Protocol-Based Multiprocessors

ERIK P. DEBENEDICTIS*

This short paper describes a programming environment on a hypercube-style multiprocessor that utilizes protocol-based programming primitives and which has been used to make an effective circuit simulator. The paper first describes the programming technique for this environment in terms of ***programming plans***. Programming plans are easily understood in terms of ***protocols*** or ***distributed programming primitives***, the emulation of which is the primary function of the runtime environment of the multiprocessor. A description is then given of the programming of the circuit simulator, which is unusually concise given the irregularity of the problem and the high degree of parallelism achieved.

1. *Introduction*. Let us start by considering the way that people think about computer programming from a psychological rather than the usual mathematical viewpoint.

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

The code shown above represents programming knowledge, or a plan, called the SENTINEL-CONTROLLED COUNTER-LOOP PLAN [1]. 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.

Figure 1 is a picture of a multiprocessor plan called the MASTER-AND-SLAVES, or SIMD PLAN (so called because the hardware of a SIMD multiprocessor operates this way).

* AT&T Bell Laboratories, Holmdel, New Jersey 07733.

The action in this plan 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 it can then perform its next activity.

An example of this plan is when a person runs 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 processing elements (PE's) of the multiprocessor, and they repeatedly input commands from the master, compute something

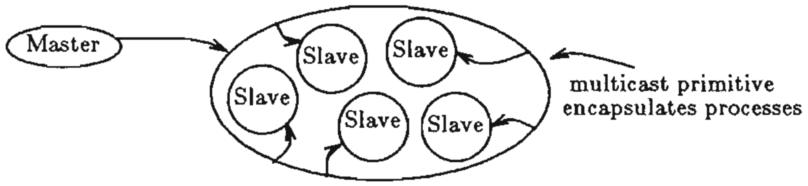

FIG. 1. MASTER-AND-SLAVES Plan

in conjunction with the other PE's, and collectively report completion to the master. Unlike a SIMD computer, however, the MASTER-AND-SLAVES plan is not restricted to having one master; [2] includes an example where many parts of a program are master for many other parts.

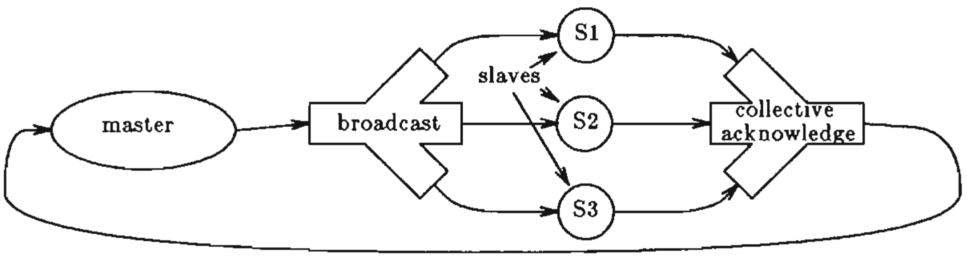

FIG. 2. Implementation of MASTER-AND-SLAVES

Figure 2 illustrates the information flow inherent in MASTER-AND-SLAVES. The master broadcasts commands to the slaves, and the slaves participate in some sort of collective acknowledgement protocol with other slaves, and the master, to indicate completion.

2. Programming Primitives and Protocols. I propose to represent distributed programming plans, or techniques, as manipulations of programming primitives. To some extent, this is already done. 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 an plan is most concisely represented as, say, broadcasting and collective acknowledgement, then this representation should prevail even if the target multiprocessor does not have that exact hardware.

When viewed as information flow on wires, programming primitives are nothing more than protocols. Again, this is done now to some extent. Protocols diagrams can be seen explicitly in the descriptions of shared memory [3] and RPC [4]. The combining elements in the fetch-and-add-based (*f&a*-based) Ultracomputer [5] use a protocol to remember which *f&a* locations have outstanding requests.

The reader may, for tutorial purposes, consider shared-memory locations and pipes (or queues) as typical instances of protocols. A protocol is associated with each memory location in a shared-memory computer, and the programmer interacts with these protocols by issuing read or write requests from any PE. It should be easy to imagine a variant of the pipe structures found in many conventional operating systems applied to a multiprocessor. A protocol would be associated with each pipe (or queue) and it could be read from one PE and written from another.

2.1. *Protocol Emulation.* The protocol-based multiprocessor executes protocols from a representation similar to the finite state representation [6]. 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***.[1] 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. The output function sometimes generates a message when applied to a state vector and at other times generates ϕ, indicating no message. As a pragmatic extension to the finite state concept, the input and output functions help the scheduler by specifying whether or not the state they return produces requires action.

