NASA-CR-3870 19850010313

NASA Contractor Report 3870

System Software for the Finite Element Machine

Thomas W. Crockett and Judson D. Knott

CONTRACT NAS1-16000
FEBRUARY 1985

NASA

3 1176 01309 4439

NASA Contractor Report 3870

System Software for the Finite Element Machine

Thomas W. Crockett and Judson D. Knott
Kentron International, Inc.
Hampton, Virginia

Prepared for
Langley Research Center
under Contract NAS1-16000

National Aeronautics
and Space Administration

**Scientific and Technical
Information Branch**

1985

SUMMARY

The Finite Element Machine is an experimental MIMD computer built by NASA to investigate the application of parallel processing to structural engineering analysis. System software has been developed for this computer to provide a support environment for parallel application programs. The overall software design is described and several important issues are discussed in detail, including processor management, communication, synchronization, input/output, debugging, and performance analysis. Use of the system for applications development has revealed several areas which require more attention, and suggestions are offered for approaching these problems.

1. INTRODUCTION

The performance potential of conventional sequential computer architectures is bounded by technological and physical constraints. As these constraints become increasingly significant in the design of high performance computers, research into parallel architectures has intensified, with a multitude of designs being proposed. One important category of parallel architecture is the mesh-connected, multiple-instruction, multiple-data (MIMD) array, where many semi-autonomous processors are embedded within a communications matrix. Several machines of this type are being built or prototyped, including CHiP [22], the Japanese PAX computers [11,12], and NASA's Finite Element Machine.

In order to conveniently use these (or other) computers, one or more layers of system software are needed to provide an interface between the hardware and users' application programs. A number of special considerations must be kept in mind when designing system software for MIMD arrays, and standard system capabilities must be re-examined in light of the multiprocessor environment. Among these considerations are processor control and coordination, inter-processor communication, problem partitioning and mapping, data management and I/O, debugging, and performance analysis.

This paper describes system-level software developed for the Finite Element Machine, summarizes experience using the system, and suggests areas for further work. In Section 2, a brief overview of the motivation and purpose of the Finite Element Machine is given, and the hardware architecture is described at a functional level. Section 3 discusses the major components of the system software and how they interact, and Section 4 describes specific system capabilities and their underlying implementations. Section 5 relates experience in designing the system software, critiques the current hardware and software system, and offers suggestions for further work.

2. THE FINITE ELEMENT MACHINE

2.1 Overview

The Finite Element Machine (FEM) [14,18,24] is a research computer being built at the National Aeronautics and Space Administration's Langley Research Center as part of an activity which is exploring the use of parallel processing for structural engineering analysis. Major areas of investigation include parallel numerical algorithms, system and application software, and computer architecture. The machine derives its name from the finite element method (see, for example, [25]) which is a fundamental tool of modern engineering analysis and the original application for which the machine was designed. FEM serves as a testbed for algorithm development and provides a focus for addressing parallel processing issues in computer science. It also represents a first attempt at designing a parallel architecture specifically suited to structural engineering computations. Experience gained from the current machine is expected to clarify hardware and software requirements for new generation MIMD computers. For a synopsis of the FEM project through mid 1983, see [2].

2.2 Hardware Architecture

The Finite Element Machine consists of an array of asynchronous microcomputers (the *Array)* attached to a minicomputer front-end (the *Controller)*, as shown in Figure 1. Throughout this paper, the term *processor* is used to refer to the individual microcomputers in the Array. Three printed circuit boards, known as the CPU, IO-1, and IO-2 boards, comprise each processor. The CPU board is based on a Texas Instruments[1] TMS9900 16-bit microprocessor. An associated Advanced Micro Devices Am9512 arithmetic unit provides single and double precision floating-point capabilities. The CPU board also contains 16K bytes of erasable, programmable read-only memory (EPROM), 32K bytes of dynamic read/write memory (RAM), an interrupt interface, two timers, and the processor's internal clock. The only memory in the Array is the local EPROM and RAM on each CPU board -- there is no shared memory in the system.

[1]Use of trademarks or names of manufacturers in this report does not constitute an official endorsement of such products or manufacturers, either expressed or implied, by the National Aeronautics and Space Administration.

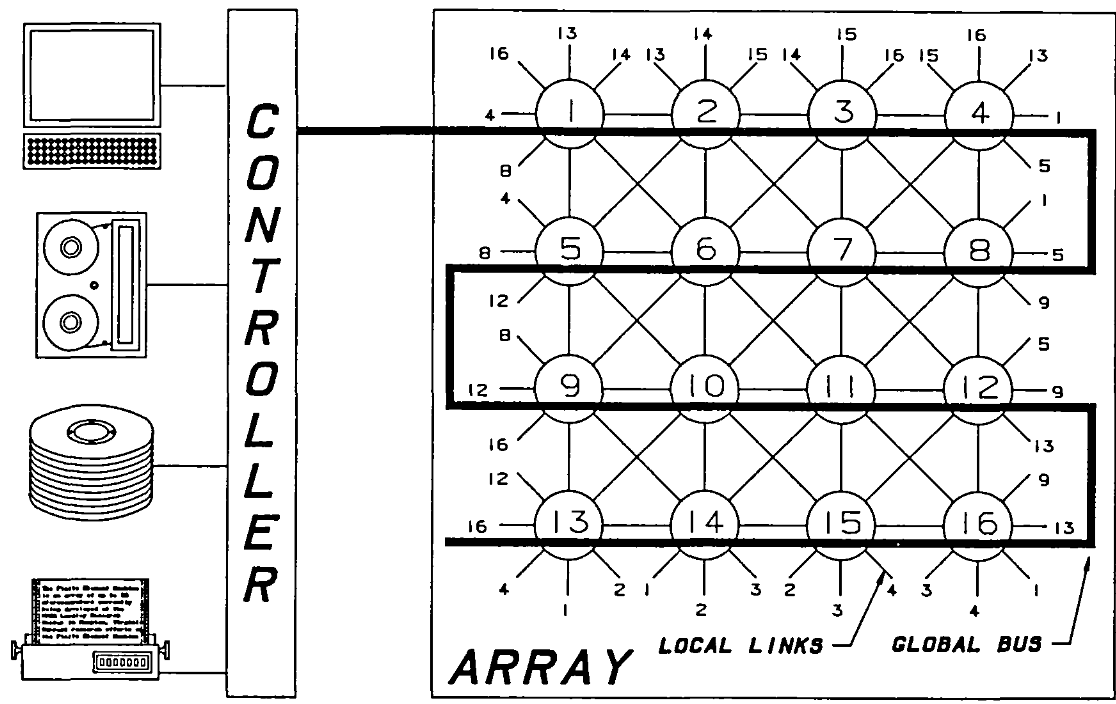

Figure 1. Block diagram of a 16-processor Finite Element Machine. The local links are configured in an eight-nearest- neighbor mesh with toroidal wraparound, leaving four links unused. The flag network and sum/max tree have been omitted for clarity.

The IO-1 and IO-2 boards contain additional circuitry which interfaces the CPU board to an extensive set of communication mechanisms, as described below:

1. The *global bus* is a 16-bit parallel time-multiplexed communication bus which provides point-to-point and broadcast communications between all of the processors in the Array. Each processor is uniquely identified by an identification number, termed the *self-id,* which serves as its global bus address. A bus transmission consists of a 16-bit data word, a 4-bit tag field, and a 10-bit source/destination address. One bit is used to determine the mode of transmission, either broadcast or point-to-point. Hardware FIFO buffers provide automatic queueing of up to 64 transmissions for both input and output operations.

2. Bit-serial bi-directional *local links* provide dedicated communication channels from each processor to a maximum of twelve other processors. Each link has a 32-byte deep hardware FIFO buffer located on the IO-1 board of the receiving processor. The interconnection topology of the local links is not fixed and can be changed by manually re-arranging the cables between processors. The most commonly proposed arrangement is an eight-nearest-neighbor planar mesh with toroidal wraparound at the edges, leaving four links available for other purposes such as a shuffle-exchange network [23], cube-connected cycles [19], or next-to-nearest neighbor connections.

3. A network of binary *signal flags* provides for global signaling and synchronization among the processors. Each processor has eight flags (numbered 0-7) which can be set to either True or False. Each flag has three global status signals, ANY, ALL, and SYNC, which can be read by every processor. ANY(i) is True whenever one or more processors have set flag i to True. ALL(i) is True when every processor has set flag i to True. SYNC(i) becomes True when ALL(i) is True, and remains True until ANY(i) becomes False; that is, SYNC(i) is True from the time all processors have set flag i to True until all processors have reset flag i to False. Flag 0 additionally has a FIRST signal which is True on the first processor to set flag 0 and False on all of the others. A processor can individually enable or disable the contribution of each of its flags to the global network, allowing subsets of the whole Array to cooperate without interference from processors which are not involved.

4. A tree-structured *sum/max* network [15] computes the global sum and maximum value of unsigned 16-bit inputs from all of the processors. Special-purpose computational elements at each node of the tree reside on the IO-1 boards of the processors. The results are distributed from the root of the tree back to each of the processors.

The Controller is a Texas Instruments 990/10 minicomputer with the usual complement of peripherals: alphanumeric and graphics terminals, disk and tape drives, and a printer. A special interface connects the global bus and flag network to the Controller. From a hardware viewpoint, the Array sees the Controller as just another station on the global bus. In terms of processing power, the Controller is about one-third faster than an individual processor in the Array.

A simple four-processor system without the global bus or sum/max network was first available for use in the spring of 1980. A serial I/O port on the CPU boards of each processor provided the initial interface to the Controller. An eight-processor Array with full global bus support has been operational since mid-1983. The sum/max network will be installed on a 16-processor system which is currently under construction. The present hardware implementation is designed to permit expansion to a maximum of 36 processors.

3. SYSTEM SOFTWARE

3.1 System Design

The FEM system software design was guided by several fundamental decisions concerning the respective roles of the Controller and the Array. The most basic decision was that the Controller would serve as a front-end, or host, for the Array. In this scenario the Controller, as its name implies, is in charge of the overall system. Activities on the Array are initiated (and sometimes terminated) by commands issued from the Controller to processors in the Array. These *Array commands* may either be directed to individual processors or broadcast to all of them, as appropriate. Table I lists a representative sample of the Array commands which have been implemented. Using this approach, the Controller also supports program development, file storage, and pre- and post-processing of data.

An alternate scenario can be envisioned where the Array is in charge of the system, with the 990/10 minicomputer serving as a back-end peripheral controller. This approach was considered and rejected because (1) the requirements for distributing control functions within the Array were poorly understood, and (2) the code volume required to implement these functions would considerably reduce the amount of memory available for application programs and data on the Array processors.

