Architecture and applications of the HEP multiprocessor computer system

Burton J. Smith
Denelcor, Inc., 14221 E. 4th Avenue, Aurora, Colorado 80011

Abstract

The HEP computer system is a large scale scientific parallel computer employing shared-resource MIMD architecture. The hardware and software facilities provided by the system are described, and techniques found to be useful in programming the system are also discussed.

Introduction

The HEP computer system[1] is a large scale scientific parallel computer employing shared-resource MIMD architecture[2]. In this particular implementation, the processors are pipelined to support many concurrent processes, with each pipeline segment responsible for a different phase of instruction interpretation. Each processor has its own program memory, general purpose registers, and functional units; a number of these processors are connected to shared data memory modules by means of a very high speed pipelined packet switching network. The extensive use of pipelining in conjunction with the shared resource idea is synergistic in several useful and important ways, and results in a very flexible and effective architecture. For example, the switch used to interconnect processors and memories is modular, and is designed to allow a given system to be field-expanded. The increased memory access times that result from greater physical distances can be compensated for by using more processes in each processor because the switch is pipelined.

An overall block diagram of a typical HEP configuration is shown in Figure 1. The switch network shown has 28 nodes; it interconnects four processors, four data memory modules, an I/O cache module, an I/O control processor, and four other I/O devices. Systems of this kind can be built to include as many as 16 processors, 128 data memory modules, and 4 I/O cache modules. Each processor performs 10 million instructions per second (MIPS), and the switch bandwidth is 10 million 64 bit words per second in every network link. Data memory module bandwidth is 10 million 64 bit words per second, and each I/O cache supports sequential or random access I/O at sustained rates of 32 million bytes per second.

The remainder of this paper discusses the hardware and software architecture of the system. An overview will be given of each of the major components of the system, followed by a programmer's view of the facilities provided by these components.

Processor and data memory

A simplified diagram of the HEP processor internal organization is shown in Figure 2. The process status word (PSW) contains the program counter and other state information for a HEP process; these PSW's circulate in a control loop which includes a queue, an incrementer, and a pipelined delay. The delay is such that a particular PSW cannot circulate around the control loop any faster than data can circulate around the data loop consisting of register memory and the function units. As the program counter in a circulating PSW increments to point to successive instructions in program memory, the function units are able to complete each instruction in time to allow the next instruction for that PSW to be influenced by its effects. The control and data loops are pipelined in eight 100 nanosecond segments, so that as long as at least eight PSW's are in the control loop the processor executes 10 million instructions per second. A particular process cannot execute faster that 1.25 million instructions per second, and will execute at a lesser rate if more than eight PSW's are in the control loop.

The PSW contains a 20 bit program counter to allow for program memory configurations ranging from 32K to 1024K words. Each instruction in program memory is 64 bits long, and typically consists of an opcode and three register memory addresses. These addresses can be modified by the addition of an index value from the PSW to allow reentrant programming. The register memory consists of 2048 general purpose 64 bit registers, augmented by a 4096 location constant memory. Constant memory may only be modified by supervisor processes.

One of the function units, the scheduler function unit (SFU), is responsible for implementing load and store instructions to transmit data between register memory and data memory. When such an instruction is executed, the SFU sends a switch message packet containing a 32 bit data memory address, a return address identifying both processor and process, and 64 bits of data if a store instruction was executed. The SFU also removes the process that executed the instruction from the control loop, and does not reinsert it until a response packet is received from the switch. When that response packet arrives, the SFU writes the data portion of the response in the appropriate register if a load instruction was executed. In order to perform these functions, the SFU is equipped with a queue similar to the queue in the control loop of the processor proper, and a process migrates freely between these two queues as it initiates and completes data memory reference instructions.

The various function units of the HEP processor support the data types shown in Figure 3. The floating point formats are sign-magnitude and use a seven bit, excess 64 hexadecimal exponent. Integer formats are twos complement. The various precisions for each data type are implemented by loading and storing partial words in data memory using either the leftmost or rightmost part of the register. In addition, load instructions can specify sign extension instead of zero extension for right justified partial word load instructions.

