The HEP Parallel Processor

by James W. Moore

Although there is an abundance of concepts for parallel computing, there is a dearth of experimental data delineating their strengths and weaknesses. Consequently, for the past three years personnel in the Laboratory's Computing Division have been conducting experiments on a few parallel computing systems. The data thus far are uniformly positive in supporting the idea that parallel processing may yield substantial increments in computing power. However, the amount of data that we have been able to collect is small because the experiments had to be conducted at sites away from the Laboratory, often in "software-poor" environments.

We recently leased a Heterogeneous Element Processor (HEP) manufactured by Denelcor, Inc. of Denver, Colorado. This machine (first developed for the Army Ballistic Research Laboratories at Aberdeen) is a parallel processor suitable for general-purpose applications. We and others throughout the country will use the HEP to explore and evaluate parallel-processing techniques for applications representative of future supercomputing requirements. Only the beginning steps have been taken, and many difficulties remain to be resolved as we move from experiments that use one or two of this machine's processors to those that use many. But what are the principles of the HEP?

Parallel processing can be used separately or concurrently on two types of information: instructions and data. Much of the early parallel processing concentrated on multiple-data streams. However, computer systems such as the HEP can handle both multiple-instruction streams and multiple-data streams. These are called MIMD machines.

The HEP achieves MIMD with a system of hardware and software that is one of the most innovative architectures since the advent of electronic computing. In addition, it is remarkably easy to use with the FORTRAN language. In its maximum configuration it will be capable of executing up to 160 million instructions per second.

The Architecture

Figure 1 indicates the general architecture of the HEP. The machine consists of a number of process execution modules (PEMs), each with its own data memory bank. connected to the HEP switch. In addition, there are other processors connected to the switch, such as the operating system processor and the disk processor. Each PEM can access its own data memory bank directly, but access to most of the memory is through the switch.

In a MIMD architecture, entire programs or, more likely, pieces of programs, called *processes*, execute in parallel, that is, concurrently. Although each process has its own independent instruction stream operating on its own data stream, processes cooperate by sharing data and solving parts of the same problem in parallel, Thus. throughput can be increased by a factor of N, where N is the average number of operations executed concurrently.

The HEP implements MIMD with up to sixteen PEMs, each PEM capable of executing up to sixty-four processes concurrently. It should be noted, however, that these upper limits may not be the most efficient configuration for a given, or even for most, applications. Any number of PEMs can cooperate on a job, or each PEM may be running several unrelated jobs. All of the instruction streams and their associated data streams are held in main memory while the associated processes are active.

How is parallel processing handled within an individual PEM? This is done by "pipelining" instructions so that several are in different phases of execution at any one moment. A process is selected for execution each machine cycle, a single instruction for that process is started, and the process is made unavailable for further execution until that instruction is complete. Because most instructions require eight cycles to complete, at least eight processes must be executed concurrently in order to use a PEM fully. However, memory access instructions require substantially more than eight cycles. Thus, in practice. about twelve concurrent processes are needed for full utilization, and a single HEP PEM can be considered a "virtual" 8- to 12-processor machine. If a given application is formulated and executed using p processes (where p is an integer from 1 to 12), then execution time for the application will be inversely proportional top.

Now what happens when individual PEMs are linked together? Each PEM has its own program memory to prevent conflicts in accessing instructions, and all PEMs are connected to the large number of other data memory banks through the HEP switch (a high-speed, packet-switched network). One result of these connections is that the number of switch nodes increases more rapidly than the number of PEMs. As one changes from a 1-PEM system toward the maximum 16-PEM configuration, the transmittal time through the HEP switch, called latency,

