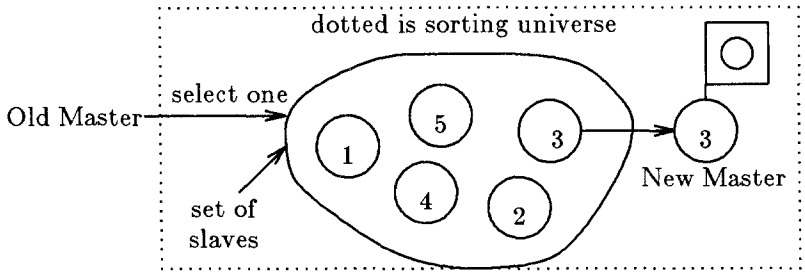

Fig. 15. Quicksort.

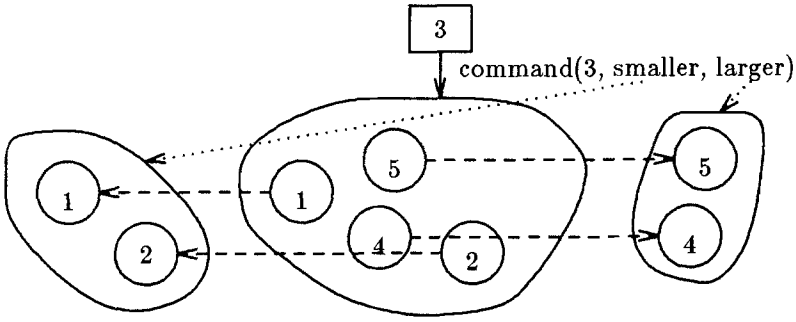

Fig. 16. Decision element selection.

main program), outside the bag. The master element does a *write to one* MS operation of a dummy value into the MS primitive representing the bag. According to the semantics of *write to one* each value written is read only once, hence the read succeeds in exactly one object. The decision element selection phase ends with one object knowing that it is the decision element (illustrated by the flag on object 3 in Fig. 15).

The second phase, bag partitioning, is illustrated in Fig. 16. Bag partitioning starts with the decision element just selected. The decision element creates two new bags by creating two initially empty MS primitives. The decision element then formats a message consisting of its key value and pointers to the two new bags, and does a write to the MS primitive representing the original bag. All the objects read this message.

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 original MS primitive.

The bag partitioning phase ends at the original decision element when the *wait for acknowledge* succeeds. At this point, there are two bags of

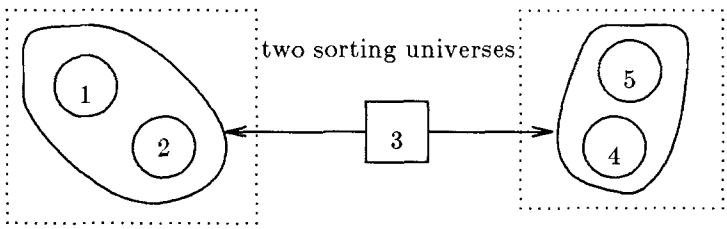

Fig. 17. Bag partitioning.

```
/* start being member of new_MS, sort key value */
for (;;) {                                  /* exit loop with break */
  old_MS = new_MS;                          /* read from a new bag */
  [ old_MS ? command(key, large, small) --> {/* compare-and-switch command */
    if (key > my_value) new_MS = large;
    else new_MS = small;
    add_to_MS(new_MS);                      /* declare membership in a new bag */
    acknowledge_command(old_MS); }
  old_MS ? send_to_one(/* no args */) --> {/* become-a-decision-element command */
    large = create_empty_MS();              /* empty bag for larger elements */
    small = create_empty_MS();              /* empty bag for smaller elements */
    old_MS ! command(value, large, small);/* send a compare-and-switch command */
    wait_for_command_done(old_MS);
    large ! send_to_one(/* no args */);  /* sort larger bag */
    small ! send_to_one(/* no args */); /* sort smaller bag */
    break; /* sort done for this element */ } ]
}
```

Fig. 18. Quicksort program in CSP/C.