In order to provide uniform Array monitoring and control functions at the system software level, the decision was made that the Controller would not directly participate in execution of parallel application programs. Instead, the Controller provides overall execution control (halt, resume, kill) and debugging services, monitors and stores output from the processors, and drives the user's terminal display. If an application-specific control process is needed to direct or coordinate execution of the parallel program, one

Command	Description
Processor management	
DEADSTART	Re-initialize the processor.
OFF	Logically disconnect the processor.
ON	Return an OFF processor to service.
Program setup	
DNLDPG	Download program from Controller.
DEFDA	Define data area.
LOADDA	Download data area from Controller.
CONNECT	Establish communication paths.
Execution control	
EXEC	Execute program.
HALT	Suspend execution.
RES	Resume execution.
KILL	Terminate execution.
Debugging	
BRKPT	Set program breakpoint(s).
STEP	Execute one or several instructions.
DUMP	Send memory contents to the Controller.
WRTMEM	Write data to processor memory.
RPS	Interrogate Pascal variable stack.
Post-processing and analysis	
READDA	Upload data area to Controller.
RDEXST	Retrieve execution statistics.

Table I. Partial List of Array Commands Used to Initiate Activities on Individual Processors.

of the processors in the Array can be singled out for this purpose.

A related assumption was that the Controller would be completely dedicated to monitoring and control functions whenever the Array was active. In practice, however, other users are allowed on the system to perform tasks which do not require use of the Array, such as editing files, compiling programs, etc. This produces some degradation of the control software performance, and can affect the Array by reducing the rate at which input and output can be processed by the Controller.

Another important decision was that the Array would not be multi-programmed; that is, all of the processes concurrently active in the Array must belong to the same (parallel) program. Multiple programs belonging to the same or different users are not allowed, even if sufficient idle processors are available. This avoids complicated contention problems for shared resources such as the flag and sum/max networks, and simplifies design of the Controller software. To enforce this restriction, the Array is treated as a resource which must be attached and locked by a user before he can gain access to it.

Finally, it was decided that multi-tasking of individual processors would not be supported; i.e., only one user process may be assigned to each processor. This approach was adopted partly because of the limited memory available on each processor, and is also in keeping with the FEM design philosophy that

(eventually) processors will become a relatively inexpensive resource in a highly parallel environment. In such a context a certain amount of idle time is tolerated if the cost of preventing it is high. This restriction also favors the development of well-balanced, highly efficient parallel algorithms which incur little or no wait time; and certain other overheads, such as scheduling and process-switching, are avoided.

3.2 Software Components

System software for the Finite Element Machine consists of several major components. Figure 2 shows the location of each component within the system and gives a general idea of their interactions. The rest of this section describes each component in detail.

3.2.1 *DX10* . The central component of system software on the Controller is Texas Instruments' standard DX10 operating system [9]. It provides all of the features usually found in a small minicomputer environment including multi-tasking and time-sharing, file and memory management, device drivers, and substantial program development support. DX10 was retained for use as part of the FEM Controller software because it supplied most of the capabilities needed and was easily extensible to include new features.

DX10 provides Controller access to the global bus through two custom device service routines (DSRs) which were added to the system. Higher level software components make supervisor calls (SVCs) to DX10 to perform I/O operations to and from the Array, through the DSRs.

3.2.2 *System Command Interpreter*. The user interface to DX10 is provided by another standard TI product, the System Command Interpreter (*SCI*). SCI implements a menu-driven command language which can be used either interactively or in batch mode. New commands can be written using a programmable job control language which invokes programs and/or other SCI commands. SCI is used by the FEM Controller software to provide a consistent, user-friendly mechanism for invoking FEM-specific operations. The user perceives FEM operations as simply extensions to the standard set of commands available under DX10.

3.2.3 *FEM Array Control Software*. A set of about 40 programs known collectively as FEM Array Control Software (*FACS*) implements the Controller's portion of initialization, data management,

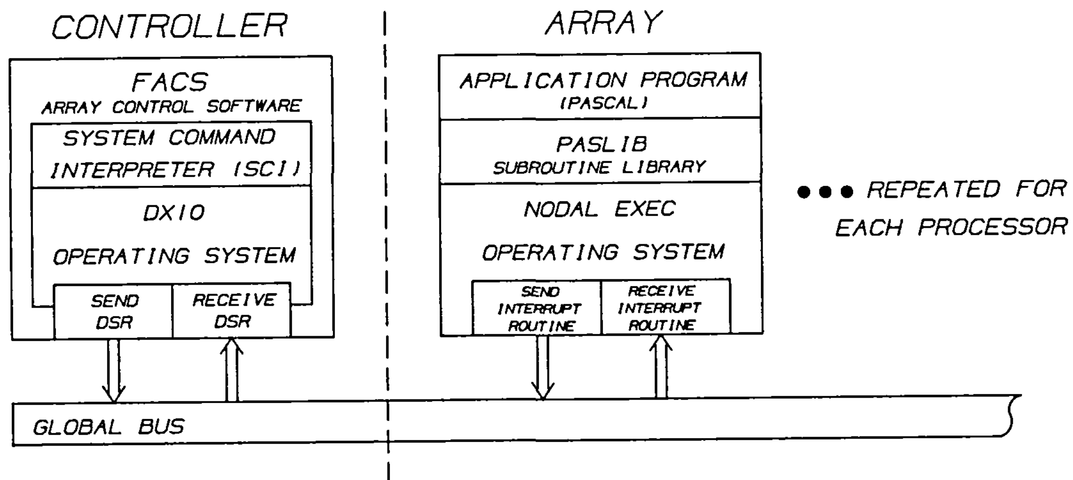

Figure 2. Major software components in the FEM system.

program control, debugging, and post-processing functions for the Array [16]. The FACS programs, invoked by SCI commands, serve as the interface between the user and the Array. (The term *FACS command* is used hereafter to mean "a FACS program invoked by an SCI command".) A typical FACS command generates one or more Array commands (Table I) directed to one or more processors, and evaluates the resulting response which can range from a simple acknowledgement to a substantial volume of data. Most FACS commands also drive the user's interactive terminal which shows the results or status of an operation. A few of the FACS programs perform operations which are purely local to the Controller, such as sorting output from the Array or creating data files to be used by other commands. A partial list of the available FACS commands is given in Table II.

3.2.4 *Nodal Exec.* A small, specialized operating system called *Nodal Exec* is stored in EPROM on each of the processors in the Array. Nodal Exec is divided into two major sections. One section provides services typical of most operating systems: memory management, process control, low level I/O and communication routines, timers, and interrupt handlers. The other section contains a set of command routines which carry out functions requested by the Controller. Each Array command (see Table I) has a corresponding command routine in Nodal Exec. A command monitor receives Array commands from the Controller, checks their validity based on the current state of the processor, and invokes the appropriate command routine.

Although only a single user process may be assigned to a processor, Nodal Exec treats command routines and interrupt handlers as prioritized processes in a limited sense. A high priority process will preempt a lower priority one and run to completion before returning control to the interrupted process. For example, a halt command will preempt a user program, and a timer interrupt will preempt a halt command. Descriptors for suspended processes are stored in what is essentially a last-in first-out stack. This simplified process control environment eliminates the need for time slicing and associated scheduling functions.

3.2.5 *Application Programs.* Parallel programs for execution on the Array consist of a collection of processes, each of which is a sequential program written in Pascal. A library of subroutines, described below, augments the language to provide additional capabilities which are needed in the parallel environment. The Controller assigns processes and data to processors using calls to FACS commands. The FACS calls are ordinarily packaged into parameterized procedures written in the SCI job control language. The combination of SCI and FACS acts as a crude language for manipulating processes and processors. Figure 3 outlines the usual procedure for setting up a program and executing it.

Processes within a single parallel program may execute either the same or different code, depending on the application. If each process is to perform a similar task, most users find it easier to write one general-purpose Pascal program which describes them all. The flow of execution is then controlled by ***case*** and ***if –then –else*** statements based on the processor's self-id or input data. On the other hand, if an application has a number of dissimilar processes, it may be best to write several programs, one for each type of process.

Parallel programs with homogeneous processes can be loaded into the Array quickly by broadcasting the same object code to all processors simultaneously. Heterogeneous processes take more time, requiring each different process-type to be broadcast once. Processors must also be told when to ignore broadcasts which are not intended for them. (Techniques for processor selection are discussed in Section 4.)

3.2.6 *PASLIB.* A library of Pascal-callable subroutines known as *PASLIB* allows application programs to access the unique architectural features of the Finite Element Machine and to obtain services from the Nodal Exec operating system [8]. PASLIB provides subroutines for communication between processes (processors), I/O to and from the Controller, timing, processor identification, and flag, sum/max, and floating-point operations. The most commonly used routines are stored in EPROM on each of the processors to reduce the amount of object code which must be loaded from the Controller.

3.2.7 *Diagnostics.* Although not shown in Figure 2, a substantial body of diagnostic software has been written for FEM to aid in hardware development and maintenance. Diagnostics take many forms: some are embedded in the system software, others operate as standalone programs, and still others run as

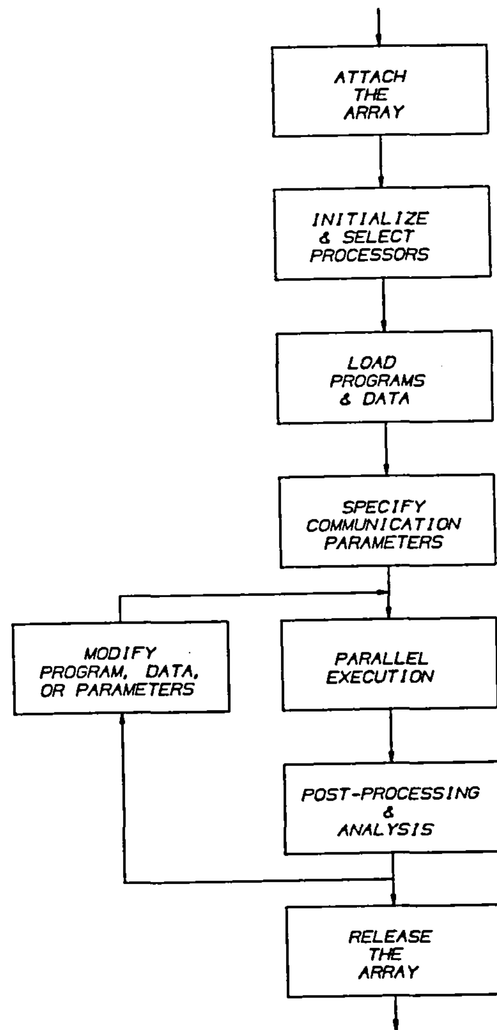

Figure 3. Scenario of a typical FEM session.

application programs under Nodal Exec. While FEM diagnostics will not be discussed in detail, they are critical to the construction and maintenance of the system, and they sometimes present interesting exercises in concurrent programming.

