NASA Contractor Report 158918

ICASE

A SPECIAL PURPOSE ARCHITECTURE FOR FINITE ELEMENT ANALYSIS

Harry F. Jordan

(NASA-CR-158918) A SPECIAL PURPOSE ARCHITECTURE FOR FINITE ELEMENT ANALYSIS Final Report (Colspan, Inc.) 32 p HC A03/MF A01 CSCL 12A

N79-18645

Unclas G3/64 17796

Report Number 78-9

March 29, 1978

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia

Operated by the

UNIVERSITIES SPACE USRA RESEARCH ASSOCIATION

A SPECIAL PURPOSE ARCHITECTURE FOR FINITE ELEMENT ANALYSIS

Harry F. Jordan*

ABSTRACT

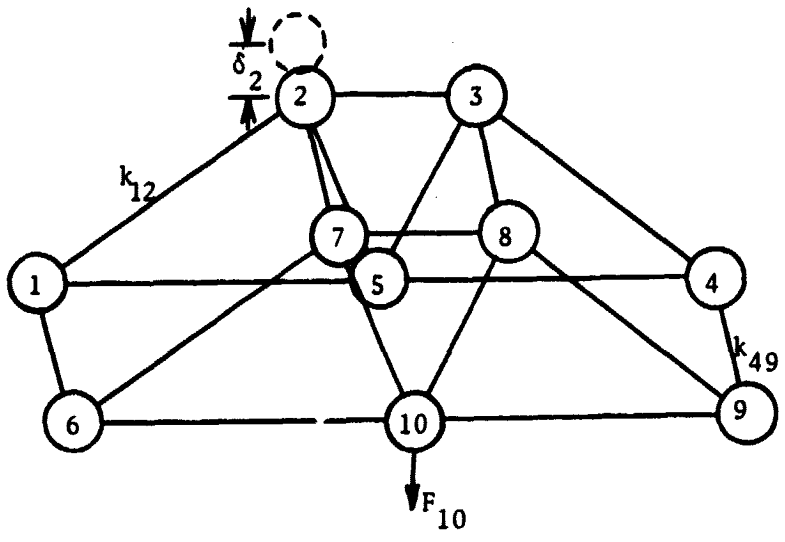
The analysis of aerospace structures by the finite element method consumes considerable computer time. The cost of this resource and the designer's desire to have rapid feedback concerning such questions as the effect of a change in loading of the structure or in a parameter of some structural material have led to the design of a special purpose parallel computing system for finite element analysis. As a special purpose computer, the architecture of this finite element computer is closely tied to computational aspects of the particular problem. This paper will relate various aspects of an MIMD array of microprocessors to the requirements of the class of finite element analysis problems which it is intended to solve.

*Electrical Engineering Department, University of Colorado, Boulder, CO

This paper was prepared as a result of work performed under NASA Contract Number NAS1-14101 at ICASE, NASA Langley Research Center, Hampton, VA 23665.

Introduction

The finite element method [1] reduces a structure to a set of discrete nodes joined together by idealized elements. The elements model the relationship between a force on node i and a displacement of node j by a stiffness coefficient k_{ij}. Rotational displacements and moment type forces may be included but the character of the problem is apparent if only translation forces and displacements are considered. The stiffness coefficients form a stiffness matrix K which relates the vector F of forces at each node to the vector δ of nodal displacements by the linear equation $F = K\delta$. In general F and δ involve several degrees of freedom, corresponding to rotational and translational movement in three dimensions, but we can restrict the discussion to one degree without losing generality. Figure 1 shows a very simple structure consisting of 10 nodes which might be considered to be connected by bar or beam elements only. It is also possible that a group of nodes and their stiffness coefficients arise from applying the finite element method to a more complex element, such as a plate.



$F = K\delta$

Figure 1: A Simple Finite Element Model of a Structure.

The computations of a finite element method solution may be divided into two parts:

1) Formation of the stiffness matrix K and

2) Solution of the linear equations for a given load case (value of F).

In forming the stiffness matrix, the elements used must be reduced to a small set of discrete nodes and stiffness coefficients. This is done using the geometry of the element and constants describing the particular material. The result is usually a set of node coordinates and a stiffness submatrix in some local coordinate system. The total stiffness matrix K is found by performing local to global coordinate transformations on each submatrix describing an element and adding in its contribution to the overall K matrix. It can be shown that the resulting K matrix is symmetric and sparse. The number of nonzero coefficients in a given row (or column) does not grow with the number of elements in the structure but depends only on the complexity of individual elements and how many of them are connected at a single point.