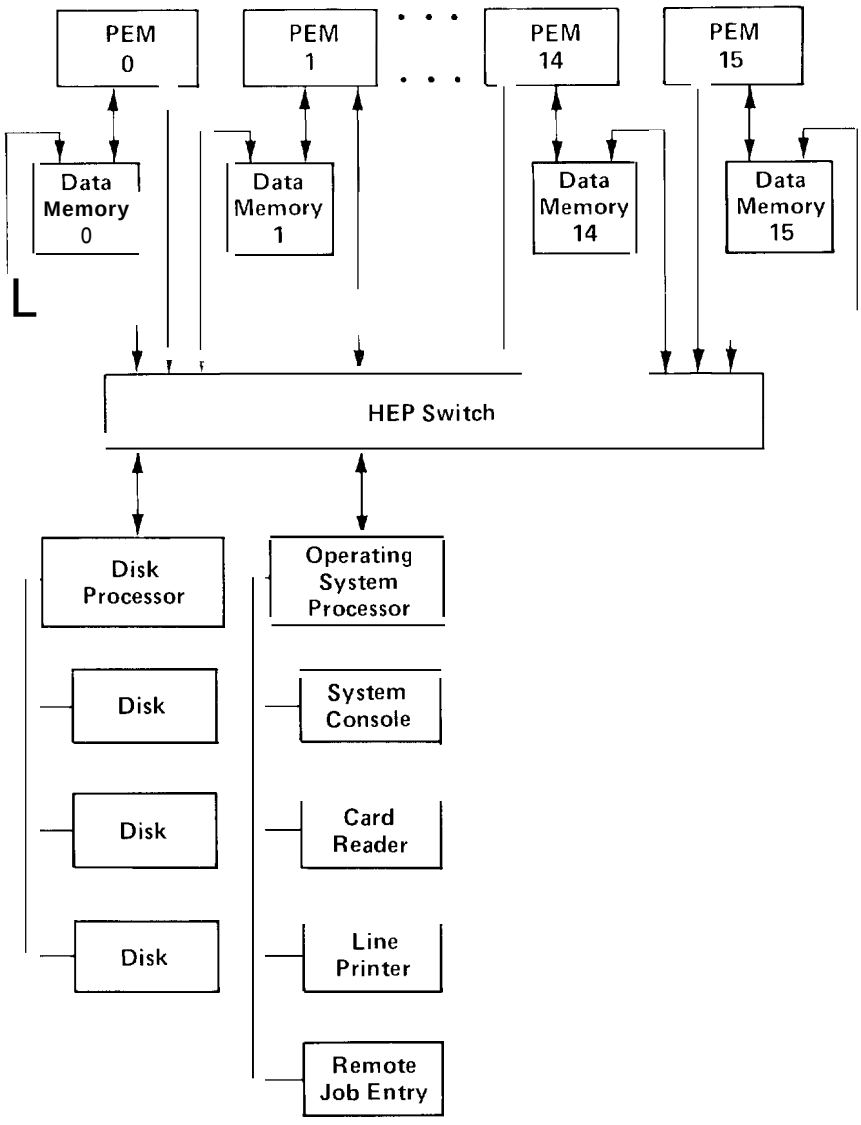