The floating point operations implemented by the processor are floating point add, subtract, multiply, divide, and floating point compare instructions that optionally produce integer 1, floating point 1.0, or a 64 bit vector of all 1's as "true" values and zero as "false" values. Unnormalized floating point add and subtract are also implemented, as are conversion instructions between floating point and integer (the Fortran functions FLOAT, INT, and AINT). Integer functions are add, subtract, multiply, arithmetic shift, and compare instructions analogous to those for floating point. Both halves of the 128 bit twos complement product of two integers are available. Bit vector instructions include all sixteen Boolean functions, logical and circular shifts, instructions which return the numeric bit position of the leftmost "1" or "0" in a register, and instructions which set or reset a bit at a given numeric position in a register.

The control instructions available provide not only for the loading, storing and modification of the executing PSW to implement conditional branches and subroutine calls, but also for the conditional creation and termination of processes. These latter functions are performed by ordinary (unprivileged) instructions, and allow the user to control the amount of concurrency with very low overhead. A supervisor call instruction allows user processes to create supervisor processes, which in turn may execute privileged instructions to manage the user processes and perform I/O.

Cooperating parallel processes must have some way of synchronizing with each other to allow data sharing. In HEP, this facility is provided by associating an access state with each register memory and data memory location. In data memory, the access states are "full" and "empty"; a load instruction can be made to wait until the addressed location is "full" and indivisibly (i.e., without allowing an intervening reference to the location) set the location "empty". Similarly, a store instruction can wait for "empty" and then set "full" at any location in data memory. In register memory, an instruction can require that both sources be "full" and the destination "empty", and then set both sources "empty" and the destination "full". To ensure the indivisibility of this kind of operation, a third access state, "reserved", is set in the destination register location when the source data are sent to the function units, and only when the function unit stores the result is the destination set "full". No instruction can successfully execute if any of the registers it uses is "reserved".

A process failing to execute an instruction because of improper register access state is merely reinserted in the queue with an unincremented program counter so that it will reattempt the instruction on its next turn for execution. A process executing a load or store instruction that fails because of improper data memory access state is reinserted in the SFU queue and generates a new switch message on its next attempt.

One simple way to exploit the parallelism available in HEP is to run several independent programs simultaneously. To protect independent programs from each other, base and limit registers in program memory, register memory, and data memory are associated with processes. A set of processes having the same protection domain (the same base and limit register values) is called a task. HEP support up to seven user tasks and seven corresponding supervisor tasks in each processor. When a user process executes a create instruction, the new process runs in the same task as the originating process; privileged instructions exist to allow supervisors to create processes in any task or to kill all of the processes in a task.

A HEP job consists of one or more tasks, normally intended to execute on a like number of processors. The tasks making up a job have disjoint allocations in program and register memory, but an identical allocation in data memory to allow them to share data and synchronize. Supervisor calls are used by a process in one task to create a process in a different task of the same job. When a job is submitted to the system, the user specifies the maximum number of processes that will ever be active in each task; the operating system only loads the job when it can guarantee that there are enough processes available in each processor. The maximum number of user processes supported by a processor is 50, and these may be distributed in any way whatsoever among the seven (or fewer) user tasks.

Switch

The HEP switch is a synchronous, modular packet switching network consisting of an arbitrary number of nodes. Each node is connected to its neighbors (which may be processors, data memory modules, or other nodes) by three full-duplex ports. Each node receives three message packets on each of its three ports every 100 nanoseconds, and attempts to route the messages in such a way that the distance from each message to its addressed destination is reduced. To accomplish this, each node has three routing tables, one per port, which are initialized when the system is initialized. The tables are indexed by the destination address and contain the identification of the preferred port out of which the packet should be sent.

When a conflict for a port occurs, the node does not enqueue messages; instead, it routes all messages immediately to output ports. It is the responsibility of the neighbors of the node to make sure that incorrectly routed messages eventually reach their correct destinations. To help accomplish this, each message contains a priority which is initially 1 and is incremented whenever a message is routed incorrectly. In a conflict between messages of differing priority, the message with the highest priority is routed correctly. As a consequence, devices connected to the switch must immediately reinsert arriving messages not addressed to them in preference to inserting new messages.