The sparseness of the K matrix is one aspect of the finite element problem having a strong influence on the architecture of a special purpose finite element computer. Of course, solving the linear equations by forming the inverse of K causes the zero elements to fill in with nonzero values. Iterative solution methods, however, work directly with the K matrix and thus avoid fill in. A test case with 63 nodes had a maximum of 12 nonzero coefficients for any node. Call a node j a neighbor of i if $k_{ij} \neq 0$ then Figure 2 shows the percentage of nodes having n or fewer neighbors. Clearly the predominant number of nodes have very few neighbors. Almost 70% have 5 or fewer.

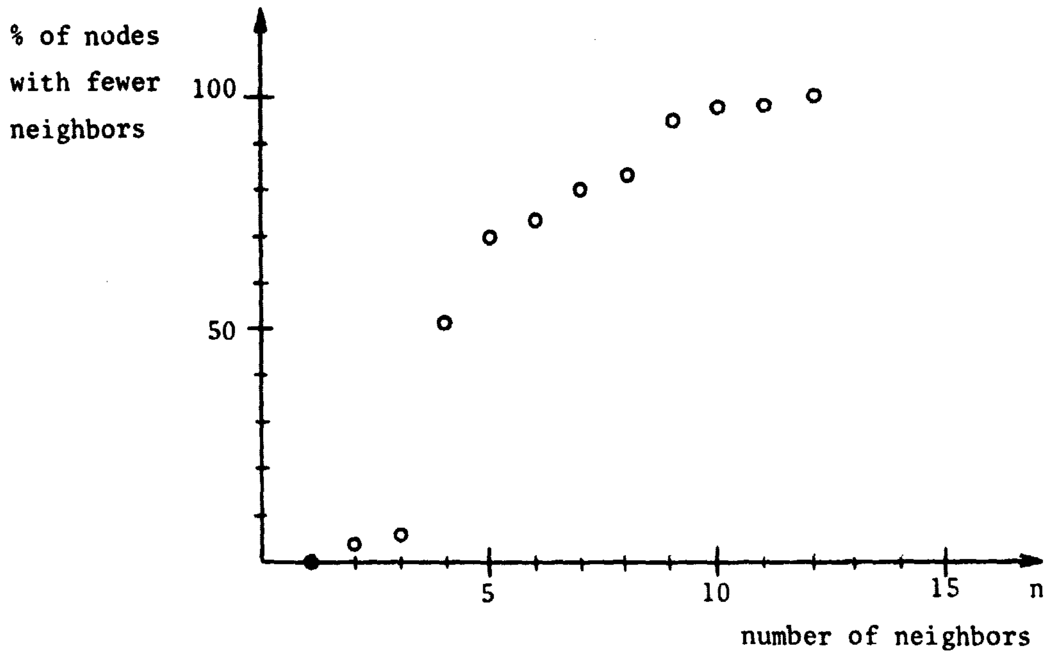


Figure 2: Number of Neighbors per Node in a Test Case.

Another aspect of finite element analysis on which the architecture will depend is the designer's tendency to do many solutions involving the same structure (or at least the same structural topology). He may vary material constants, element dimensions or loads without changing the topology defined by the nonzero elements of the K matrix. This implies that a fair amount of computational effort in transforming the problem topology to fit a given computer structure may be profitably traded for faster solution time with the fixed topology. Thus two of the guiding principles in the development of a finite element computer architecture are the limited number of neighbors per node in a finite element model and the willingness to use more problem set up time for a new model in order to gain faster response when the model topology is fixed. Parallel computation is a chief method of reducing response time and the limited number of neighbors per node holds out hope that the major obstacle of communication between processing elements can be overcome in this case.

To approach the introduction of parallelism into the solution of a finite element problem we use the fact that methods can be developed for computing the stiffness matrix and for solving the resulting equations which partition the work into portions associated with each node of the structure. Data is shared among the nodal subtasks in a way isomorphic to the interconnection of nodes by elements of the structure. The subtasks are not identical but depend upon the connectivity of the particular node and the types of the finite elements joined at that node. This is in contrast to the identical subcomputations performed by other proposed special purpose processor arrays such as the Navier-Stokes Computer [2]. Introducing parallelism by running nodal subtasks concurrently implies a computer architecture of the multiple instruction stream, multiple data stream (MIMD) type [3]. Even though a general parameterized program might be written which subsumed the computations necessary at every node, loops would be performed a different number of times at each node and major subcomputations might be skipped entirely for nodes with simple structure. A single instruction stream architecture would thus be inefficient and hard to synchronize if the work is partitioned into subtasks according to the physical topology of the structure.