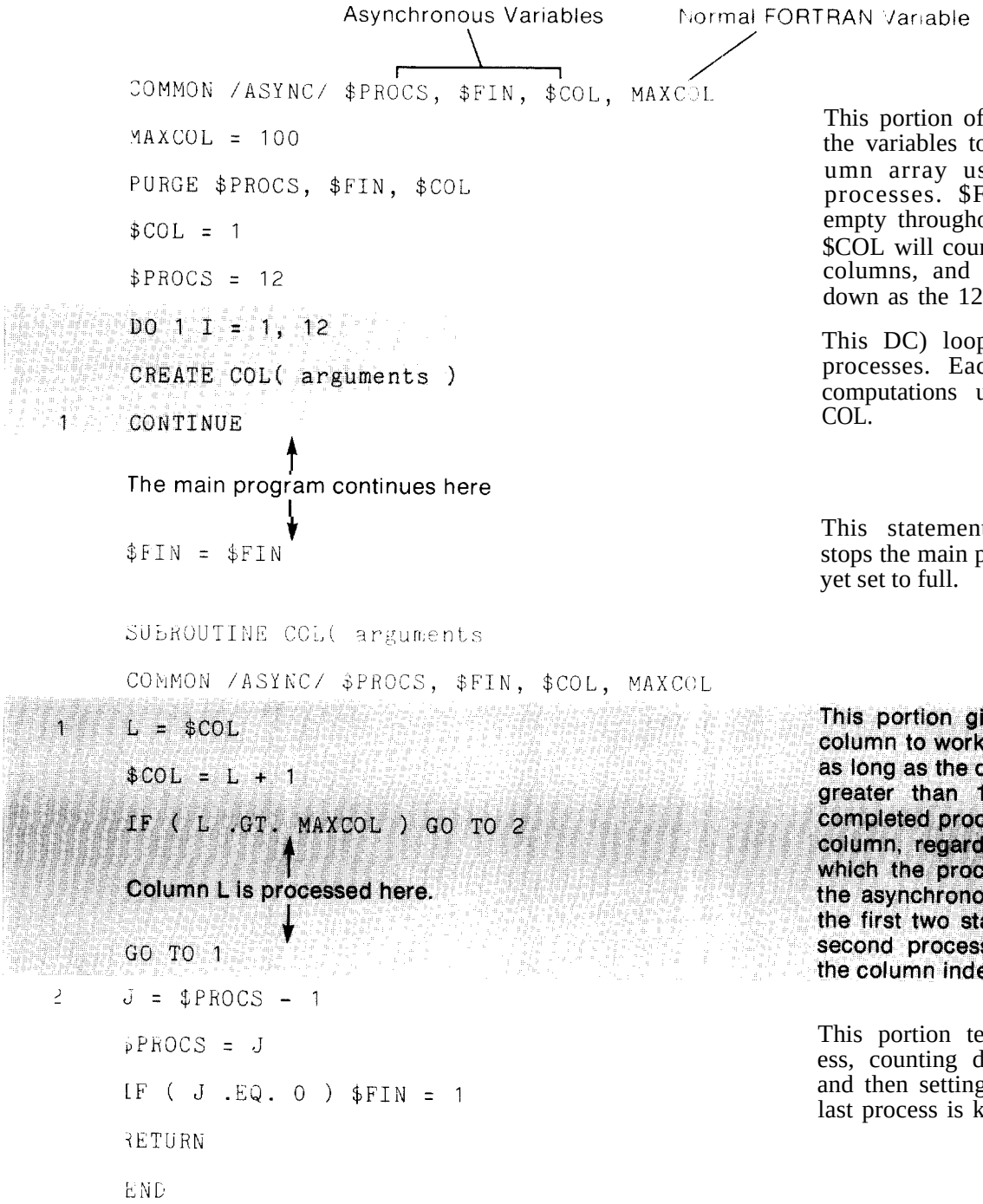