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 by the decision elements doing a *send to one* of a dummy value to the MS primitives of the new bags.

Sorting ends when every element is a decision element. I suggest that the elements to be sorted be initially included in a distributed set and that the sorting be started by the master program doing a *write to all* operation. If each element then acknowledges its *read* when it becomes a decision element, a *wait for acknowledge* in the main program will complete only when sorting is done.

Figure 18 shows the inner loop of quicksort in *C* augmented with CSP-style communication statements. In the code, communication operations interact with distributed MS objects. Messages of type *command* are implicitly broadcasted from the master and collectively acknowledged, whereas *send_to_one* messages are transmitted from the master to an arbitrary slave.

8.2. Asymptotic Execution Time of Set-Based Quicksort

Since there is no universally accepted way of timing multiprocessors, the asymptotic execution time of the algorithm depends on the gamesmanship of the analyzer. To analyze quicksort, there are algorithmic and architectural issues. The choice or design of the timing model (an architectural issue) is a determining factor in the execution time per level.

The execution time of the complete algorithm is assumed to be the product of the number of recursion levels and the execution time per level.

The key to estimating the number of recursion levels is that the output of the algorithm is a random binary tree. [Assuming no duplicates in the input data; the result is the same or better if duplicates are allowed.] The number of recursion levels to sort n elements is simply the height, H_n, of a n node binary search tree. With a very nontrivial proof, Ref. 22 shows that asymptotically $H_n = 4.31107... \log n$ as $n \to \infty$. The order of this quicksort depends on the height being $O(\log n)$.

By analogy to the PRAM model of multiprocessors, quicksort will have an execution time of $O(\log n)$. A PRAM machine is a multiprocessor with n PEs addressing a common shared memory. Timing analysis assumes that every PE can access memory in unit time. Set-based quicksort does not use shared memory, so the model has to be modified. If we look at the architecture of a scalable shared memory machine such as the Ultracomputer,[9] we find that each memory access goes through a logarithmic-depth network. To implement its combining functions, the network processes addresses and data at each stage. I outlined earlier how the set operations used in quicksort can be implemented on a logarithmic-depth network with simple protocol processing at each stage. In a cavelier analogy to the PRAM model, I assign unit cost to the $\log n$ information transmission and processing operations of a tree operation. Execution time is therefore $O(\log n)$ for set-based quicksort, which is the same as quicksort on a PRAM machine.

Unit cost is assessed to each message transmission, reception, and handling operation in execution time analysis of other distributed machines. The cost associated with the logarithmic-depth network protocols would be $O(\log n)$ according to this model. Execution time is therefore $O(\log^2 n)$ for set-based quicksort, which the same as common sorting networks (although the best result for a sorting network is $O(\log n)$).

If this execution time analysis seems inconclusive, it is partly because multiprocessor timing models are inadequate. It seems that unit cost is assigned to whatever primitive the hardware implements—even if that primitive is costly. If you try enhancing the hardware by making it programmable, you lose because the cost basis changes. Timing estimates for algorithms go up even though programs run faster. Hence, I did the quicksort analysis twice, once with programmable hardware, and once moving the cost basis. For similar reasons, I avoided the I/O issue completely. These are topics for further research.

8.3. The Hardware-Operating System-User Division

Figure 19 illustrates a proposed design where a CPU can interact efficiently with distributed objects. Distributed objects are accessible in the address space of the CPU. Each distributed object would respond to several adjacent memory addresses—like a data structure—with different addresses corresponding to different functions. For instance, accessing one address might invoke a connection protocol to a queue, and reading another address might get data.