This MIMD approach is consistent with current single chip microprocessor technology since a group of loosely coupled but independent microprocessors with local instruction memory can be expected to be less expensive than a network for distributing single instructions from a common stream to multiple processing elements. Another step taken to avoid the costs associated with distributed networks demanding high performance is to avoid the use of shared memory. The processors will communicate data by explicit transmissions from a source processor to a specific destination. (Broadcast is also possible.) Since all processors cooperate on a single job, there is only one input stream and one output stream. This implies that I/O should not be distributed but under control of a single sequential process, i.e. attached to a single processor.

PMS Level Description of the Architecture

The structure of the machine is adapted to the solution of finite element structural analysis problems by iterative schemes. The property of the solution method which influences the architecture is that the computations carried on by a single processor will be associated with a given node. We will think in terms of problems with on the order of 1000 nodes. With a correspondence between nodes and processors the topology of the physical node interconnection pattern will have to be modelled by the interconnection topology built into the hardware. The simplest processor interconnection which is general enough to model any topology is the single time-multiplexed bus connecting all processors. The aspect of the problem which rules out this approach is the short period, measured in bus transmission time units, between exchanges of information between processors. Permutation networks (crossbar switches or rearrangeable switching networks, RSN) provide for N simultaneous information transfers, where N is the number of processors, but must be switched frequently for one processor to communicate with several others during an iteration. Crossbars are easily switched, but are expensive in gate count (N^2), while RSNs are expensive in terms of time to set up a given permutation (order $N \log_2 N$ time units [4]). Since nodes in a structure modelled by finite elements are typically connected to a small number of other nodes, it is tempting to try to provide a network of local interconnections which may carry the interprocessor communications. Any network of local interconnections will have a fixed topology, however, so that the node interconnection topology of the finite element model must be mapped onto the processor interconnection topology. Such a mapping is not possible with an interconnection network growing linearly with the number of nodes and an arbitrary model topology.

The approach adopted in this design is that of a fixed interconnection pattern backed up by one (or more) time multiplexed bus for connections which do not fit this topology. The interconnection patterns to be investigated include 8 nearest neighbors on a square planar grid and cubic close packing in 3 dimensions, both with boundary nodes considered as neighbors of appropriate boundary nodes on the opposite side. From the point of view of hardware design costs the simplicity and regularity of this interconnection pattern reduces the number of long cables and the number of external connections to a given hardware module (circuit board, cabinet, etc.). From the point of view of structural analysis, most problems have a large component amenable to mapping onto a topologically regular grid in 3 dimensions or which can be developed onto a plane, leaving a small percentage of the nodes which must compete for use of the time multiplexed bus. Furthermore, the bus is likely to be used most often by nodes which connect to a large number of others, such as the vertex node in a conical structure. These nodes have a longer computation to do per iteration, thus reducing the average rate of bus usage. The interconnection pattern is schematically shown for 16 processors assuming 8 nearest neighbor planar topology to Figure 3. The bus is omitted for clarity.

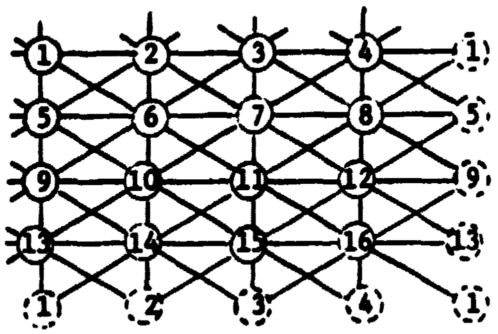


Figure 3: Fixed Interconnection Pattern for Processors.

The individual processors will be 16 bit, single chip microprocessors, with hardware multiply and divide, coupled to a read/write memory, RAM, for programs and data and a read-only memory, ROM, for the start up algorithm. The interconnecting paths appear to the processor as I/O ports (memory locations for memory-mapped I/O) and provide not only for data (and address in the case of the bus) but also for handshake control signals between source and destination. It is felt that the most appropriate way to control such a multiprocessor system is by using a control algorithm which can be distributed among the processors. Indeed, some aspects of control must be distributed in an MIMD machine and we provide some simple synchronization mechanisms, discussed below, to help implement such algorithms. However, at the current state of development of parallel control algorithms we prefer to provide the option of using a central control processor for the initial phases of development. We expect this processor to be important in initial startup of the array in doing I/O. Since a single problem will be running on the system, the I/O is best handled by one process running on one processor. We take the number of processors to be $N = 2^{2k}$ and all indices are taken modulo $N = 2^k$. Where order of magnitude is important we will take $k = 5$, $N = 1024$. An abbreviated PMS [5] diagram for the processor array is shown in Figure 4. A diagram for one processor is given in Figure 5.