Empty messages are just those of priority zero; zero priority messages do not increase in priority and always lose conflicts. Also, since the ports of the switch nodes and the devices connected to it are full duplex, an Eulerian circuit of the switch is guaranteed to exist. Such an Eulerian circuit traverses every port exactly once in each direction. Packets with the maximum priority of 15 are sent on such an Eulerian circuit, independent of destination address, to ensure that no conflicts between two of these messages occur. When a priority 15 message eventually reaches its addressed destination, it is recognized and removed from the switch in the normal manner. Like the routing table data, the Eulerian circuit information is loaded into each switch node when the system is initialized.

Each switch node checks the parity of the incoming messages, and also checks that the routing it performs is a permutation of the ports. If a check fails, the switch node signals the diagnostic and maintenance subsystem of the HEP that an error has occurred. The failing port or node, or a failed processor or memory for that matter, can be removed from the system just by reprogramming the routing tables to reflect the reduced configuration. To avoid splitting the system in half, the graph of the switch must be at least biconnected; that is, there must exist two disjoint paths from any node to any other. If the port connecting a processor or a memory to the switch fails in either direction, the effect is the same as if the processor or memory itself fails.

The propagation delay for a message in the switch is 50 nanoseconds for each port traversed. The pipeline rate is one message per 100 nanoseconds per port. To ensure that message routing conflicts are synchronized, the switch must be two-colorable, so that messages conflict at nodes of one color on even multiples of 50 nanoseconds and at the other color on odd multiples. Excepting this constraint, the graph of the switch is totally arbitrary. Adjacent nodes may be separated by up to four meters of connecting wire to allow a great deal of flexibility in system configuration. A HEP system may be field-expanded by adding processors, memories, and switch nodes up to the maximum allowed.

I/O facilities

The HEP High Speed I/O Subsystem (HSIOS) includes from one to four I/O cache modules, each of which serves as a buffer between the switch and 32 I/O channels. Each I/O channel can support transfer rates up to 2.5 million bytes per second. The I/O cache supports this transfer rate by all channels simultaneously, and can concurrently transfer 80 million bytes per second via its switch port for a total bandwidth of 160 million bytes per second. An I/O cache can range in size from 8 million to 128 million bytes in 8 million byte increments, so that the maximum amount of I/O cache memory implementable in a single HEP system is 512 million bytes. The reason for this large amount of memory is to allow a large page size to be used in conjunction with a large number of disks or tapes. The large page size (40 kilobytes nominal) means that mechanical delays can be made

insignificant compared to data transfer times to increase channel utilization, and the large number of channels allows data to be distributed across many disks to provide high I/O bandwidth through the use of parallelism. The distribution of data is provided for by interleaving logical records among many files, one file per disk, and is accomplished not by the file system itself but by the library I/O routines that the user calls. In this way, the management of distributed data is entirely under the user's control, allowing him to exchange reliability for speed as required by his application.

The I/O channels are controlled by an I/O control processor (IOCP). The IOCP is interrupted both by the channels, when I/O operations are completed, and by arrivals of switch messages from supervisor processes running in the processors. The function of the IOCP is to schedule the I/O requests from the supervisors on the channels. To a supervisor, the IOCP-switch interface appears as a sequence of special memory locations in the data memory address space, and an I/O request is made by storing the request at one of these special locations. When the switch message arrives at the IOCP-switch interface, the IOCP services the interrupt by acquiring the message from the interface and enqueueing it internally on a queue served by the I/O channel that was requested. When the request reaches the head of the queue, it is serviced; on completion, a response message is loaded into the switch interface and sent back to the processor from which the request came. The effect seen by the supervisor process itself is extremely simple; a page I/O request was made and completed by executing a store instruction. The fact that the store instruction may have taken several milliseconds to execute has no effect on the performance of the processor if enough processes remain in the control loop.

When a file is opened, the number of cache frames allocated to the file may be specified as a parameter (the default quantity is 2). Another parameter specifies the sequential direction (forward or backward). All data transfer instructions perform sequential I/O in the specified direction, and the supervisor handling the requests attempts to "read ahead" and "write behind" the pages surrounding the current file position. To accomplish random I/O, a separate command is implemented to allow the current file position to be changed. This feature allows caching of data to proceed concurrently with user processing.