In addition to diagnostics, Nodal Exec, PASLIB, and FACS perform a considerable amount of consistency checking on both hardware and software functions. The general philosophy for writing system software is to assume that the hardware will fail. The aim is not to recover from failures, but to detect and report them so that corrective action may be taken by maintenance personnel. Error reporting from the Array is described in Section 4.

4. SYSTEM CAPABILITIES

A number of important issues confront the software designer when developing a system for an MIMD array such as the Finite Element Machine. This section presents some of these issues and describes how they have been resolved for FEM. These are not necessarily the only solutions possible, nor even the best, but for the most part they represent workable strategies which are currently in use. Several of the ideas presented here are critiqued in Section 5.

Command	Description
Array Management	
ATTACH	Obtain exclusive access to the Array.
RELEASE	Release the Array.
RESET	Initialize the Array and specify a logical-to-physical mapping.
SAC	Select a set of active processors.
Program setup	
LDPG	Load program.
DEFDAI/DEFDAD	Define data areas (interactively or from a control file).
LDAI/LDAD	Load data areas.
SYNCON/ASYNCON	Select communication parameters and establish communication paths (blocking or non-blocking mode).
Execution control	
XFEM	Execute (on FEM).
HFEM	Halt execution.
RFEM	Resume execution.
KFEM	Terminate execution (kill).
Debugging	
SFB	Set/show FEM breakpoints.
SS	Single (or multiple) step execution.
DAMA/DAMC	Dump absolute memory (by address or word count).
MMA/MMR	Modify or browse through memory interactively (absolute or relative addressing).
SPSF	Show Pascal stack (on FEM).
SFS	Show processor state table.
Post-processing and analysis	
RDAI/RDAD	Read (upload) data areas.
RES	Retrieve and analyze execution statistics.
TRACE	Process statistical trace data.
SORT	Sort text output by processor number.

Table II. Representative FACS Commands.

4.1 Processor Management

FEM system software has been designed to support an Array containing from 1 to 36 processors with arbitrary interconnection topologies. Since the Array is being built in stages (4, 8, 12, 16 processors, etc.), and since processors are sometimes removed because of malfunctions, a way is needed to tell the system software which processors are available. This is accomplished with a privileged FACS command used by systems personnel to specify the set of installed, operational processors. When the Array is initialized, the actual set of responding processors is compared to the expected set, and any discrepancies are reported.

This consistency check is important, since an installed processor which fails to respond properly cannot be trusted to have its shared circuitry (flags, sum/max, global bus) set to a correct state and may interfere with the other, operational processors.

In order to keep track of processors, FACS maintains a table describing the current state of each processor. Five states are defined: *Unknown, On, Off, Active,* and *Halted.* A processor not installed in the Array, or which fails to respond following a command, is set to the Unknown state and is unavailable for further use. Following Array initialization, each responding processor is placed in the On state. The On state signifies that a processor is available and idle, awaiting work to do (in the form of an Array command directed to it). A processor may be deactivated by a command which places it in the Off state. A processor which is Off disables its flags and local links, places a zero value in its sum/max output register, and flushes all subsequent global bus input except for an ON command. In this state the processor will behave as though it were not physically installed in the machine. While busy executing a command or user process, a processor is in the Active state. When an Active user process is suspended (by a HALT command or after encountering a breakpoint), the processor is placed in a Halted state. A Halted processor will accept a limited number of commands, notably those used for debugging and to resume or terminate a process. Figure 4 shows a simplified diagram of the allowable state transitions.

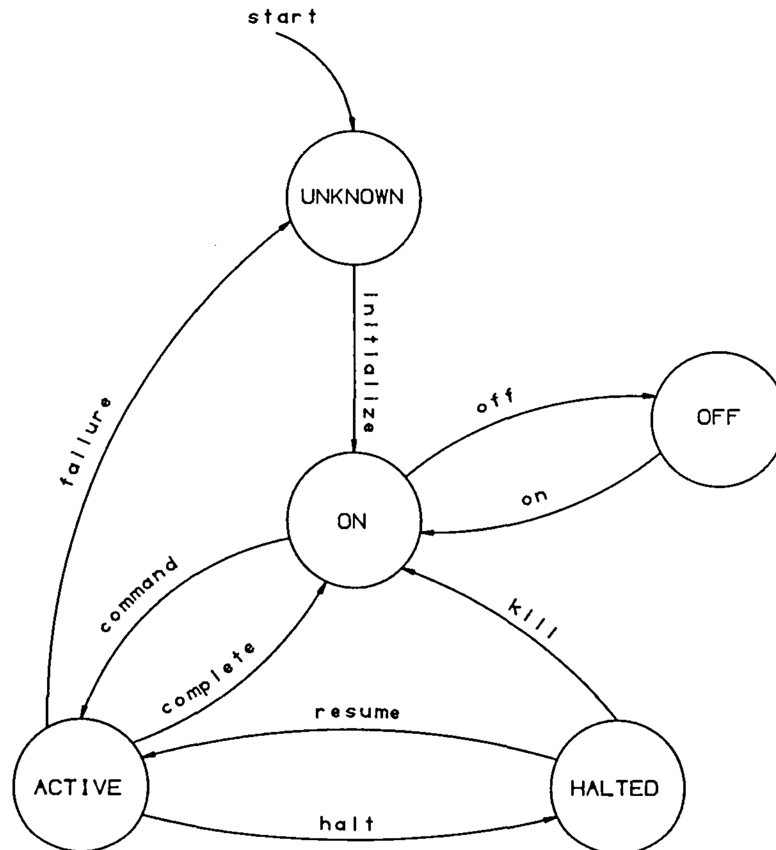

Figure 4. Processor state diagram.

- To direct Array commands to the intended processors, most FACS commands include a "SELECT PROCESSOR(S):" prompt. The response is called a *processor select string,* examples of which are given below:

1,5..9,18
2,4,6,8
ALL

The processor select string is based on a Pascal-like set notation and has proven to be a convenient way to specify subsets of the Array. The exact definition of "ALL" is context-sensitive, but it usually means "all processors which are not in an Off or Unknown state." In certain situations "NONE" may also be an appropriate response. The syntax could easily be extended to include other shorthand notations such as "ODD" and "EVEN".

4.2 Processor Assignment and Mapping

Related to processor management is the task of assigning processes to processors. FEM system software provides support for this with the concept of *logical processors.* For an Array A with p physically installed processors,

$$A = \{a_1, a_2, a_3, \ldots, a_p\},$$

we define a corresponding logical array L composed of p logical processors,

$$L = \{l_1, l_2, l_3, \ldots, l_p\},$$

and a function $m : L \rightarrow A$ such that m is one-to-one and onto; that is, each processor in A is assigned a distinct logical processor number from L. We call m a *logical-to-physical mapping* from the conceptual array L onto the physical Array A. This mapping allows application programs to be written in terms of a collection of n processes which execute on n logical processors numbered 1 through n, $(1 \leq n \leq p)$, without regard to the actual physical processor numbers.

When the Array is initialized at the beginning of each program run, the user specifies the logical-to-physical mapping in the form of a table, or selects one of the default mappings supported by FACS. From that point on, all references to processors (processes) by FACS commands or PASLIB routines are in terms of logical processor numbers. The user need never be aware of physical processor numbers except to define a non-standard mapping function. Sometimes an identity mapping is used, as is the case with most diagnostic programs. In other cases mappings are used to avoid disabled processors, or to improve the fit between the logical communication topology of the program and the physical topology of the local links. Bokhari [6] has shown that, for the general case, the problem of determining if the communication requirements of a program can be satisfied using only local links is equivalent to the intractable Graph Isomorphism problem [20]. He developed heuristics for obtaining a reasonably good fit which reduces usage of the global bus (a shared resource). Bokhari's Mapper algorithm has been modified to reduce its average execution time and implemented on the FEM Controller.

The mapping problem was investigated further in an attempt to partition it into independent subproblems. This investigation revealed that the problem of finding an optimal mapping (minimizing global bus usage) is exactly the Largest Common Subgraph problem as reported in [10] (problem GT49), and as such is known to be NP-complete. No decomposition into independent subproblems was found, but two parallel versions of the mapping algorithm have been implemented on FEM which permit the simultaneous exploration of different mappings on each processor. The first version executes Bokhari's algorithm independently on each processor and sends the best mapping found to the Controller when all processors have terminated. The second version also uses Bokhari's algorithm but updates each processor to the best known mapping at the end of each iteration. In either case, the search is terminated when a perfect mapping is found, or when two successive iterations fail to improve the mapping. Preliminary results indicate that the second version is somewhat faster (on average) in arriving at a good fit. While helpful in reducing the heuristic search time, the speedup attainable using the parallel mappers is bounded by the number of processors available. Thus for sizable processor arrays, the problem of finding an optimal mapping remains intractable.

4.3 Array Input/Output

4.3.1 *Data Areas.* The primary mechanism for input to the Array, and one of the mechanisms for output, is the ***data area***. Data areas are blocks of memory which are allocated on individual processors in response to commands from the Controller. Data areas are considered to be structured entities, with the size and number of data items specified at the time of creation. Data areas may be read from or written to by both the FACS software and processes executing on Array processors. For FACS, access is provided via Nodal Exec by commands sent to the processors. The commands can specify either the entire data area or portions of it, including individual data items. Within processes, data areas are defined with type declarations, usually as arrays, records, or arrays of records. Access is provided via Pascal pointer variables which are initialized with a special PASLIB function. It is the programmer's responsibility to ensure that the type declarations are compatible with the structure of the data area as defined by the FACS command which created it.

In the usual scenario (Fig. 3), data areas are first allocated, and then those used for input are loaded with data from disk files, prior to execution of the parallel program. Following execution, the contents of data areas containing results are uploaded from the Array and stored in files. The FACS commands used to manipulate data areas are driven by control files which specify data areas, logical processor numbers, disk files, etc. FACS provides interactive utilities to simplify creation of the control files.

4.3.2 *Interactive Input.* FEM system software allows parallel programs to interact with the user subject to certain restrictions. Processes active in the Array may request input from the user's terminal by sending an interrogatory message, called a *query,* to the Controller. The query message is usually accompanied by a text string which serves as a prompt to the user. Upon receipt of a query, FACS displays the prompt and waits for input from the keyboard.

The restrictions on queries are (1) that all processes must participate, and (2) that the same input value from the terminal be returned to all of them. Together these two restrictions protect the user from having to respond to a large number of prompts (potentially one from every process) asking the same or similar questions, each of which may need a different answer. The query operation is synchronized across all of the processes so that all query messages arrive at the Controller at about the same time. To avoid screen clutter, FACS displays only the prompt string from a designated reference processor.

To summarize, queries provide interactive input to parallel programs in those situations where all of the processors need the same information. They are most useful for specifying global problem parameters which vary from run to run. For other types of input, data areas are the preferred mechanism.