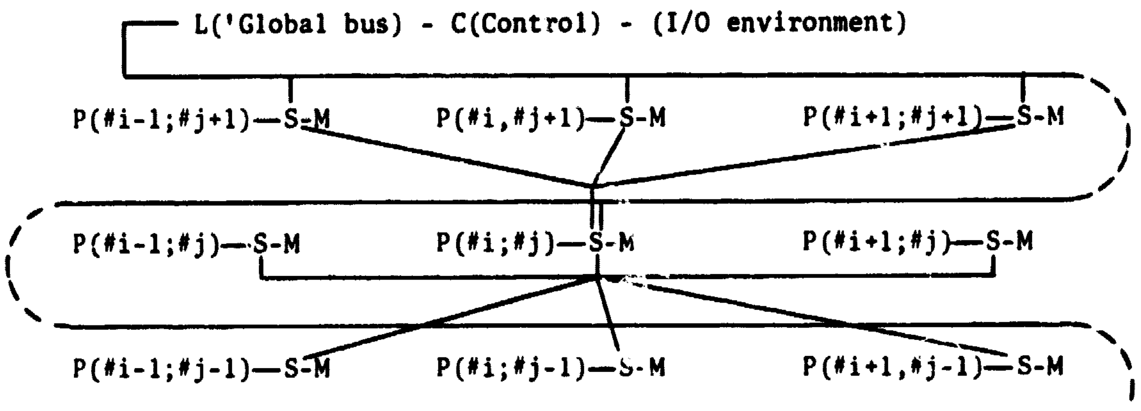


Figure 4: Partial PMS Diagram for the Array

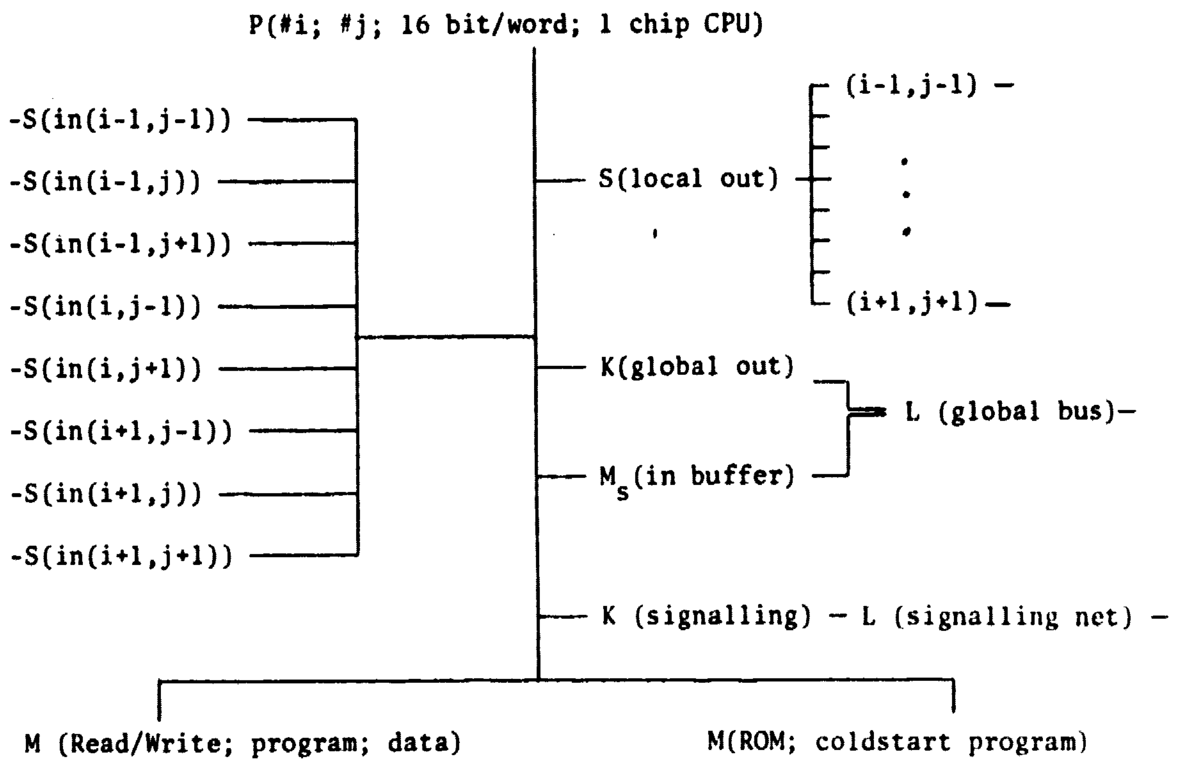


Figure 5: PMS Diagram for One Processor.

Influence of the Architecture on the Solution Algorithm

To see how the proposed architecture would be used for an iterative solution to a finite element analysis problem, consider the calculation done by a processor associated with node j in one step of the Jacobi method for iterative solution of the set of linear equations, $K\delta = F$, where K is the stiffness matrix, δ is the dis`lacement vector and F the load vector. Note that the architecture is not limited to this method, but it serves to illustrate the ideas in a simple context. We do not consider formation of the stiffness matrix and assume that processors have already been assigned to nodes for the structure in a way such that most physical node interactions occur between locally connected processors. The computation performed by the processor for node j is

$$\delta_j = \frac{F_j}{k_{jj}} - \sum_{i\neq j} \frac{k_{ji}}{k_{jj}} \delta_i \quad ,$$

where the sum ranges over nodes i for which $k_{ji} \neq 0$. We call such nodes physical neighbors of node j and call nodes assigned to processors with local connections to the processor for node j local neighbors of node j. The optimal condition is that in which all physical neighbors of node j can also be made local neighbors by proper assignment of processors to nodes. Since this is unrealizable in general, the global bus will need to be used for some of the physical neighbor interactions.