A HEP file consists of a header page and zero or more data pages. The header is kept in the I/O cache as long as the file is open, and contains either the data itself or pairs of pointers to the disk locations of the data pages and the cache frames holding those pages, if any. Since all pages are the same size, both disk and cache space allocation are performed using bit tables and are extremely fast; this is an important consideration when a large number of supervisor processes are all attempting to allocate space in parallel. To the user, a file appears to be a randomly addressable sequence of records or words, and there are no "access methods" provided by the file system itself.

When a user process executes a supervisor call to perform I/O, the supervisor process first computes the page or pages containing the data from the current file position and from the amount of data to be transferred. It then verifies that those pages are in the cache, requesting them, from the IOCP as necessary. Next, it transfers the data between the cache and the user's buffer, and changes the current file position based on the sequential direction (forward or backward). Finally, the supervisor schedules page "write behind" and "read ahead" with the IOCP as required, and returns to the user without waiting for the IOCP. If a page was never modified, it is not written on the disk by "write behind". Moreover, a reference count is maintained for each cached page and for the file as a whole to allow a file to be multiply opened by a large number of processes. A page is actually uncached only when its reference count is zero, and the file header itself is uncached when the file reference count is zero.

Fortran extensions

Two kinds of extensions were added to HEP Fortran to allow the programmer to write explicit parallel algorithms. The first class of extensions allows parallel process creation. In HEP Fortran, a CREATE statement, syntactically similar to a CALL, causes a subroutine to run in parallel with its creator. The RESUME statement, syntactically like RETURN, causes the caller of a subroutine to resume execution in parallel with the subroutine. If a subroutine was CREATEd, a RESUME has no effect, and a RETURN causes the termination of the process executing the subroutine if the subroutine was CREATEd or if it previously executed a RESUME.

The second extension allows the programmer to use the access states provided by the HEP hardware. Any variable whose name begins with the character "$" is called an asynchronous variable, and has the property that an evaluation of the variable waits until the associated location is "full" and sets it "empty" while fetching its value. An assignment to the variable waits until the variable is "empty" and then sets it "full" while storing the new value. A PURGE statement is used to unconditionally set the access state to "empty".

HEP Fortran generates fully reentrant code, and dynamically allocates registers and local variables in data memory as required by the program. For example, it is often useful to place a CREATE statement in a loop so that several parallel processes will execute identical programs on different local data. An example is shown in Figure 4; in this instance, the program creates NPROCS-1 processes all executing subroutine S, and then itself executes the subroutine S by calling it, with the result that NPROCS processes are ultimately executing S. The parameter $IP is used here to identify each process uniquely. Since parameter addresses rather that values are passed, $IP is asynchronous and is filled by the creating program and emptied within S. This prevents the creating program from changing the value of $IP until S has made a copy of it. The asynchronous variable $NP is used to record the number of processes executing S. When S is finished, $NP is decremented, and when the creating program discovers that $NP has reached zero, all NPROCS processes have completed execution of S (excepting possibly the RETURN statement).

HEP programming

The facilities provided by the HEP hardware and software are clearly well suited to pipelining as a means of parallel algorithm implementation. The access states "full" and "empty" can be viewed as a mechanism for passing messages between processes using single-word queues. It is also clear that a location can be used to implement a critical section merely by requiring processes to empty the location upon entry and fill it upon exit. Other common synchronization mechanisms can be implemented using similar techiques. Processes can be dynamically initiated and terminated to avoid busy waiting.

One frequently occurring situation in parallel programming involves the computation of recurrences of the form

$$X = X \text{ op } Y$$

where op is commutative and associative and Y does not depend on X. One example occurs in the use of $NP in Figure 4; another obvious example is vector inner product. The difficulties with merely writing

$$\$X = \$X \text{ op } Y$$

are twofold. First, all processes are competing for access to $X and interfere with each other; second, the final value of $X often must be made available to the various processes in some independent way, perhaps involving another recurrence similar to the recurrence for $NP in Figure 4.

One attractive method which avoids these difficulties requires log P locations for each of the P processes. The method simultaneously computes the final value of X and broadcasts it to the P processes in time O(log P). The idea is to implement an appropriate interconnection network (in software) that has the required computational property at each element. In this case, the property required is that the two identical element outputs be the result of applying op to the two inputs. One HEP Fortan program to accomplish this task is shown in Figure 5 with op = + and with a Staran flip network[3] interconnection.