Fig. 1. The HEP parallel processor.



This portion of the program sets up the variables to process an 100-column array using 12 concurrent processes. $FIN remains set at empty throughout the computations, $COL will count up through the 100 columns, and $PROCS will count down as the 12 processes die off.

This DC) loop CREATES the 12 processes. Each process does its computations using SUBROUTINE COL.

This statement, otherwise trivial, stops the main program if $FIN is not yet set to full.

This portion gives each process a column to work on in the array and, as long as the column index L is not greater than 100, will send each completed process back for another column, regardless of the order in which the processes finish. Use of the asynchronous variable $COL in the first two statements prevents a second process from starting until the column index has been reset.

This portion terminates each process, counting down with $PROCS, and then setting $FIN to full as the last process is killed.

Fig. 2. An example of HEP FORTRAN,

quickly becomes substantial. Such latency increases the number of processes that must be running in each PEM to achieve full utilization. Although there is not enough data yet to provide good estimates on how fast latency actually increases with the number of PEMs, experience with a 2-PEM system suggests that a 4-PEM system will require about twenty concurrent processes in each PEM.

Process Synchronization

A critical issue in MIMD machines is the synchronization of processes. The HEP solves this problem in a simple and elegant manner, Each 64-bit data word has an extra bit that is set to full each time a datum is stored and is cleared to empty each time a datum is fetched, In addition, two sets of memory instructions are employed. One set is used normally throughout most of the program. This set ignores the extra bit and will fetch or store a data word regardless of whether the bit is full or empty. Data may then be used as often as required in a process.

The second set of instructions is defined through the use of asynchronous variables, Typically, this set is used only at the start or finish of a process, acting as a barrier against interference from other processes, The set will not fetch from an empty word or store into a full word. Thus, to synchronize several processes an asynchronous variable is defined that can only be accessed, using the second set of instructions, at the appropriate time by each process. A process that needs to fetch an asynchronous variable will not do so if the extra bit is empty and will not proceed until another process stores into the variable, setting it full. Because the full and empty properties of this extra bit are implemented in the HEP hardware at the user level. requiring no operating system intervention. the usual synchronization methods (semaphores, etc.) can be used, and process synchronization is very efficient.

FORTRAN Extensions to Support Parallelism

Only two extensions to standard FORTRAN arc required to exploit the parallelism inherent in the HEP: process creation and asynchronous variables. Standard FORTRAN can. in fact, handle both, but the current HEP FORTRAN has extensions specifically tailored to do so.

These extensions allow the programmer to create processes in parallel as needed and then let them disappear once they are no longer needed. Also the number of PEMs being used will vary with the number of processes that are created at any given moment.

Process creation syntax is almost identical to that for calling subroutines: CALL is replaced by CREATE, However, in a normal program, once a subroutine is CALLed, the main program stops; in HEP FORTRAN. the main program may continue while a CREATEd process is being worked on. If the main program has no other work, it may CALL a subroutine and put itself on equal footing with the other processes until it exits the subroutine. A process is eliminated when it reaches the normal RETURN statement.

We can illustrate these techniques by showing how the HEP is used to process an array in which each column needs to be processed in the same manner but independently of the other columns (Fig. 2). First, one defines a subroutine that can process a single column in a sequential fashion. We could usc this subroutine by creating a process for each Column and then scheduling all the processes in parallel, but there is a limit on the number of processes that each PEM can handle. A better technique would be to CREATE eight to twelve processes per PEM and let the processes *self-schedule.* Each process selects a column from the array, does the computation for that column, then looks for additional columns to work on. Several asynchronous variables are the key to this technique. Each process that is not computing checks the first of' these variables both to see if it can start a computation and, if so, which column is next in line. At the end of that computation and regardless of what stage any other process has reached. the process checks again to see if there are further columns to be dealt with. If not, the process is terminated. A second asynchronous variable counts down as the processes die off. When the last operating process completes its computation. a number is stored in a third, previously empty asynchronous variable, setting its extra bit to full. This altered variable is a signal to the main program that it may use the recently generated data. This method tends to smooth irregularities in process execution time arising from disparities in the amount of processing done on the individual columns and, further, does not require changes if the dimension of the array is changed.

More elegant syntactic constructs can be devised. but the HEP extensions are workable. For well-structured code, conversion to the HEP is quite easy, For more complex programs the main difficulty is verifying that the parallel processes defined are in fact independent. No tools currently exist to help with this verification.

Early Experience with the HEP

Several relatively small FORTRAN codes have been used on the HEP at Los Alamos. One such code is SIMPLE, a 2000-line, two-dimensional, Lagrangian, hydro-diffusion code. By partitioning this code into processes, about 99 per cent of it can be executed in parallel. The speedup on a 1-PEM system is close to linear for up to eleven processes and then flattens out, indicating that the PEM is fully utilized, To achieve this degree of speedup on any MIMD machine requires a very high percentage of parallel code. How difficult it will be to achieve a high percentage of parallelism on full-scale production codes is an open question, but the potential payoff is significant. ■

U.S. GOVERNMENT PRINTING OFFICE 1983 — 776-101/1007