Each processor is assumed to be provided with a table of physical neighbor nodes giving their mapping onto the processor array and the route by which data travels to and from them. The local data paths are named by the compass points N,NE,E,SE,S,SW,W, and NW. The connections using the bus require that processor j know the processor number $P(i)$ of the physical neighbor. The receiver i of a bus transmission uses the number j of the source processor to determine a buffer register in its memory for data from processor j. The physical neighbor table might appear as in Figure 6. R(i) is the register in the memory associated with global bus input from node i. R(N), R(NE), etc. are local connection input registers. The algorithm performed by

Physical Neighbors of j

node number	route	coefficient
i_1	NE	$a_{jk_1} = k_{ji_1}/k_{jj}$
i_2	E	a_{ji_2}
i_3	S	.
i_4	NW	.
i_5	$R(i_5)$	.
i_6	$R(i_6)$	a_{ji_6}

Figure 6: Routing Table for Processor j.

processor P(j) would have the general form shown in Figure 7. Note that some synchronization must be provided so that an input register of P(j) associated with P(i) is not read before P(i) has filled it.

Since the long distance bus is a potential bottleneck, care should be taken that the transmission over this bus does not interfere any more than necessary with the progress of the transmitting and receiving processors. To allow overlapping of bus transmission delays with other computation, data transfer requests should be made as early as possible. Thus transfers should be initiated by the source processor as soon as the data is produced rather than