4.3.3 *Text Output.* In addition to data areas, parallel programs may generate output as lines of ASCII text using PASLIB routines which provide essentially the same capabilities as the standard Pascal WRITE and WRITELN procedures. FACS maintains buffers on the Controller, one for each process, in which lines are assembled as the text arrives. When a line is completed, it is written to a common file which stores text from all of the processes. If the source process is in a set of designated reference processes, the text line will also be displayed at the user's terminal. When the parallel program terminates, the user is left with a file of text lines identified by logical processor number, but ordered only by time of arrival at the Controller. Several techniques which have been developed for ordering this text are described below.

One option is to post-process the data on the Controller to impose the desired order upon it. In many cases, this is as simple as sorting the file by processor number, and FACS provides a utility to do this. For more specific requirements, users may write their own post-processing programs.

An alternate solution is to have the processors cooperate when generating output, so that data is already in the desired order when it is received by the Controller. For many applications, this is easily programmed. The cooperative technique does tend to serialize the output portions of the parallel program, but this is compensated for by not having to post-process the scrambled results; and in any case, the output data stream is processed serially when it reaches the Controller.

A refinement of the cooperative technique is to designate one processor as a reporter for the others. The reporter runs a special process whose sole task is to receive results from the other, computational, processors, and to format them into a finished report. By removing most of the code needed for

generating a finished report from the computational processors, more memory is available for other purposes. The reporter needs only enough of the control code of the computational processors to remain synchronized with them, and most of the report-generation function can be localized in a single processor. The single output stream from the reporter also eliminates the problem of guaranteeing correct message arrival order at the Controller which can arise when the multiple-stream cooperative approach is used.

4.4 Interprocess Communication

4.4.1 *Communication Primitives.* Processors communicate with each other using calls to three PASLIB procedures. In the calling sequences below, *lpn* is a logical processor number, *tag* is a data identifier, *addr* is the starting address of a data item, and n is the size of the data item in 16-bit words.

SEND(*lpn* , *tag* , *addr* , *n*);

SENDALL(*tag* , *addr* , *n*);

RECV(*lpn* , *tag* , *addr* , *n*);

SEND transfers n contiguous words of data, beginning at *addr* and identified by *tag* , to logical processor *lpn*. SENDALL is similar, but transfers the data to all processors predefined by the programmer to be neighbors of the sending processor. SENDALL provides a considerable time savings for programs which can make use of it, since the local link hardware directly supports simultaneous output on multiple designated links. Thus a single output instruction can send the same data item to as many as twelve different destinations. RECV receives n words of data, identified by *tag* , from processor *lpn* and stores them in contiguous locations beginning at *addr*.

The address-and-size calling convention used by the communication routines avoids type conflicts which arise in Pascal when trying to call a procedure with a variety of data types. By specifying only the address and size of a data item, a programmer can send or receive scalars, arrays, records, etc. using a single procedure. The address of any data item can be readily obtained with a special TI Pascal LOCATION function. A SIZE function also assists in determining the size of data items.

The tag parameter is used in several situations to identify data being sent and received. For some parallel programs, the arrival order of data at a receiving processor (from a given source) may not be sufficient to determine its content. In this situation a tag can be used to determine what information is being conveyed. Tags are also helpful in programs which communicate multiple data types between processors. For example, a processor may send integers, reals, and a row of a matrix to a neighboring processor using several SEND calls. Even if the arrival order is sufficient to determine the data type, the programmer may find it more convenient (and safer) to use a different tag value for each type, thereby reducing the danger of inadvertently misinterpreting the data. Finally, tags are essential to prevent overwriting of data in the non-queued receive mode when multiple SEND calls are used. This last application of tags is discussed in more detail in the paragraphs describing data reception. Many programs with straightforward communication requirements do not need to use tags at all, so PASLIB also provides alternate routines which omit the tag parameter.

4.4.2 *System Support* for *Communications.* In order to implement the communication primitives outlined above, considerable software support is needed from Nodal Exec and PASLIB. This section describes interprocessor communication from the standpoint of system software, using the terminology defined in [3].

System software treats the data items specified in calls to the PASLIB communication routines as records of length n words. To allow maximum asynchrony between processes and to minimize the opportunities for a programmer to create deadlock situations, ***buffered interrupt-driven message passing*** is used for both sending and receiving. The overall structure of the communication system is shown in Figure 5.

For SEND calls, an output buffer is allocated of sufficient size to contain the data plus a header word and a trailing checksum. The record length and tag value are placed in the header word and the

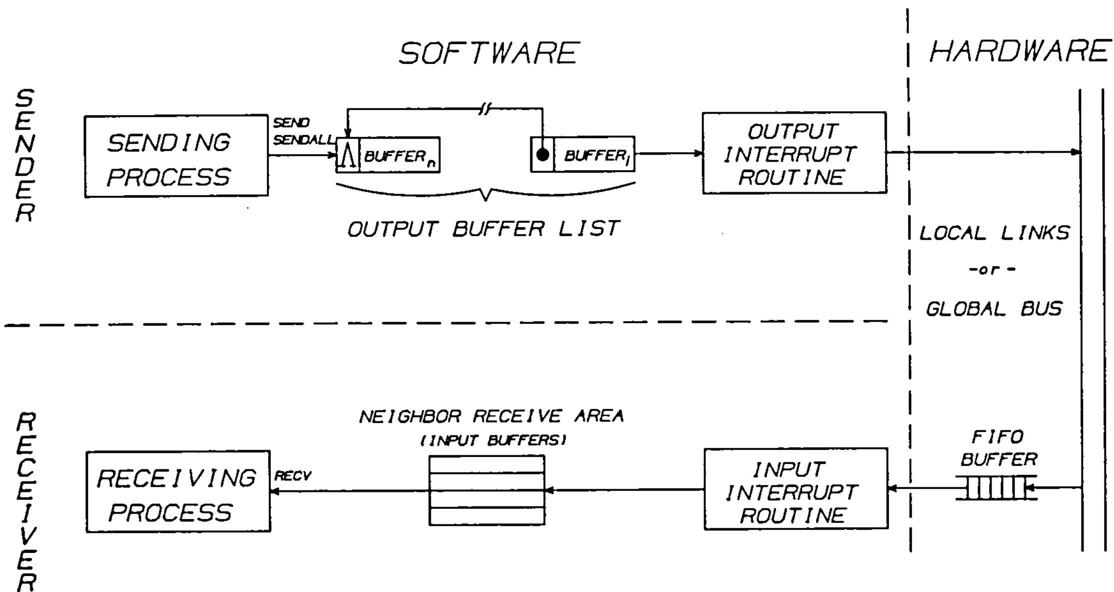

Figure 5. Interprocess communication.

data is copied into the buffer, along with the computed checksum. The SEND routine next determines (from tables maintained by Nodal Exec) if the requested logical processor is accessible via a local link, or whether the global bus must be used. The buffer is then queued for output at the tail of the first-in first-out buffer list for the desired communication medium, either local links or global bus; and the appropriate output interrupt is enabled (if not already enabled for a previous buffer).

When the output interrupt occurs, it signifies that the communication medium is able to accept data. The interrupt routine begins transferring data from the first buffer in the list to the hardware output port, a word at a time. The interrupt routine continues working as long as the hardware is able to accept data, or until there are no more buffers to be sent, in which case the interrupt is disabled. When all of the data in a buffer has been sent, the buffer is returned to free memory. If the hardware FIFOs fill up in the middle of transfering a record (because data is being sent faster than it is being received), the buffer is checkpointed so that the interrupt routine can resume where it left off when the output medium becomes available again.

SENDALL is similar to SEND, except that the data is routed to all (logically) neighboring processors. A single buffer is allocated for all neighbors accessible via the local links, and a destination mask determines which local links are to be enabled during the output transfers. All other neighbors must be accessed via the global bus, and a separate output buffer (separate copy of the data) is allocated for each of these. The buffers are queued as before on the appropriate output lists for service by the interrupt routines.

SEND and SENDALL are usually *non-blocking;* that is, the process is not delayed while the data transmission occurs, since the interrupt routines perform this job asynchronously with respect to process execution. Control is returned to the caller as soon as the output buffers are queued on the buffer lists. In some cases, however, the caller may be temporarily blocked if there is insufficient free memory from which to allocate a buffer. This can occur when a processor is sending much faster than its neighbors are receiving, and multiple send calls have resulted in a large backlog of queued buffers which have not been serviced by the output interrupts. In this situation the SEND or SENDALL routine must wait until sufficient free memory is available, which will eventually occur when receiving processors get caught up and the output interrupts are able to proceed. Hence, the size of free memory (whatever is left after

program code and variables, data areas, input buffers, and system data structures have been allocated) serves as an upper bound on how far sending processes can get ahead of receiving processes.

When data arrives at the destination processor (via either the local links or global bus), it generates an interrupt which transfers control to an input routine in Nodal Exec. The input routine reads data, a word at a time, from the hardware input FIFOs and re-assembles it into records. The checksum is also verified, and any discrepancies are reported.

The input record is buffered in a software structure called a ***neighbor receive area*** (*NRA*). Nodal Exec maintains a separate NRA for each processor defined to be a neighbor of the receiving processor. The structure of the NRA depends on the ***input mode***. Two modes are available, queued and non-queued (or overwriting). In queued input mode, the NRA is an array of t circular queues, each of depth d maximum-sized records (Figure 6a). The value of t corresponds to the range of tag values used in SEND and RECV calls, with $1 \leq tag \leq t$. Each queue can hold d records of maximum size r, or more if the records are of size sufficiently less than r. The values of t, d, and r, as well as the input mode, are runtime parameters. As data arrives, it is placed at the end of the queue corresponding to its associated tag value. Queue overflow is treated as a programming error, indicating that too much data has been sent, too little data has been received (by the process), or the queue was dimensioned too small; and an appropriate error message is generated.

For non-queued mode, the NRA is simply an array of t records, each of maximum size r (Figure 6b). A separate buffer in each NRA is used to assemble records as they arrive. When a record is completed, it is copied into the array slot corresponding to its tag value, replacing the previous record of the same tag. The copy operation is implemented as an atomic action to prevent partially updated records from being accessed.

