A PIPELINED, SHARED RESOUCE MIMD COMPUTER

Burton J. Smith
Denelcor, Inc.
Denver, Colorado 80205

Abstract -- The HEP computer system currently being implemented by Denelcor, Inc., under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn. In this type of organization, skeleton processors compete for execution resources in either space or time. In the HEP processor, spatial switching occurs between two queues of processes; one of these controls program memory, register memory, and the functional units while the other controls data memory. Multiple processors and data memories may be interconnected via a pipelined switch, and any register memory or data memory location may be used to synchronize two processes on a producer-consumer basis.

Overview

The HEP computer system currently being implemented by Denelcor, Inc., under contract to the U.S. Army Ballistics Research Laboratory is an MIMD machine of the shared resource type as defined by Flynn [1]. In this type of organization, skeleton processors compete for execution resources in either space or time. For example, the set of peripheral processors of the CDC 6600 [5] may be viewed as an MIMD machine implemented via the time-multiplexing of ten process states to one functional unit.

In a HEP processor, two queues are used to time-multiplex the process states. One of these provides input to a pipeline which fetches a three address instruction, decodes it, obtains the two operands, and sends the information to one of several pipelined function units where the operation is completed. In case the operation is a data memory access, the process state enters a second queue. This queue provides input to a pipelined switch which interconnects several data memory modules with several processors. When the memory access is complete, the process state is returned to the first queue. The processor organization is shown in Figure 1, and the over-all system layout appears in Figure 2.

Each processor of HEP can support up to 128 processes, and nominally begins execution of a new instruction (on behalf of some process) every 100 nanoseconds. The time required to completely execute an instruction is 800 ns, so that if at least eight totally independent processes, i.e. processes that do not share data, are executing in one processor the instruction execution rate is 10^7 instructions per second per processor. The first HEP system will have four processors and 128K words of data memory.

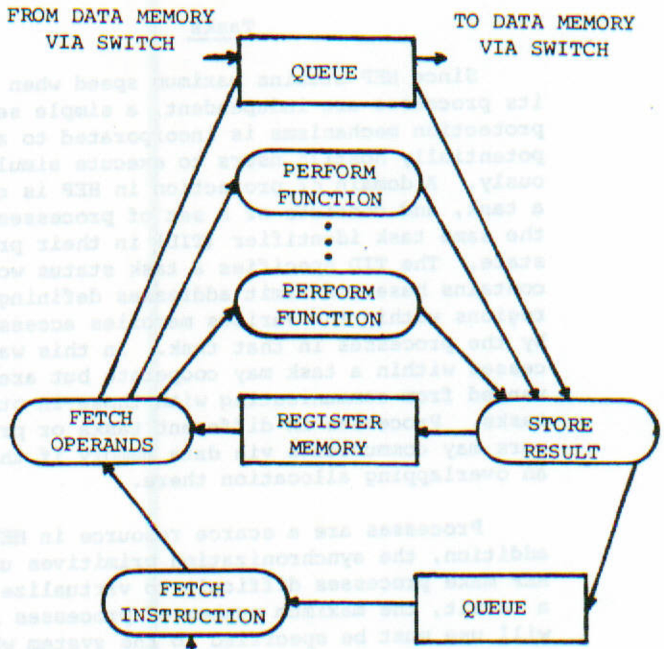

Figure 1. Processor Organization

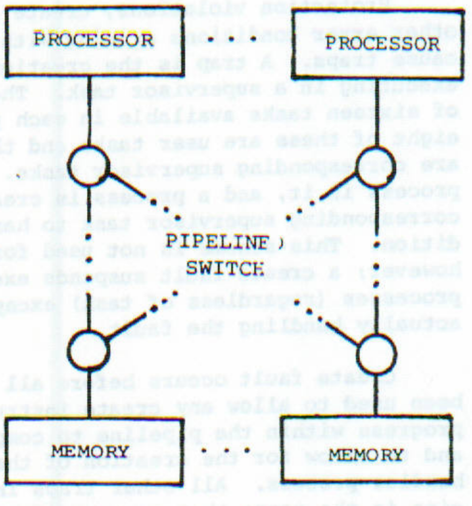

Figure 2. Overall System Layout

HEP instructions and data words are 64 bits wide. The floating point format is sign magnitude with a hexadecimal, seven-bit, excess-64 exponent. All functional units can support one instruction execution every 100 nanoseconds except the divider, which can support this rate momentarily but is slower on the average.