Notice that since the function used to compute K from I is always a bijection, there is no conflict for the elements of $A by the processes. That is, only one process is attempting to fill a given location in $A and only one process is attempting to empty it. Overtaking by processes is not possible because of this fact. In addition, the same array $A can be reused immediately to perform a different function (e.g., global minimum) as long as the same network topology is used. Analogous techniques, sometimes requiring other network topologies, can be used to permute the elements of X, to sort X, to perform fast Fourier transforms, and so forth.

Another important HEP programming technique allows processes to schedule themselves. In the simplest case, a number of totally independent computational steps is to be performed that significantly exceeds the number of processes available; moreover, the execution time of the steps may be widely varying. The self-scheduling idea is to allow each process to acquire the next computational step dynamically when it finishes the previous one. In a more complex situation involving dependencies and priorities, the processes might perform a significant amount of computation in scheduling themselves, but in most cases the method is quite efficient. Figure 6 shows an example of an ordinary DO loop in which all iterations are presumed independent; Figure 7 is a parallel version of Figure 6 which uses self-scheduling. One of the most attractive benefits of self-scheduling a loop is that there is no need to worry about poor process utilization resulting from an unsatisfactory a priori schedule.

Conclusions

The HEP computer system represents a unique and very flexible architecture. First, its modularity is exceptional even for an MIMD computer. Second, the availability of a very natural synchronization primitive at every memory location allows the programmer a large amount of freedom in developing parallel algorithms. Finally, a mechanism is provided to allow the user to control the number of processes dynamically in order to take advantage of varying amounts of parallelism in a problem.

References

1. Smith, B.J., "A Pipelined, Shared Resource MIMD Computer," Proceedings of the 1978 International Conference on Parallel Processing, pp. 6-8.
2. Flynn, M.J., "Some Computer Organizations and their Effectiveness, IEEE Transactions on Computers, Vol. 21, pp. 948-960 (Sept. 1972).
3. Batcher, K.E., "The Flip Network in Staran," Proceedings of the 1976 International Conference on Parallel Processing, pp. 65-71.