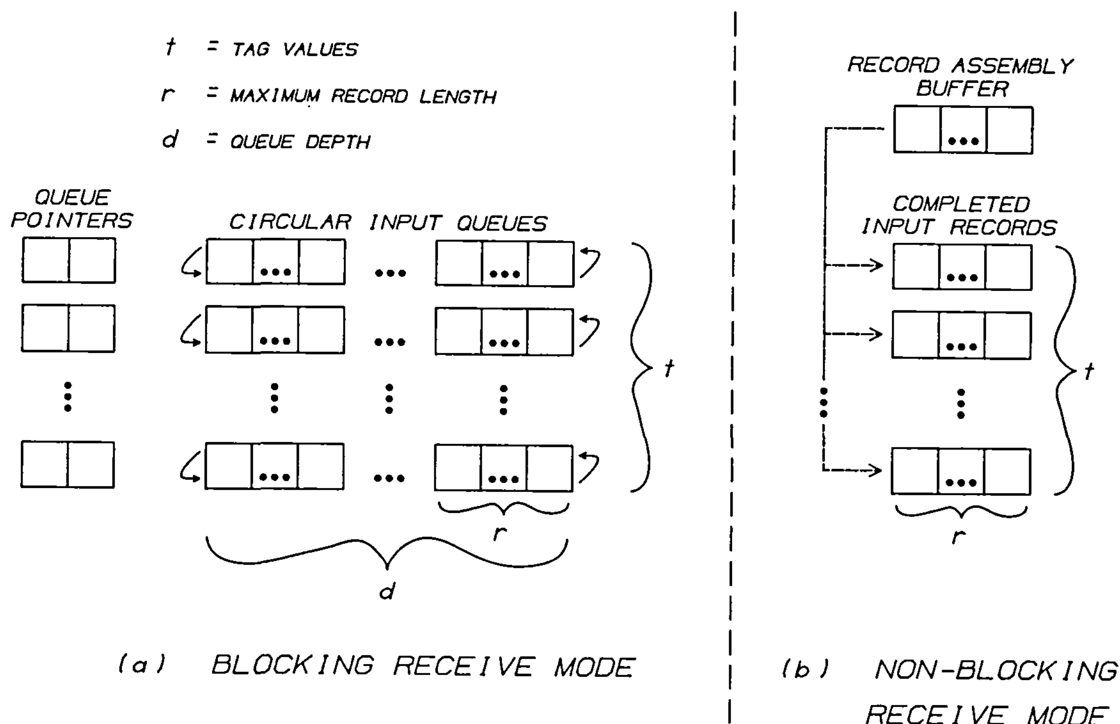

Figure 6. Structure of neighbor receive areas. (a) An array of queues for blocking mode. (b) An array of records for non-blocking mode. Transmitted records can be any size less than or equal to r.

The RECV routine, called by an application program, retrieves records from the NRA and returns them to the process. The RECV call will either be ***blocking*** or ***non-blocking***, depending on the input mode being used. For queued input mode, the RECV routine will wait until data arrives if it is not already available, thereby blocking the calling process. In non-queued mode, the RECV call never blocks the process, but always returns the most recently received value of the record. In the absence of other synchronization, if the receiving process is running faster than the sending process, the RECV call may return the same value more than once; but, if the receiving process is running slower, some values may be overwritten between RECV calls and never seen by the receiving process.

The non-blocking (non-queued) form of RECV was implemented specifically to allow development of asynchronous and chaotic algorithms [4,7]. The non-blocking RECV is also useful for monitor processes which periodically want to examine execution progress, but do not wish to make RECV calls every time a neighbor issues a SEND. If necessary, a non-blocking RECV can be synchronized with a corresponding SEND by using flag barriers, discussed in the next section.

4.5 Synchronization.

The blocking form of RECV can be used to loosely synchronize processes with a message passing scheme, and in some situations this technique may be appropriate. However, the preferred synchronization primitive for most applications is the ***flag barrier***, attributed to Jordan [13] and refined by W. Calvert. A barrier on signal flag i causes all processors with flag i enabled to wait until the slowest processor has caught up. The barrier algorithm, using the SYNC flag signal described in Section 2.2, is given below.

```
{ Barrier on flag i.                                  }
{ Each participating processor must have flag i enabled }
{ and set to false before entering the barrier.       }

while SYNC[i] do
  {wait for all processors to set i false} ;
flag[i]:=true;
while not SYNC[i] do
  {wait for all processors to set i true} ;
{all processors are synchronized at this point}
flag[i]:=false;  {leave flag in consistent state}
```

The key here is the SYNC signal whose value is based on transitions of the ANY and ALL signals. The first ***while*** loop will ordinarily fall through on the first evaluation, but is necessary in the general case where processes may be running at drastically different rates. After determining that the SYNC signal is false, each process(or) sets its flag i to true as it arrives at the barrier. As soon as all participating processes have set the flag, the SYNC signal becomes true and the second ***while*** loop is terminated. At this point the processes are all synchronized (give or take a couple of instructions). Upon exiting the barrier, each process sets its flag to false, leaving it in a consistent state for the next execution of a barrier.

The barrier operation can be thought of in several ways. As the name implies, it is frequently used to prevent processes from proceeding until all processes have had a chance to take some action, for example transmitting initial values to neighbors or setting another flag. The barrier can also be thought of as enforcing the opposite of mutual exclusion, i.e., ensuring that all processes are in the same piece of code (the barrier routine) simultaneously. This is easily extended to larger code segments by bracketing the segment with barriers at the entry and exit points, which guarantees that all processes will be within the segment at the same time. The term ***mutual inclusion*** might be appropriate to describe this situation, which is a fairly common occurrence in MIMD programming. The barrier operation is also similar to a ***busy-wait*** protocol with flags substituted for spin locks [3], and is semantically equivalent to Schwartz's *SYNC* operation for a proposed ultracomputer programming language [21].

The barrier operation is very efficient (excluding wait time), requiring only about seven machine instructions on the TMS9900. In the absence of programming language and compiler support, however,

the barrier operation has been implemented as a PASLIB procedure, which requires additional instructions for subroutine entry and exit. Even so, experience has shown that for well-balanced algorithms with work divided evenly among processors, wait time is minimal and synchronization overhead due to barrier calls is negligible.

4.6 Global Decision-making

There are many instances in parallel algorithms when the action to be taken by one or more processors depends on global conditions across all of the processors. In the absence of shared memory, an alternate mechanism is needed for signaling such conditions. This task could be accomplished by passing messages among processors, but the overhead for this is relatively high. The Finite Element Machine provides direct hardware support for this problem with the ANY and ALL flag signals. PASLIB extends this support to the application program level with corresponding Boolean-valued ANY and ALL functions. The ANY signal is equivalent to a multi-way OR operation on the inputs from all (participating) processors, while ALL is equivalent to an AND operation. Thus a processor can determine the Boolean value of some local condition and set its flag accordingly. Then, with proper synchronization, the global status across all processors can be quickly determined.

A common example of this occurs when testing for convergence in iterative numerical algorithms. Each processor is working on a portion of the problem, and at the end of each iteration makes a local convergence check. Local convergence does not imply overall convergence, however, since other processors may not yet have converged; so the iteration must continue until all processors have converged during the same iteration. A program fragment illustrating global convergence checking is given below:

```
{ Two flags are used:                    }
{   flag i - global convergence test     }
{   flag j - process synchronization     }
{ Both flags are initially set to false.}
            . . .
repeat
            . . .
  {computation and communication code}
            . . .
  flag[i]:= {Boolean-valued local convergence test} ;
  barrier(j)
until ALL(i);
            . . .
```

The barrier immediately preceding the ALL test is necessary to ensure that all processors test ALL at the same time; otherwise, some processors might see one value for ALL while others might see another. The above algorithm assumes that every processor will finish the ALL test before any processor can progress as far as the local convergence test during the next iteration; and this is usually a reasonable assumption, given the relative times for computation and communication versus flag testing. To be strictly correct, however, a second barrier is needed in the ***repeat*** loop at some point before the local convergence test to guarantee that all processors have finished the ALL test before the flags are updated on a subsequent iteration. (In other words, the ALL test should be a mutually inclusive operation.)

Although not currently supported by PASLIB, the FEM hardware also allows flags to be grouped in small sets of eight or fewer members. The flag network could therefore be used straightforwardly to perform global set union or set intersection operations if an application needed such a capability. Larger sets could be processed by making several passes through the network using slices of a few bits at a time.

Some additional possibilities for global control using the sum/max network are described in the next section.

4.7 Sum/Max Operations

Perhaps the most unique feature of the FEM architecture is the cooperative sum/max network (see Section 2.2). Since the network will not be installed until the 16-processor Array is completed, very little system software has been written to use it. However, it is instructive to examine some of the potential applications for this versatile network, and the PASLIB subroutines which will be needed to support them.

4.7.1 *Global Calculations.* The sum/max network was added to the FEM architecture based on an analysis of global computation requirements for the solution of systems of linear equations using the iterative conjugate gradient method [15]. Parallel implementation of conjugate gradient requires a summation across values from all of the processors at each iteration. Techniques for computing this using the global bus and local links for communication were felt to be too time consuming.

The sum/max hardware designed for FEM will directly compute the sum and maximum of 16-bit unsigned integer inputs from all of the processors. Minima can be easily obtained by passing the one's complement of the input values through the network. Operations on more complex data types such as signed integers and floating-point numbers can be built up with software which makes several passes through the network.

To support global summations, PASLIB will need to include a SUM function:

FUNCTION SUM(x:*real*; i:*flag*):*real*;

where x is one processor's contribution to the sum, and i is a flag used for internal barrier synchronizations within the function. The function result will be the global sum.

4.7.2 *Global Control.* The sum/max network can also be used to control global or distributed decision-making, offering somewhat more flexibility than the ANY and ALL flag signals. Operations analogous to ANY and ALL can be performed as follows: To signal a true condition, a processor writes a value of 1 to its sum/max output register; to signal false, a 0 value is written. To test for ANY, either the sum or maximum can be compared to 0; a non-zero result implies ANY is true. To test the ALL condition, the sum is compared to the number of active processors; when they are equal, ALL is true. The number of processors can be found dynamically using an earlier pass through the sum/max network in which each processor outputs a 1. This technique for signaling ANY and ALL provides more information than the flag network, since the number of processors signaling a true condition can be determined directly from the sum. Software overheads for these operations using sum/max should be only slightly greater than for the corresponding flag operations.

The sum/max network can also be used to control the execution sequence of serial sections in parallel algorithms. One example of this is the cooperative output technique described in Section 4.3.3. We will assume that processors wish to execute a section of code in order, based on the value of some local variable x. To support this, three PASLIB routines are needed:

FUNCTION MAX(x:...; i:flag):...;

PROCEDURE CLEAR;

PROCEDURE DISABLE(i:flag);

Function MAX returns the global maximum value of x across all of the participating processors. Flag i is used by MAX to perform internal barrier synchronizations. (In practice a family of MAX functions will be needed, one for each different data type of x.) Procedure CLEAR sets the sum/max output register to zero, effectively eliminating a processor's contribution to the sum and maximum. Procedure DISABLE disables flag i, eliminating the processor's contribution to the flag network for that flag. The algorithm is as follows:

```
selected:=false;
repeat
  if MAX(x,i) = x then
```

```
        begin {this processor is selected}
          . . .
        selected:=true
        end
    until selected;
    CLEAR;
    DISABLE(i);
```

When a processor is selected, the other processors will return to the beginning of the loop and hang on a barrier in MAX until the selected processor disables its flag, at which time they will proceed. Depending on the application, a processor can either be allowed to proceed following the DISABLE call, or else held up using a barrier on another flag until all processors have finished the loop. As written above, the algorithm allows processors with identical values of x to proceed in parallel; if this is not desirable, additional logic could use the FIRST flag signal to select one processor at a time. Also note that the ordering produced here is by decreasing value of x. To obtain an increasing sequence, a MIN function could be substituted which returns the global minimum.