2.2. *Protocol Multiplexing.* Every communication in a protocol-based multiprocessor is part of a protocol that is interpreted by the system. Furthermore, the multiprocessor supports an essentially unlimited number of 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. Ease of programming suggests a large virtual space of protocols of which only a small fraction is used. Also, most protocols interact with only a few PEs, leaving their state vectors on other PEs in the ***0 state*** (initialization state). For example, protocol numbers of 32 bits are 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 non-zero 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 necessary.

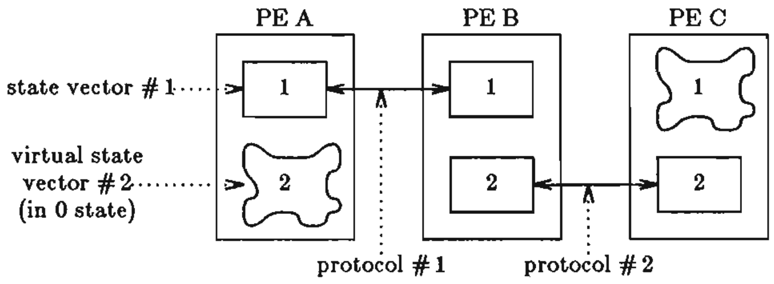

FIG. 3. Illustration of Protocol Multiplexing

Figure 3 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. In this case, 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.

3. *An Example Protocol.* This section illustrates protocol design with a detailed example.

1 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 example chosen is a simple message-passing implementation of shared memory. The example has been chosen because the semantics of shared memory are well known, and this implementation is simple if not efficient.

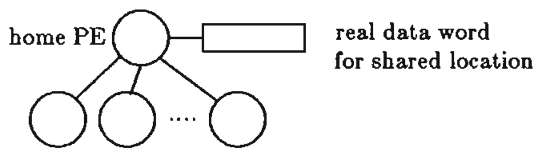

FIG. 4. Simple Implementation of Shared Memory

Figure 4 illustrates the chosen approach to shared memory. A protocol and a *home PE* (shown as the upper circle) are associated with each memory word. Within the home PE is a memory word (shown as a rectangle) which represents the actual value of the shared memory location. Accesses to this word from within the home PE are made by conventional accesses to this location. Accesses from other PEs are done by sending a R (read) or W (write) message to the home node and waiting for an A (acknowledge) message. The protocol is consistent with memory semantics where the actual read or write occurs at an unspecified time during the period between the R or W message and the A message.

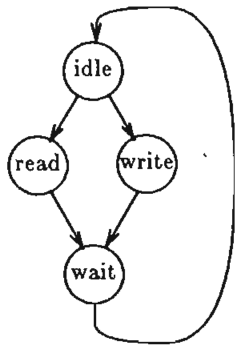

FIG. 5. Slave Node Protocol

A state transition diagram for the protocol executed by non-home PEs is illustrated in figure 5. 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*, writing a data word into the data part of the state vector if appropriate. 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, as opposed to a ϕ.

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

When an A message is eventually received, the input function changes the state to *idle* and stores the data portion of the A message in the data portion of the state.

The input and output functions and the application program code to do a write for this protocol (on the non-home PE) are illustrated below 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, and the message is a structure containing a *data* field. When the state vector returned by either the input or output function is not *active*, meaning it will not generate an output message, the function returns a 0, otherwise it does not. The write function uses the statement *activate(s)* to inform the system, that the state vector is in a condition where it will generate an output message. Finally, the identifiers *idle*, *read*, *write*, and *wait* are manifest constants representing the different states of the protocol.

```
struct state_vector {
   int state;                                /* idle, read, write, or wait */
   int data; } ;                             /* data-to-write or read data */

struct message {
   char type;                                /* R, W, or A */
   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 R message */
   else if (s->state == write)               /* CPU requested write */
      /* send W message with s->data */
   else return(0);                           /* indicates no output message */
   s->state = wait;                          /* change state */
   return(1);                                /* indicates 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 */
}
```

4. A Circuit Simulator. Integrated circuit simulation is an important task in industry, and may be the most computing intensive computer-aided design task. Circuit simulations which use exact transistor models and accurately model the analog functional and timing behavior of integrated circuits are currently applied to portions of integrated circits with around 100 transistors. It is important to industry, however, that whole integrated circuits, containing perhaps 100,000 transistors be simulated. Whole integrated circuits can only be simulated by abstracting the analog and timing behavior of many small portions of the circuit and then functionally simulating the entire circuit on the basis of these abstractions. Functional simulation is relatively inaccurate at modeling timing and analog properties. This section discusses a distributed algorithm for a simulator midway between circuit and functional simulators. The simulator discussed here [7] allows larger circuits to be simulated as part of the integrated circuit design cycle that has been previously possible.