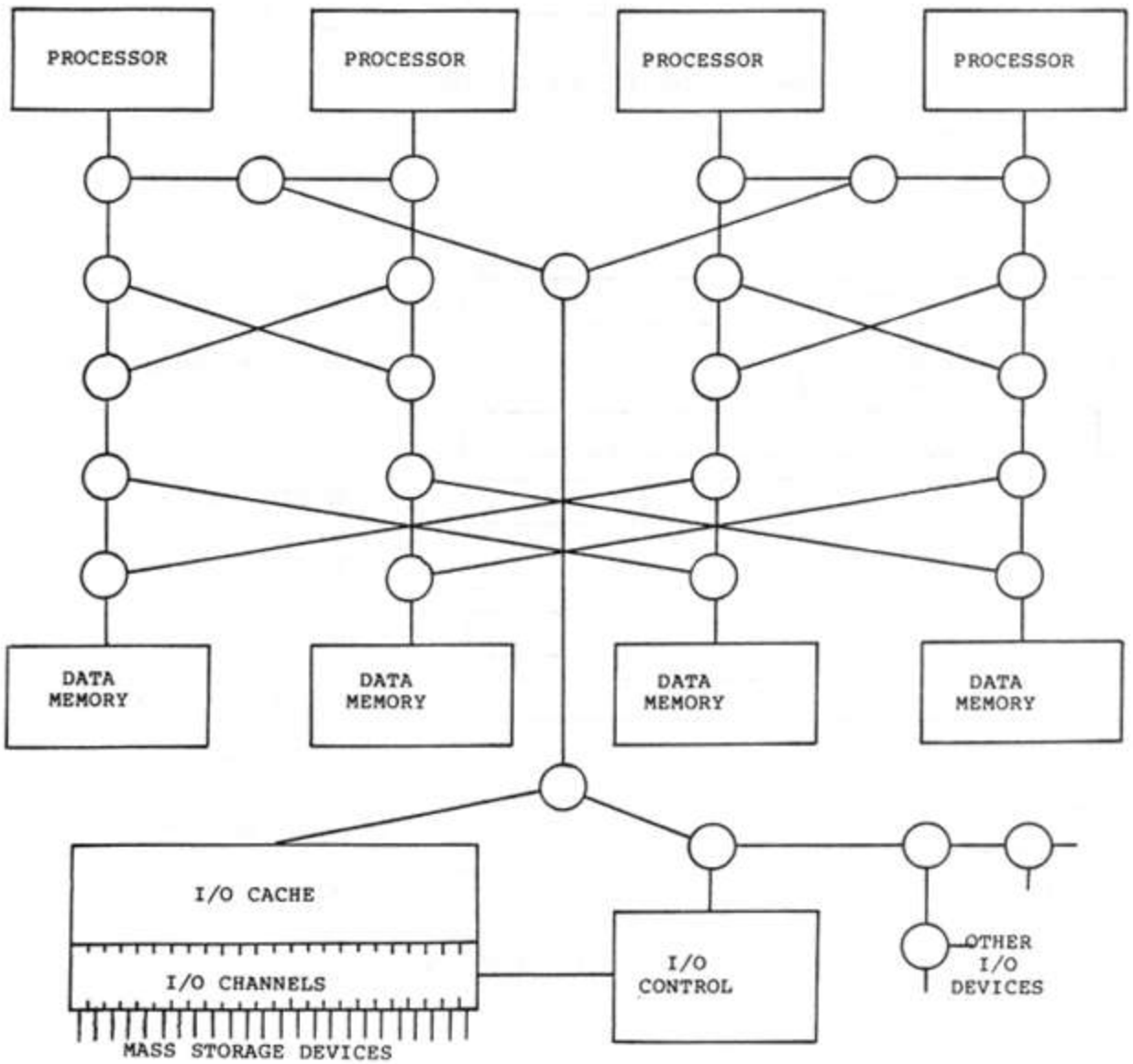

Figure 1. A typical HEP system

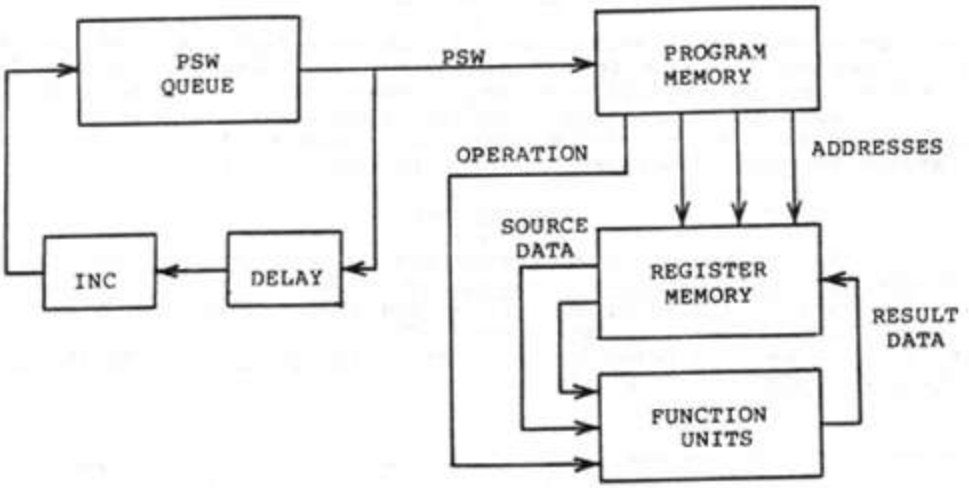

Figure 2. Simplified HEP processor

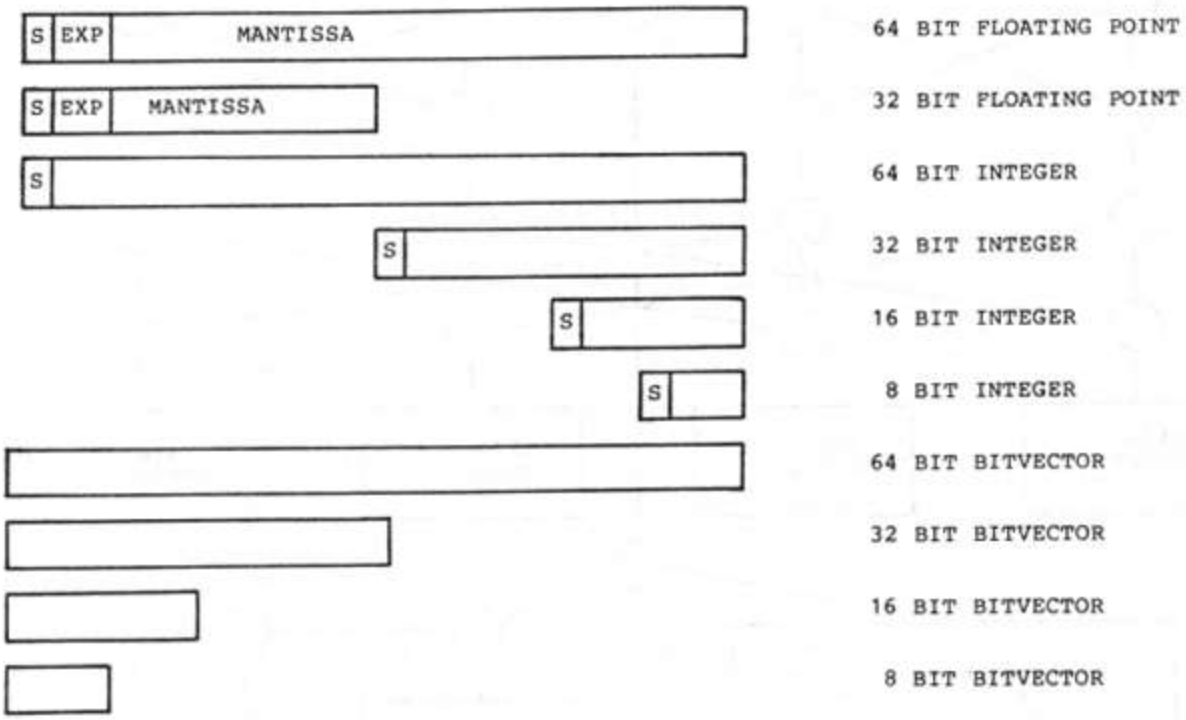

Figure 3. Data types

```
      :
      :
      PURGE $IP, $NP
      $NP = NPROCS
      DO 10 I = 2, NPROCS
      $IP = I-1
      CREATE S($IP,$NP)
   10 CONTINUE
      $IP = NPROCS
      CALL S($IP,$NP)
C     WAIT FOR ALL PROCESSES TO FINISH
   20 N = $NP
      $NP = N
      IF (N .NE. 0) GO TO 20
      :
      :
      SUBROUTINE S($IP,$NP)
      MYNUM = $IP
      :
      :
      $NP = $NP-1
      RETURN
      END
      :
      :
```

Figure 4. HEP Fortran example

```
C     REPLACE EACH ELEMENT OF THE VECTOR X
C     BY THE SUM OF THE ELEMENTS OF X
C     USING THE INITIALLY EMPTY ARRAY $A.
C     P = 2**L PROCESSES EXECUTE THIS PROGRAM.