4.8 Debugging

Debugging parallel MIMD programs is often more difficult than debugging sequential programs because of potentially complex interactions among multiple asynchronous processors. To aid program debugging, FACS and Nodal Exec together provide a set of tools which allow users to inspect and modify memory and registers, set breakpoints, and execute programs in a single-instruction mode. In addition, an "intelligent" Pascal debugger is available which incorporates knowledge of the runtime variable stack and current nesting level.

FACS provides a sophisticated interface to these debugging utilities with screen-oriented interactive displays driven by special function keys on the user's keyboard. Some commands select processors using a processor select string (see Section 4.1), while others are appropriate to only a single processor at a time. An example of the latter case is shown in Figure 7, which is a typical display from the Modify Memory command. This command displays a 16-word segment of memory from a single processor, and function keys can be used to move the cursor within the display (in order to modify memory contents), to scroll the display forward or backward, or to jump to a completely different area in memory. To view the same address range on another processor, the user presses the F7 key and enters the new processor number in response to a prompt which appears on the screen. A maximum of four keystrokes are required.

Memory can be referenced in either of two modes, absolute or relative. Absolute mode uses absolute addresses within a processor's address space, and is useful primarily for displaying system data structures. Relative mode uses addresses which are interpreted by Nodal Exec as offsets from a program's load point (program-relative addresses). These correspond to addresses listed in the module map produced by the link editor, and are used for examining memory within the bounds of the user's program. In the current system, variables must be referenced as hexadecimal offsets within the Pascal stack, according to variable maps produced by the Pascal compiler. In principal, however, the variable maps could be used to provide automatic symbolic addressing capabilities.

For large numbers of processors, debugging could become a tedious process. In practice, though, the difficulty is not great. Many programs can be successfully debugged using a small number of processors to find logic problems which generalize to larger numbers of processors. Often, selecting one or two key processors for detailed inspection is sufficient to isolate a problem.

One of the more common sources of difficulty related to parallelism is deadlock between processors. On FEM, this can occur in only two locations, the flag barrier routine and (the blocking form of) the RECV routine. A quick inspection of program counter values across all of the processors (provided in one screenful by a FACS command) usually reveals the offending processor(s). More intensive debugging efforts can then be directed at this processor or its neighbors to pinpoint the cause of the problem. The simple technique of inserting strategically placed write statements within the program code is also quite effective.

```
INSPECT & CHANGE ABSOLUTE MEMORY      PROCESSOR  3

                FA80:  0003
                FA82:  0000
                FA84:  FFFF                  FUNCTION KEYS
                FA86:  C123
                FA88:  C523
                FA8A:  C020                    F1 - NEXT LOCATION
                FA8C:  C022                    F2 - LAST LOCATION
                FA8E:  FFFF                    F3 - COPY LAST VALUE
                FA90:  FFFF                    F4 - CHANGE REPLACE
                FA92:  FFFF                    F5 - NEXT PAGE
                FA94:  1000                    F6 - LAST PAGE
                FA96:  FFFF                    F7 - NEW PROCESSOR
                FA98:  1000                    F8 - NEW ADDRESS
                FA9A:  FFFF
                FA9C:  FFFF
                FA9E:  0000


TYPE "CMD" TO TERMINATE
```

Figure 7. Typical screen display for an interactive debugging command. In this example, a 16-word "page" of memory from logical processor 3 is displayed, beginning at address FA80. If the user modifies any of the displayed fields, FACS transmits the changes to processor 3's copy of Nodal Exec, which updates the corresponding memory locations.

4.9 Error Reporting

As mentioned previously, FACS, Nodal Exec, and PASLIB contain substantial hardware and software error checking code. Errors from all three sources are reported to the user in a standard format by FACS. Errors which occur on the Array are reported to the Controller using a short message which contains an error code and information about the source of the error. To conserve the limited memory on the processors, text for error messages is stored in a direct access file on the Controller. When an error message arrives at the Controller, FACS reports it to the user, identified by processor number. An error summary is also stored in a file and displayed when the FACS command terminates.

In certain situations, the volume of error messages may be quite large because of cascading errors from multiple processors. Since only the first few errors from each processor are usually needed to identify a problem, FACS can flush subsequent error messages after a pre-set limit is exceeded to keep the size of the error report manageable.

4.10 Confidence Checking

Because hardware and software development on the Finite Element Machine proceeded in parallel, programs were often run on a machine which was only marginally operational, and failures were common. For long programs which did not generate output at regular intervals, there was no easy way (short of suspending the program) to determine if execution was proceeding normally, until the point had been reached at which results should be generated. To address this problem, a *check-in* option was added to Nodal Exec which could be enabled at runtime to give an early indication of catastrophic hardware or software failures. If enabled, the check-in option causes Nodal Exec to send a short, one-word message

from each processor to the Controller at 60-second intervals. Every 65 seconds, the FACS program monitoring Array execution looks to make sure that check-in messages have been received from each processor during the preceeding time interval. Processors which fail to check-in are promptly reported to the user. Since the check-in option is driven by a timer interrupt within Nodal Exec, it will not detect minor hardware failures or software errors such as deadlocks or infinite loops. Nevertheless, check-in does provide a useful service in potentially unreliable environments.

4.11 Performance Analysis

While the performance of computationally intensive programs is of interest on sequential computers, it is even more important on parallel computers, especially in a research environment. Synchronization, communication, and other global operations, as well as differing workloads, can all introduce factors which reduce the efficiency of parallel programs [1]. To support performance analysis, FEM system software includes three major tools: (1) timing routines, (2) execution statistics, and (3) statistical tracing.

The CPU in each FEM processor contains two interrupt-driven timers, and PASLIB provides user programs with access to both of them. One timer, with a 16-millisecond period, is under control of Nodal Exec and is used to measure elapsed process execution time. The elapsed time may be read at any point by a user program. The second timer, with a programmable period from 1 to 349 milliseconds, is controlled by user programs, which can start, stop, and read the timer at will. These two timers provide considerable flexibility for making internal measurements of code sections.

Nodal Exec and PASLIB also keep a running count of floating-point, flag, and I/O operations, as well as information about memory and buffer allocation. These execution statistics can be uploaded to the Controller following program termination and post-processed by a FACS utility to yield additional derived measures such as average floating-point rate and communication workload. This information provides the programmer with general insight into program behavior, as well as accurate counts of performance-related system operations.

To provide a detailed breakdown of execution time, Nodal Exec includes a trace option which can be enabled at runtime. The trace option samples the processor's program counter at a specified rate and sends the values to the Controller, where they are saved on a file by the FACS execution monitor. The sampling rate and processors selected for sampling can be varied to achieve a balance which neither overloads the Controller nor significantly perturbs Array execution, but is frequent enough to give an adequate sample size. Often, tracing one or two key processors is sufficient, since other processors will behave in a similar manner. The trace data is post-processed on the Controller to give a summary of the approximate percentage of execution time spent in each program module, including time spent in Nodal Exec and PASLIB modules. An instruction-level breakdown may also be requested to pinpoint time spent in specific loops, etc. The TI Pascal compiler includes a reverse assembler so that particular instruction addresses can be located within high-level code.

5. EXPERIENCE WITH THE SYSTEM

System software for FEM, in one form or another, has been in use since early 1980. The version described in this paper has been in production use since October 1983, and substantial portions of it were being used experimentally for up to two years before that. This section summarizes experiences with the FEM hardware/software system and suggests areas for additional work.

5.1 Design Evolution

The system software has evolved through several versions over a period of about four years. The initial software requirements (as well as the hardware architecture design) were based on physical properties of the finite element method and analysis of a few iterative algorithms for solving finite element problems. These requirements soon became too restrictive as algorithm developers continually widened the scope of their research. As a result, successive versions of the system software have become increasingly general-purpose. This process is a prime example of the *system sculpture* model of software development [5],

where the applications are experimental and system requirements are therefore subject to frequent revision. After several iterations of this process, the authors now believe that the range of parallel algorithms of potential interest in structural engineering analysis is so broad that the system requirements are essentially the same as those needed for general-purpose scientific computing in an MIMD environment.

5.2 System Software Critique

5.2.1 *Data Areas.* The least satisfactory aspect of the current system design is the reliance on data areas as the primary mechanism for input to processors in the Array. Data areas suffer from the following drawbacks:

1. *Problem partitioning.* Except in the case where the same data is being used by all processors, global input files must be split into a multitude of smaller files which will be sent to individual processors. Since the partitioning requirements depend on the details of the parallel algorithm involved, it is difficult to develop general purpose system software to accomplish this task. Although users are free to write their own partitioning programs to run on the Controller, most balk at this extra step. Having written one or more programs specifying processes for their parallel program, they are reluctant to write another program to partition their data. Their usual reaction, at least for regular problem topologies, is to enhance their parallel code to generate the data in place based on global problem parameters. However, this hard-coded approach is not suitable for general purpose programs which must contend with the irregularities of realistic problems.

2. *Performance.* Loading data areas to and from the Array is a relatively slow process because of high I/O overheads in the DX10 operating system and the many layers of software through which the data must pass going from secondary storage to the global bus. A special-purpose operating system for the Controller, with fast disk-to-global bus I/O as one of its primary goals, could alleviate much of this problem.

3. *Memory limitations.* A processor can only operate on external data which is resident in data areas or is sent to it by a neighboring processor. Therefore the amount of memory available for data areas constrains the volume of data which can be processed by the Array.

Potential solutions to these problems are discussed in the section dealing with future work.

Data areas are very useful in two situations, however. The first of these occurs when it is desirable to send the same data to most or all of the processors. Examples of this include global problem parameters such as geometry specifications and material properties. In this case no partitioning is required and the common data can be simultaneously downloaded to all of the processors using a global bus broadcast.

The second situation occurs when the processes comprising a parallel program become too large to fit in memory on the individual processors. (This typically happens when a process reaches roughly 1000 lines of code.) In this situation the program must be broken (if possible) into several distinct phases which are successively loaded into the Array and executed. Data areas, since they exist independently of processes, are a convenient way to pass intermediate results between phases, without having to store and reload the results from disk.

5.2.2 *Interprocessor Communication.* The system communication software has been generalized several times in response to expanding requirements from algorithm developers. The current software is flexible enough to support a wide range of communication needs. There are, however, situations in which even more flexibility would be useful. The system assumes that communication requirements are uniform for all paths between processors. The programmer specifies (via a FACS command) the receive mode (blocking or non-blocking), maximum record length, and queue depth (for blocking mode). These specifications then apply to communication paths between each pair of neighboring processors, and

neighbor receive areas (NRAs) are allocated accordingly. Furthermore, each communication path is assumed to be bi-directional, so that NRAs are allocated at both ends.