Tasks

Since HEP attains maximum speed when all of its processes are independent, a simple set of protection mechanisms is incorporated to allow potentially hostile users to execute simultaneously. A domain of protection in HEP is called a task, and consists of a set of processes with the same task identifier (TID) in their process state. The TID specifies a task status word which contains base and limit addresses defining the regions within the various memories accessible by the processes in that task. In this way, processes within a task may cooperate but are prevented from communicating with those in other tasks. Processes in different tasks or processors may communicate via data memory if they have an overlapping allocation there.

Processes are a scarce resource in HEP; in addition, the synchronization primitives used in HEP make processes difficult to virtualize. As a result, the maximum number of processes a task will use must be specified to the system when the task is loaded. It is the job of the operating system to insure that its total allocation of processes to tasks does not exceed the number available, so that a create fault (too many processes) can only occur when one or more tasks have created more processes than they were allocated. In this event, the offending task or tasks (not necessarily the task that actually caused the create fault) are removed from the processor.

Protection violations, create faults, and other error conditions arising within a process cause traps. A trap is the creation of a process executing in a supervisor task. There are a total of sixteen tasks available in each processor; eight of these are user tasks and the other eight are corresponding supervisor tasks. When any process in it, and a process is created in the corresponding supervisor task to handle the condition. This scheme is not used for create fault, however; a create fault suspends execution of all processes (regardless of task) except those actually handling the fault.

Create fault occurs before all processes have been used to allow any create instructions in progress within the pipeline to complete normally and to allow for the creation of the create fault handler process. All other traps in HEP are precise in the sense that no subsequent instructions will be executed from the offending task, a useful feature when one is trying to debug a concurrent algorithm.

Synchronization

The synchronization of processes in HEP is made simple by virtue of the fact that any register or data memory location can be used to synchronize two processes in a producer-consumer fashion. This requires three states in general: a reserved state to provide for mutual exclusion, a full state, and an empty state. The execution of an instruction tests the states of locations and modifies them in an indivisible manner; typically, an instruction tests its sources full and its destination empty. If any of these tests fails, the instruction is reattempted by the process on its next turn for servicing. If all tests succeed, the instruction is executed; the process sets both sources empty and the destination reserved. The operands from the sources are sent to the function unit, and the program counter in the process state is incremented. When the function unit eventually writes a result in the destination location that was specified in the instruction it sets the destination full. Provisions are made to test a destination full rather than empty, to preserve the state of a source, or to totally override the state of a source or destination with the proviso that a reserved state may not be overridden except by certain privileged instructions. Input-output synchronization is handled naturally by mapping I/O device registers into the data memory address space; an interrupt handler is just a process that is attempting to read an input location or write an output location. I/O device addresses are not relocated by the data memory base address and all I/O-addressed operations are privileged.

Switch

The switch that interconnects processors and data memories to allow memory sharing consists of a number of nodes connected via ports. Each node has three ports, and can simultaneously send and receive a message on each port. The messages contain the address of the recipient, the address of the originator, the operation to be performed by the recipient, and a priority. Each switch node receives a message on each of its three ports every 100 nanoseconds and attempts to retransmit each message on a port that will reduce the distance of that message from its recipient; a table mapping the recipient address into the number of a port that reduces distance is stored in each node for this purpose. If conflict for a port occurs, the node routes one of the contending messages correctly and the rest incorrectly. To help insure fairness, an incorrectly routed message has its priority incremented as it passes through the node, and preference is given in conflict situations to the message(s) with the highest priority.