C     THE PROCESS IDENTIFER IS I.
      DIMENSION X(P), A(P,L)
      JPOW = 2
C     JPOW IS 2 TO THE J POWER
      DO 10 J = 1,L
C     COMPUTE THE PROCESS K = (I-1) EXOR(JPOW/2)+1
C     WITH WHICH THIS PROCESS WILL EXCHANGE DATA
      K = ((I-1)/JPOW)*JPOW + MOD (I-1 + JPOW/2,JPOW)+1
      JPOW = JPOW*2
C     NOW EXCHANGE DATA AND ACCUMULATE
      $A(K,J) = X(I)
      X(I) = X(I) + $A(I,J)
   10 CONTINUE
```

Figure 5. Summation by network simulation

```
      DO 10 I = J, K, L
      :
      :
   10 CONTINUE
```

Figure 6. A DO loop

```
      PURGE $IV
      $IV = J
C     CREATE ANY NUMBER OF PROCESSES EXECUTING
    1 I = $IV
C     THIS PROCESS HAS SEIZED AN ITERATION INDEX
C     LET ANOTHER PROCESS OBTAIN THE NEXT INDEX
      $IV = I + L
C     TERMINATE IF THROUGH
      IF (I .GT. K) RETURN
      :
      :
      GO TO 1
```

Figure 7. A self-scheduled loop