Access to a distributed object can complete in several ways. A read from a nonempty queue could complete immediately, putting the data values on the bus. An access that invokes a connection protocol could return immediately, but the function might be executed later. A read from an empty queue can complete only after the data arrives; the hardware could delay completion of the access through the facility that accommodates slow memory, or alternatively the hardware could signal a memory fault to turn control over to software. If the access path for distributed objects goes through memory management hardware, the operating system can exert considerable control while retaining fast access.

Figure 20 illustrates a state machine for emulating protocols. The *combinational emulator* updates state vectors, sometimes using additional input information, and sometimes generating an output message. The combinational emulator is generally divided into *formal state* and *data path* parts, like a microcoded CPU. The formal state part in analogous to microcode—there is a lookup table that describes the progression from one state to another. The data path part moves data between the input, output, and a small amount of storage. The formal state part directly controls the data path part, and the data path part generates a few signals that the formal state part monitors.

Table IV summarizes the three cycle types of the protocol emulation hardware. On a CPU access the protocol number comes from the address bus and is routed to the state vector memory. This applies the proper state

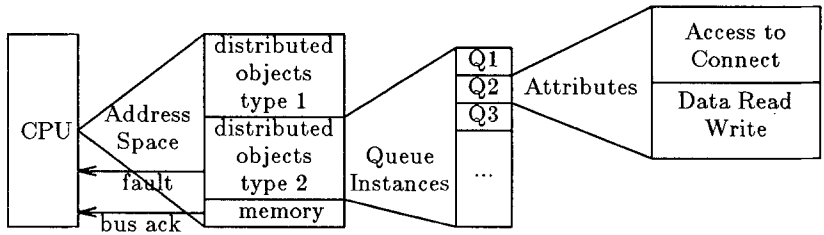

Fig. 19. CPU memory addressing.

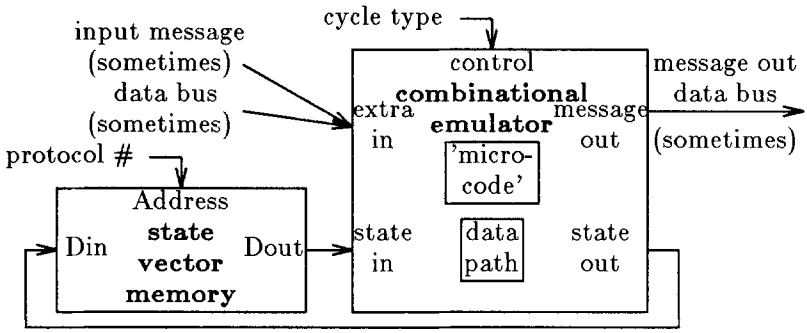

Fig. 20. Protocol emulation hardware.

vector to the combinational emulator. The data bus is connected to the input of the combinational emulator on write cycles, and to the output on read cycles. When messages are received, the protocol number is derived from the message header, causing the appropriate state vector to be updated. For output cycles, the protocol number is supplied by an *activity queue*. The activity queue records the current protocol number when a cycle indicates to the hardware the *next* cycle will generate output. When the output buffer is empty and the activity queue is nonempty, the hardware runs an output cycle using a protocol number extracted from the activity queue.

This design is a current research project at Bell Labs, and should be considered untested. The design should be capable of single cycle execution of CPU, input, or output functions for any of the protocols discussed in this paper. This would correspond to $M = 100nS$ for a 10 MHz clock, which is a speed improvement of 3000 over the BTL Hypercube. The current PE design approach is to use a microprocessor for a CPU and bit-slice components for the protocol hardware. We expect a PE to have the complexity of two microprocessors and fit on one pc board.

The mere proposal of this design lends some credibility to the idea of a cost effective multiprocessor with n between 10K–100K. There is now a

Table IV. Protocol Engine Cycle Types

cycle type	protocol # from	action
CPU	address bus	data bus = access(data bus, selected state)
input	input buffer	input_fcn.(input buffer, selected state)
output	activity queue	output = output_fcn.(selected state)

commercially available hypercube with $n = 1024$ using a one-chip CPU. To achieve my goal, the complexity on the CPU chip would have to double and n would have to increase by 10–100. Such advances are optimistic but not unreasonable.