The time required to complete a memory operation via the switch includes two message transmission times, one in each direction, since the success or failure of the operation (based on the state of the memory location, i.e. full or empty) must be reported back to the processor so that it can decide whether to reattempt the operation or not. The propagation delay through a node and its associated wiring is 50 nanoseconds. Since a message is distributed among two (or three) nodes at any instant, the switch must be two-colorable to avoid conflicts between the beginning of some message and the middle part of another. When the switch fills up due to a high conflict rate, misrouted massages begin to "leak" from the switch. Every originator is obliged to reinsert a leaking message into the switch in preference to inserting a new message. Special measures are taken when the priority value reaches its maximum in any message to avoid indefinite delays for such messages; a preferable scheme would have been to let priority be established by time of message creation except for the large number of bits required to specify it.

FORTRAN Extensions

Two extensions have been made to FORTRAN to allow the programmer to incorporate parallelism into his programs. First, subroutines whose names begin with "$" may execute in parallel with their callers, either by being CREATEd instead of CALLed or by executing a RESUME prior to a RETURN. Second, variables and arrays whose names begin with "$" may be used to transmit data between two processes via the full-empty discipline. A simple program to add the elements of an array $A is shown in Figure 3. The subroutines $INPUT and $OUTPUT perform obvious functions, and the subroutine $ADD does the work of adding up the elements. There are a total of 14 processes executing as a result of running the program.

```
C     ADD UP THE ELEMENTS OF
C     THE ARRAY $A
      REAL $A(1000),$S(10),$SUM
      INTEGER I
      CREATE $INPUT($A,1000)
      DO 10 I=1,10
      CREATE $ADD($A(100*I-99),$S(I),100)
   10 CONTINUE
      CREATE $ADD($S,$SUM,10)
      CREATE $OUTPUT($SUM,1)
      END

C     NOELTS ELEMENTS OF $V
C     ARE ADDED AND PLACED IN $ANS
      SUBROUTINE $ADD($V,$ANS,NOELTS)
      REAL $V(1),$ANS,TEMP
      INTEGER J, NOELTS
      TEMP=0.0
      DO 20 J=1,NOELTS
      TEMP=TEMP+$V(J)
   20 CONTINUE
      $ANS=TEMP
      RETURN
      END
```

Figure 3. HEP FORTRAN Example

Applications

As a parallel computer, HEP has an advantage over SIMD machines and more loosely coupled MIMD machines in two application areas. The first of these involves the solution of large systems of ordinary differential equations in simulating continuous systems. In this application, vector operations are difficult to apply because of the precedence constraints in the equations, and loosely coupled MIMD organizations are hard to use because a good partition of the problem to share workload and minimize communication is hard to find. Scheduling becomes relatively easier as the number of processes increases [3], and is quite simple when one has one process per instruction as in a data flow architecture [4].

A second type of application for which HEP seems to be well suited is the solution of partial differential equations for which the adjacencies of the discrete objects in the model change rapidly. Free surface and particle electrodynamics problems have this characteristic. The difficulty here is one of constantly having to rearrange the model within the computer to suit the connectivity implied by the architecture. Tightly coupled MIMD architectures have little implied connectivity. Associative SIMD architectures of the right kind may perform well on these problems, however.

Conclusion

The HEP system described above represents a compromise between the very tightly coupled data flow architectures and more loosely coupled multicomputer systems [2]. As a result, it has some of the advantages of each approach: It is relatively easy to implement parallel algorithms because any memory location can be used to synchronize two processes, and yet it is relatively inexpensive to implement large quantities of memory. In addition, the protection facilities make it possible to utilize the machine either as a multiprogrammed computer or as an MIMD computer.

References

[1] Flynn,M.J. "Some Computer Organizations and Their Effectiveness", IEEE-C21 (Sept. 1972).

[2] Jordan,H.F. "A Special Purpose Architecture for Finite Element Analysis", International Conference on Parallel Processing (1978).

[3] Lord,R.E. "Scheduling Recurrence Equations for Parallel Computation", Ph.D. Thesis, Dept. of Computer Science, Wash. State Univ. (1976).

[4] Rumbaugh,J. "A Data Flow Multiprocessor", IEEE-C26, p. 138 (Feb. 1977).

[5] Thornton,J.E. "Parallel Operation in the Control Data 6600", Proc. FJCC vol 26, part 2, p. 33 (1964).