4.1. Uniprocessor Circuit Simulation. The type of simulator discussed here divides the simulation into 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
    update voltages on output wires
```

The straightforward plan shown above 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 actually 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 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
```

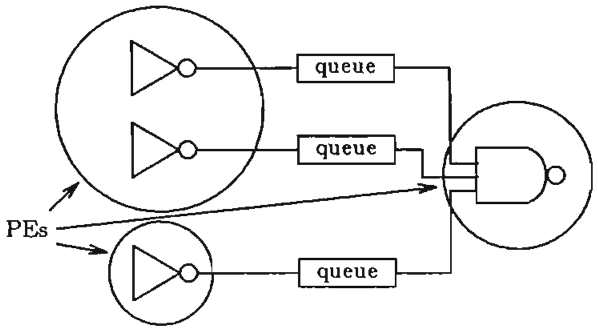

FIG. 6. Multiprocessor Simulator with Queues

4.2. Multiprocessor Circuit Simulation. Figure 6 illustrates a multiprocessor plan for circuit simulation. The SIMULTANEOUS UPDATE PLAN is managed by the use of queues which are written by circuit elements with outputs, and read by circuit elements with inputs. Different circuit elements may be on different PEs with only the requirement that the PEs refer to the wire by a single protocol number. If there is synchronization to assure that all PEs are computing for the same value of t, then there are a maximum of two voltage values in a queue.

Figure 7 illustrates the MASTER-AND-SLAVES 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 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 acknowledgement directed toward the MASTER. The MASTER might then decides if more simulation is in order or if the answer is to be printed.

5. Conclusions. The approach presented in this paper addresses the spectrum of issues

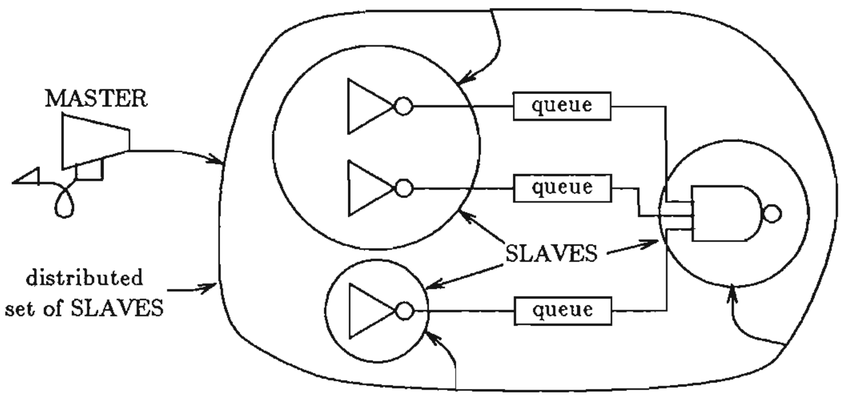

FIG. 7. Multiprocessor Simulator with Queues

between technology and programming technique consistently and with thought to future developments. It should be possible to write programs and build machines with up to 100,000 PEs and retain adequate efficiency. Research not reported here has explored computer architectures where protocol emulation is an intrinsic function of the hardware. The ***communication to computation*** cost ratio of a machine with such an architecture would be substantially lower, and would therefore support a broader class of applications (such as, for example logic simulation instead of just circuit simulation). An evolution in programming primitives is occurring in my laboratory as more programs are written; certain programming primitives (protocols) see increasing usage (and are enhanced) whereas others see decreasing usage (and are abandoned). There is a possibility that a standard set of distributed programming primitives will eventually emerge, in the same way as conventional computer hardware has standardized on 2's complement integers, pointers, and floating point as data types.

6. References.

[1] E. SOLOWAY, *Learning to Program = Learning to Construct Mechanisms and Explanations.* In *Communications of the ACM,* September 1986. Pages 850-858.

[2] E. DeBENEDICTIS, *Multiprocessor Programming with Distributed Variables.* In M. HEATH (ed.), *Hypercube Multiprocessors 1986* , August, 1985, pages 70-86.

[3] L. RUDOLPH, Z. SEGALL, *Dynamic Decentralized Cache Schemes for MIMD Parallel Processors.* In *Proceedings of the 11th Annual International Symposium on Computer Architecture,* June 1984. Pages 340-347.

[4] A. BIRRELL and B. NELSON, *Implementing Remote Procedure Calls.* In *ACM Transactions on Computer Systems,* February 1984. Pages 39-59.

[5] A. GOTTLIEB, R. GRISHMAN, C. KRUSKAL, K. McAULIFFE, L. RUDOLPH, M. SNIR, *The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.* In *IEEE Transactions on Computers,* February 1983. Pages 175-189.

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

[7] B. ACKLAND, S. AHUJA, E. DeBENEDICTIS, T. LONDON, S. LUCCO, D. ROMERO, *MOS Timing Simulation on a Message Based Multiprocessor.* In *Proceedings of the IEEE International Conference on Computer Design* , October, 1986, pages 446-450.