For most of the applications run to date, these assumptions have been reasonable. As parallel programs become more sophisticated, however, we expect an increase in heterogeneous process types, with a corresponding non-uniformity of communication requirements. In this environment, it would be desirable to specify communication parameters separately for interfaces between each type of process. For example, some communication paths might be one-way and non-blocking with a short record length, while others might be bi-directional, blocking, and have long record lengths. To conveniently specify this level of detail would require appropriate programming language support.

The price of generality in the communication system is relatively slow performance. For blocking receives, the total time from initiation of a SEND call by one process to termination of the corresponding RECV call by another process ranges from 0.3 to 2.5 milliseconds (roughly 50 to 430 instruction times) per word of data transferred (Fig. 8). This assumes that the receiving processor is ready to accept data as soon as it arrives. The exact cost per word varies with the record length. Studies [17] have shown that execution times for the SEND, RECV, and interrupt routines are divided into two components, a fixed overhead for invocation and an incremental cost for each word transmitted, and that the fixed overhead is substantially larger than the incremental cost. Therefore, there are significant advantages in sending and receiving a single large record rather than several smaller ones, since the large fixed overhead is only incurred once and can be amortized over a large number of data words, resulting in reduced cost per word. The minimum cost occurs for maximally sized records (255 16-bit words in the current implementation) and the maximum cost is realized when a single word at a time is transferred.

The communication times shown here are not the optimum attainable with the present strategy. By restructuring the neighbor tables, modifying the interrupt priority scheme in Nodal Exec, and carefully optimizing the rest of the communication code, we expect to reduce communication times by 25-50%. Other contributions to communication overhead (not necessarily in order) include subroutine entry and exit housekeeping, parameter checking, output buffer allocation and deallocation, queue pointer maintenance, transmitting and verifying checksums, and data copying to and from buffers.

It seems clear that assembly language routines tailored to each different application could result in better communication performance than the general-purpose software provided by Nodal Exec and PASLIB. On the other hand, most application programmers would rather not be concerned with such low-level details, and turning over interrupt control to user programs could seriously compromise Nodal Exec's ability to maintain the processor in a consistent state.

Furthermore, high communication costs do not necessarily imply that a program's parallel efficiency will be low. Rather, it is the ratio of communication time to computation time which (along with other factors) influences an algorithm's efficiency [1]. Thus communication time can be large as long as the computation load is even larger. The value of this ratio is important in designing algorithms and for determining optimum partitioning of a problem. This latter consideration in turn may influence the amount of data memory needed per processor.

5.2.3 *Programmability.* In spite of its complexity, the FEM system is not unduly difficult to use. Experienced programmers can ordinarily have parallel programs running within a few days of being introduced to the system. The greatest obstacle to use is the large body of information which must be assimilated. In addition to the usual procedures for editing and compiling programs and manipulating files, programmers must also become familiar with the capabilities provided by FACS and PASLIB, and must learn the rudiments of the SCI control language. Also, the disjoint nature of control procedures written in FACS/SCI with respect to the Pascal processes which form a parallel program leads to difficulties in manipulating global data files. (This is related to the shortcomings of data areas mentioned previously.)

A potential solution for these problems is a high-level parallel programming language which encompasses the concepts of parallel processes, communication channels, and data partitions. Ideally, processes would be declared in a manner similar to that in which procedures and functions are declared in Pascal. Communication channels and data partitions would be defined globally in an outer nesting level. The number of processes, communication parameters, and dimensions of data partitions should be

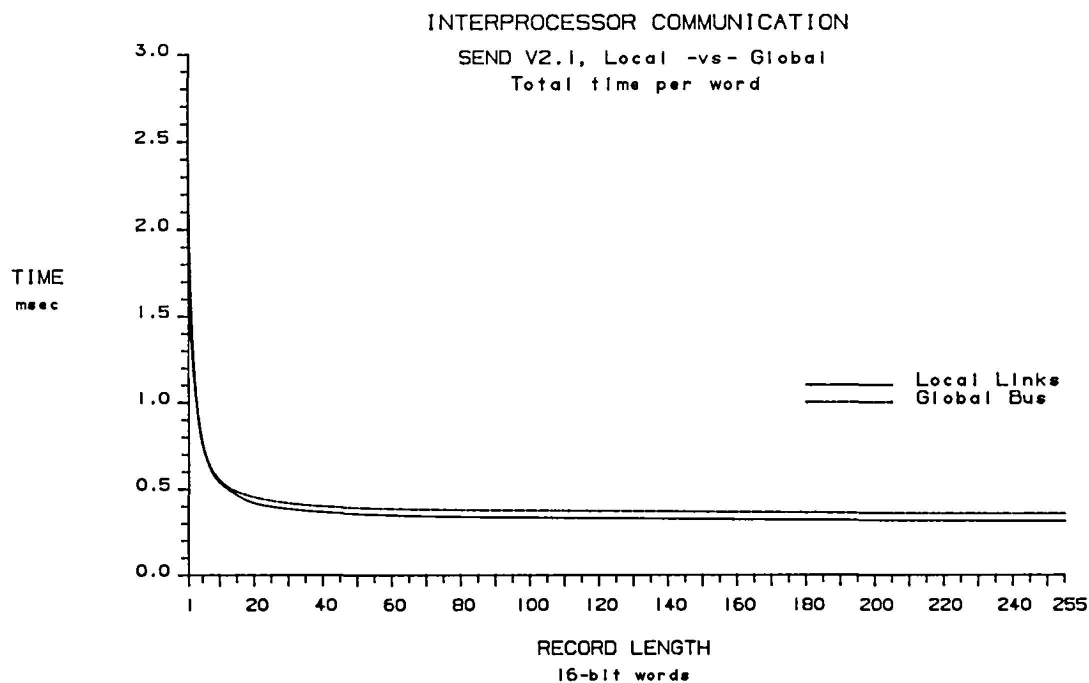

Figure 8. Interprocessor communication costs as a function of record size. The discrepancy between local link and global bus times is due to differences in interrupt handling software, and not to hardware considerations.

dynamically determined at runtime to allow adaptation by the program based on the number of processors available and size of the problem. In such an environment, FACS commands (or their equivalent functions) become implicit operations invoked by the runtime system, and most PASLIB routines are subsumed within features of the language.

Our experience also indicates that designing parallel programs is not especially difficult. As programmers become more accustomed to thinking in parallel and as the body of parallel programming techniques grows, the difficulty will decrease. The greatest challenge is to produce efficient parallel programs, and this requires a good understanding of the underlying system and hardware architectures. While such an understanding is also needed in optimizing performance on conventional sequential computers, the problems are compounded in MIMD programming because of the effects of synchronization and communication. The situation is similar to that for vector computers, where detailed knowledge of the hardware architecture is often needed to effectively utilize the machine.

5.2.4 *Scalability.* All of the system software written for FEM assumes a maximum Array size of 36 processors, and easily supports any number of processors less than that. However, it is interesting to consider what portion of the design could be successfully scaled up to much larger Arrays. Essentially all of the code in Nodal Exec and PASLIB is independent of the number of processors available, since it is concerned primarily with functions local to individual processors. One notable restriction is on the

number of neighbors which any given processor may have. Since separate input buffers (NRAs) are allocated for each neighbor, the amount of memory available on a processor limits the number of communication paths which can be accommodated. This restriction may not be severe, since algorithms which require such a large number of neighbors are probably inherently ill-suited for a mesh-connected array architecture.

The FACS software on the Controller is more sensitive to the Array size. One example of this is the command/acknowledge protocol used to direct operations on the Array. While commands are frequently broadcast, each processor obeying the command must individually acknowledge compliance, and this takes time linear in the number of processors. Another example is the sequential loading or unloading of data areas to or from each processor. Experience with larger Arrays containing 16 or more processors is needed to fully assess the impact of bottlenecks in the Controller software.

5.3 Applications Performance

It is impossible to make a general statement about performance of parallel programs on the Finite Element Machine. The only firm rule is that performance is highly algorithm dependent. Parallel efficiencies ranging from less than 40% to better than 95% have been observed. Typical efficiencies are in the 70-90% range, although this is expected to improve somewhat with better communication software. Algorithms which perform well generally have the following characteristics:

1. Wait time is minimal. The workload is well-balanced between processors, and they are approximately synchronized. Send and receive operations are sequenced so that communicated data is usually available when needed.

2. The computation load is high with respect to the communication load. This usually implies a limited number of neighbors and a minimum number of send and receive calls. Algorithms which can make use of SENDALL have a distinct advantage.

3. The overhead from extra control code needed to support parallelism is low. Processes spend only a small amount of time determining their position within the larger problem, and global interactions are limited to flag (and presumably sum/max) operations.

4. Redundant computation is avoided. There is a clean partitioning of the problem between processes, with very little overlap. In some cases, however, redundant computation may be preferable to communication.

The importance of parallel efficiency is open to interpretation. In a processor-rich environment, efficiencies of 50% or less may be perfectly acceptable if (1) the speedup obtained is still significantly better than alternative sequential or parallel methods, and (2) the solution is cost-effective. This latter point means that if processors are going to be used wastefully, their cost should be relatively low.

5.4 Machine Architecture Critique

Based on experience with the system software and a number of different applications, we are able to make some general comments about the present implementation of the FEM architecture. These are summarized in the following paragraphs.

5.4.1 *Flag Network.* The signal flags have proven to be a very efficient mechanism for synchronization and global signaling. For many programs, the overhead due to their use is negligible. The number of flags available (eight) is also sufficient for the types of algorithms

being run to date, with most programs requiring from two to four flags.

However, as the number of processors available increases, we expect experimentation with programs which are structured as semi-independent clusters of processes. Synchronization and communication requirements within each cluster are independent of those in other clusters, except for occasional points at which results from each cluster must be communicated to other clusters or controlling processes. In such a situation, each cluster would require two or three flags for use locally, and another flag or two would be needed for overall synchronization and control. Using this approach, the number of clusters on the current FEM would be limited by the availability of flags to three or four, at most.

5.4.2 *Communications.* The interprocessor communication hardware (local links and global bus) is fast enough so that data transmission times are almost negligible compared to the execution times of the communication software. Therefore the hardware transfer rates will not be a performance bottleneck in the current system. This disparity between hardware and software speeds suggests that a better balanced system might be obtained by implementing portions of the communication algorithms in hardware, possibly through the use of a special-purpose I/O unit associated with each processor. However, the usefulness of such features is likely to be problem-dependent, and more study is needed.

One potential source of performance degradation is contention for the global bus. Studies have shown that for a 36-processor Array, program delay due to bus contention will not occur, even under worst-case conditions [17]. In fact, in its intended role as a backup mechanism where direct local link paths are not available, the global bus should adequately support a few hundred processors. The exact point at which bus contention becomes a problem is, of course, application dependent, and could vary considerably.