CONCLUSION

A multiprocessor strategy has been proposed. With the strategy, programmers use programming Plans and scenarios adapted from software engineering to write functionally correct distributed programs. Rough execution time estimates can be obtained directly from the programming Plans and network parameters. Furthermore, a CPU architecture was outlined to show that protocols are amenable to speed improvements through hardware assistance. The successful coding of quicksort and circuit simulation proved the strategy can be effective for writing functionally correct programs. The fact that analytical execution time estimates for quicksort yielded linear speedup, and that both analytical and measured results for circuit simulation revealed good performance for 64 processors showed the strategy can produce efficient code. Of course, the generality of the strategy can only be shown with extensive use.

The key concept has been to divide the programming task into two parts. The lower-level part is the construction of distributed programming primitives through use of protocols. The higher-level part is the application of program decomposition techniques, of the psychological variety, to multiprocessors. Compatibility between these parts requires that the distributed programming primitives be independent of each other. This in turn required new hardware or operating system structures, including virtually allocated state vectors, input and output functions, and an activity queue for state vectors.

REFERENCES

1. A. Karp, Programming for Parallelism, *IEEE Computer*, pp. 43–57 (May 1987).
2. H. Katseff, Using Data Partitioning to Implement a Parallel Assembler, in preparation.
3. E. Soloway, Learning to Program = Learning to Construct Mechanisms and Explanations, *Communications of the ACM*, pp. 850–858 (September 1986).
4. R. Waters, The Programmer's Apprentice: A Session with KBEmacs, *IEEE Transactions on Software Engineering*, pp. 1296–1320 (November 1985).
5. D. Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages*, pp. 80–112 (January 1985).
6. L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem, *Communications of the ACM*, pp. 453–455 (August 1974).

7. 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 1984).
8. A. Birrell and B. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, pp. 39–59 (February 1984).
9. 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).
10. C. Seitz, The Cosmic Cube, *Communications of the ACM*, pp. 22–33 (January 1985).
11. C. A. R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, pp. 666–677 (August 1978).
12. D. Cheriton and W. Zwaenepol, Distributed Process Groups in the V Kernel, *ACM Transactions on Computer Systems*, pp. 77–107 (May 1985).
13. E. DeBenedictis, A Communications Operating System for the Homogeneous Machine. Caltech Computer Science Department Technical Report 4707, (1982).
14. H. Sullivan and T. Brashkow, A Large Scale Homogeneous, Fully Distributed Parallel Machine I, *Proceedings of the 4th Symposium on Computer Architecture*, pp. 105–117 (March 1977).
15. K. Günther, Prevention of Deadlocks in Packet-Switched Data Transport Systems, *IEEE Transactions on Communications*, pp. 512–524 (April 1981).
16. A. Danthine, Protocol Representation with Finite-State Models, *IEEE Transactions on Communications*, pp. 632–643 (April 1980).
17. A. Wu, Embedding of Tree Networks into Hypercubes, *Journal of Parallel and Distributed Computing*, pp. 238–245 (August 1985).
18. B. Ackland, S. 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).
19. S. Lucco, A Heuristic Linda Kernel for Hypercube Multiprocessors, Hypercube Multiprocessors 1987, (ed.), M. Heath, *SIAM*, pp. 32–37 (1987).
20. E. DeBenedictis, Multiprocessor Programming with Distributed Variables, Hypercube Multiprocessors 1986, (ed.), M. Heath, *SIAM*, pp. 70–86 (1986).
21. C. A. R. Hoare, Quicksort, *Computer Journal*, pp. 10–15 (April 1962).
22. L. Devroye, A Note on the Height of Binary Search Trees, *Journal of the ACM*, pp. 489–498 (July 1986).