Although mesh-connected arrays are often thought to be restrictive in terms of allowable communication topologies, this is not a serious problem for FEM. The large number of links available on each processor (twelve) and the ability to reconfigure them (albeit manually), coupled with the global bus as a backup path, make for a very flexible communication structure. As described previously, the system software takes advantage of this flexibility to make the transmission medium transparent to the user, while still allowing him to make optimum use of a particular topology by specifying a mapping function.

5.4.3 *Memory.* The amount of memory available on each processor is clearly insufficient for the types of programs currently being written for the machine. Of the 32K bytes of RAM available, approximately 2K are reserved for use by Nodal Exec. The remaining 30K are divided between program code and variables, data areas, neighbor receive areas, and output buffers. It is not uncommon for a moderately complex program to reach or exceed 30K, leaving little or no space for data and buffers. Given the current code density, it is recommended that RAM be expanded to at least 128K bytes; and a RAM of 256K or 512K would not be excessive, especially for large problems with double precision variables. For many applications, larger local memory translates to more local computation and a lower ratio of communication to computation, thereby improving parallel efficiencies.

The 16K bytes of EPROM allocated to Nodal Exec and PASLIB are marginally adequate. However, doubling the size of this memory to 32K would allow more functionality to be built into Nodal Exec, partially reducing the Controller's workload, and would allow additional PASLIB routines and system diagnostics to be processor-resident.

5.4.4 *Input/Output.* The current system organization is inadequate for loading data to and from the Array in a timely manner. Although this is due in part to slow Controller software, a better solution would be to install one or more disk controllers as stations on the global bus, and allow processors to initiate their own I/O requests on direct access files. The disk(s) connected to the global bus should also be accessible to programs running on the Controller, using either the global bus or a separate path.

5.5 Recommendations for Further Work

Several areas of the system software need additional work. Perhaps the most pressing of these is Array I/O, primarily due to the shortcomings of data areas listed above. A much more convenient strategy from a programmer's point of view would be to allow read/write operations to files from within processes

executing on the Array. As a first step, PASLIB routines are being implemented to generate read/write requests to direct access files on secondary storage. This approach has the advantage of allowing data partitioning to be embedded within the logic of parallel programs by letting each process access only those specific records which it needs. Given the current system architecture, the I/O requests will have to be serviced by the Controller, and this will be slow. However, the approach can easily be adapted if a peripheral subsystem is someday attached directly to the global bus.

A natural extension of the direct access concept is to store problem data in a database, in which case read/write requests from processes become queries and updates. The requirements are similar to those for multi-user database systems, except that performance demands are likely to be greater. A dedicated database backend attached to the global bus could be helpful in this regard.

Another area for further work is programming language support, which is needed to tie together the concepts of the FEM system software into a more cohesive, easier-to-use package. The programming task could be considerably simplified by integrating the functions performed by FACS, SCI, and PASLIB into a Pascal-like language.

At least one more version of Nodal Exec and PASLIB should be developed to optimize the communication software and make other improvements. In particular, the fundamental performance limitations of the current communication strategy need to be determined to decide if alternative techniques are worth pursuing.

In addition, a need has been demonstrated for another type of send call, *SENDSET*. The SENDSET routine would transmit data to a subset (more than one, but less than all) of a process' neighbors, using the simultaneous output capability of the local links wherever possible.

Support for global bus broadcast should also be added to the communication software. Originally, broadcasting was only thought to be needed by the Controller, so no provisions were made in Nodal Exec to allow broadcasts initiated within the Array. However, more recent applications have shown a need for this capability, which can be interpreted as a SENDALL to all other processors in a parallel program.

Installation of the sum/max circuit would allow evaluation of its efficacy for global computations in parallel conjugate gradient and other programs. It also seems likely that sum/max would find important uses in applications which require multi-way comparisons and global decision-making.

6. CONCLUSION

The preceding sections have described system software which has been implemented for a mesh-connected MIMD array consisting of up to 36 processors. The system design has been outlined, and some of its more interesting features described. Several problem areas and recommendations for future work have also been presented. Although originally intended for use in the limited context of structural engineering analysis, the system has become increasingly general-purpose and could be expected to support a wide range of applications. It is presently in daily use on an 8-processor machine, providing support for research in parallel numerical methods and computational structural mechanics.

Based on experience to date, no inherent obstacles are seen to creating system software which could allow effective use of MIMD arrays consisting of up to a few hundred processors. Extrapolation to much larger numbers of processors (thousands and beyond) will require a somewhat different approach to processor control than that adopted here. The important problem of input/output, long neglected by designers of parallel architectures, will also have to be addressed. Nevertheless, it is expected that workable solutions can be found to most, if not all, of the problems which have been encountered. Some problems will need to be addressed at the hardware architecture level, others at the system software level, and still others at the numerical algorithm and problem formulation levels. Much work remains to be done, but the mesh-connected array appears to be a viable structure for parallel computing.

REFERENCES

1. Adams, L.; and Crockett, T.: Modeling Algorithm Execution Time on Processor Arrays. *Computer, 17,* 7 (July 1984), 38-43.

2. Adams, L.; and Voigt, R.: Design, Development and Use of the Finite Element Machine. NASA CR-172250, October 1983.

3. Andrews, G.; and Schneider, F.: Concepts and Notations for Concurrent Programming. *ACM Computing Surveys 15,* 1 (Mar. 1983), 3-43.

4. Baudet, G.: Asynchronous Iterative Methods for Multiprocessors. *Journal of the ACM 25,* 2 (Apr. 1978), 226-244.

5. Blum, B.: The Life Cycle – A Debate over Alternate Models. *ACM SIGSOFT Software Engineering Notes 7,* 4 (Oct. 1982), 18-20.

6. Bokhari, S.: On the Mapping Problem. *IEEE Transactions on Computers C-30,* 3 (Mar. 1981), 207-214.

7. Chazan, D.; and Miranker, W.: Chaotic Relaxation. *Linear Algebra and Applications 2,* 2 (Apr. 1969), 199-222.

8. Crockett, T.: PASLIB Programmer's Guide for the Finite Element Machine (Revision 2.1-A). NASA CR-172281, April 1984.

9. *DX10 Operating System, Vols.I–VI,* Texas Instruments Incorporated, Digital Systems Group, Austin, Tex., September 1982.

10. Garey, M.; and Johnson, D.: *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

11. Hoshino, T.; Kawai, T.; Shirakawa, T.; Higashino, J.; Yamaoka, A.; Ito, H.; Sato, T.; and Sawada, K.: PACS: A Parallel Microprocessor Array for Scientific Calculations. *ACM Transactions on Computer Systems 1,* 3 (Aug. 1983), 195-221.

12. Hoshino, T.; Shirakawa, T.; Kamimura, T.; Kageyama, T.; Takenouchi, K.; Abe, H.; Sekiguchi, S.; Oyanagi, Y.; and Kawai, T.: Highly Parallel Array "PAX" for Wide Scientific Applications. *Proceedings of the 1983 International Conference on Parallel Processing* (Aug. 1983), 95-105.

13. Jordan, H.: A Special Purpose Architecture for Finite Element Analysis. *Proceedings of the 1978 International Conference on Parallel Processing* (Aug. 1978), 263-266.

14. Jordan, H.; and Sawyer, P.: A Multi-Microprocessor System for Finite Element Structural Analysis. *Computers and Structures 10,* 1, 2 (1978), 21-29.

15. Jordan, H.; Scalabrin, M.; and Calvert, W.: A Comparison of Three Types of Multiprocessor Algorithms. *Proceedings of the 1979 International Conference on Parallel Processing* (Aug. 1979), 231-238.

16. Knott, J.: FEM Array Control Software User's Guide. NASA CR-172189, August 1983.

17. Knott, J.: A Performance Analysis of the PASLIB Version 2.1X SEND and RECV Routines on the Finite Element Machine. NASA CR-172205, August 1983.

18. Loendorf, D.: Advanced Computer Architecture for Engineering Analysis and Design. Ph.D. dissertation, Department of Aerospace Engineering, University of Michigan, Ann Arbor, 1983.

19. Preparata, F.; and Vuillemin, J.: The Cube-Connected Cycles: A Versatile Network for Parallel Computation. *Communications of the ACM 24,* 5 (May 1981), 300-309.

20. Read, R.; and Corneil, D.: The Graph Isomorphism Disease. *Journal of Graph Theory 1,* 4 (Winter 1977), 339-363.

21. Schwartz, J.: Ultracomputers. *ACM Transactions on Programming Languages and Systems 2,* 4 (Oct. 1980), 484-521.

22. Snyder, L.: Introduction to the Configurable, Highly Parallel Computer. *COMPUTER 15,* 1 (Jan. 1982), 47-56.

23. Stone, H.: Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers C-20,* 2 (Feb. 1971), 153-161.

24. Storaasli, O.; Peebles, S.; Crockett, T.; Knott, J.; and Adams, L.: The Finite Element Machine: An Experiment in Parallel Processing. *Research in Structural and Solid Mechanics - 1982,* NASA CP-2245 (Oct. 1982), 201-217.

25. Strang, G.; and Fix, G.: *An Analysis of the Finite Element Method.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-3870		

4. Title and Subtitle	5. Report Date
System Software for the Finite Element Machine	February 1985
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Thomas W. Crockett and Judson D. Knott	
9. Performing Organization Name and Address	10. Work Unit No.
Kentron International, Inc. Aerospace Technologies Division 3221 N. Armistead Ave. Hampton, VA 23666	11. Contract or Grant No. NAS1-16000
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered Contractor Report
National Aeronautics and Space Administration Washington, DC 20546	14. Sponsoring Agency Code 505-37-33-01

15. Supplementary Notes

Langley Technical Monitor: W. Jefferson Stroud

16. Abstract

The Finite Element Machine is an experimental parallel computer developed at Langley Research Center to investigate the application of concurrent processing to structural engineering analysis. This report describes system-level software which has been developed to facilitate use of the machine by applications researchers. The overall software design is outlined, and several important parallel processing issues are discussed in detail, including processor management, communication, synchronization, and input/output. Based on experience using the system, the hardware architecture and software design are critiqued, and areas for further work are suggested.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing System software Operating systems Finite Element Machine	Unclassified - Unlimited Subject Category - 62

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	30	A03

* For sale by the National Technical Information Service, Springfield, Virginia 22161

National Aeronautics and
Space Administration

Washington, D.C.
20546

Official Business

Penalty for Private Use, $300

THIRD-CLASS BULK RATE

LANGLEY RESEARCH CENTER
3 1176 01309 4439

Postage and F[illegible]
National Aeronautics and
Space Administration
NASA-451

NASA

POSTMASTER: If Undeliverable (Section 158 Postal Manual) Do Not Return