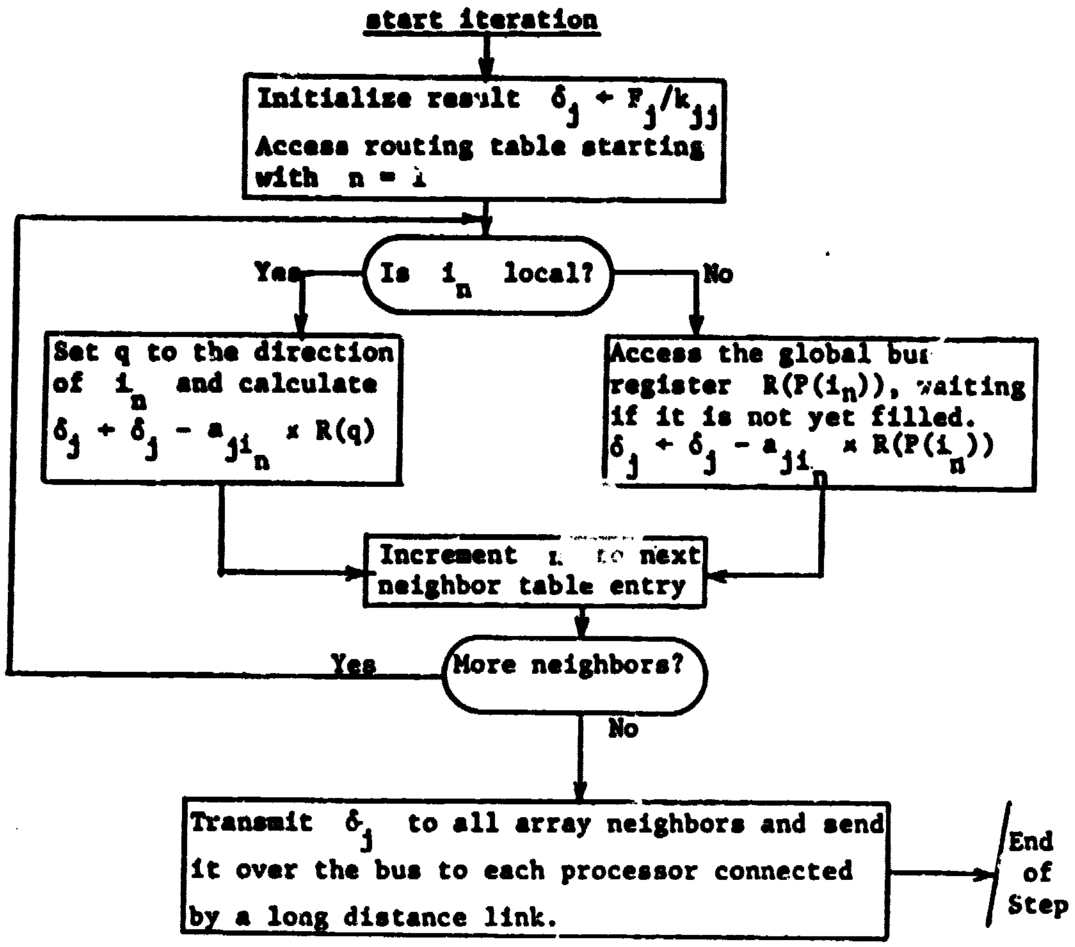

start iteration
Initialize result $\delta_j \leftarrow F_j/k_{jj}$ Access routing table starting with $n = 1$
Is i_n local?
Yes
No
Set q to the direction of i_n and calculate $\delta_j \leftarrow \delta_j - a_{ji_n} \times R(q)$
Access the global bus register $R(P(i_n))$, waiting if it is not yet filled. $\delta_j \leftarrow \delta_j - a_{ji_n} \times R(P(i_n))$
Increment n to next neighbor table entry
More neighbors?
Yes
No
Transmit δ_j to all array neighbors and send it over the bus to each processor connected by a long distance link.
End of Step

Figure 7: Iteration Step for Processor j.

by the destination processor when the data is required in order for the computation to proceed. A possible design of the bus communication system would then behave as follows:

1. The source processor would place a request for data transmission to a given bus input cell of a given destination processor by entering address and data into a FIFO buffer.

2. A hardware transmission controller would examine the FIFO and, if it was nonempty, would request the bus, wait for bus control and transmit the data to the appropriate address.

3. A reception controller would be responsible for monitoring the bus for its processor number and, when it was addressed, placing the data into a memory register associated with the source processor. Each word would have a full/empty flag which would be set to full by the receiver when data was stored into that cell. (See note 1.)

4. The receiving processor would access a particular cell in the memory associated with the global neighbor, waiting until the cell was marked full. On reading the data from the cell the processor would mark it empty.

The algorithms for the bus communication system in processor P are indicated by the flowcharts of Figure 8. The one level buffering and full/empty indicator should also be included in the local connection input registers making possible full data synchronization of an iteration step.

Note 1: If the destination register is still marked full from a previous reception the receiver must buffer the data until it is emptied by the processor. In the Jacobi iteration step, it is guaranteed that only one level of buffering is required. Let the source processor be i and the receiving processor j. Processor i writes $\delta_i^{(1)}$ into processor j and proceeds to compute $\delta_i^{(2)}$ which it also transmits to processor j. Assuming that processor j has not yet read $\delta_i^{(1)}$ this $\delta_i^{(2)}$ must be buffered. But processor i cannot continue to produce $\delta_i^{(3)}$ since this requires using $\delta_j^{(2)}$ which cannot be produced until processor j has emptied $\delta_i^{(1)}$.

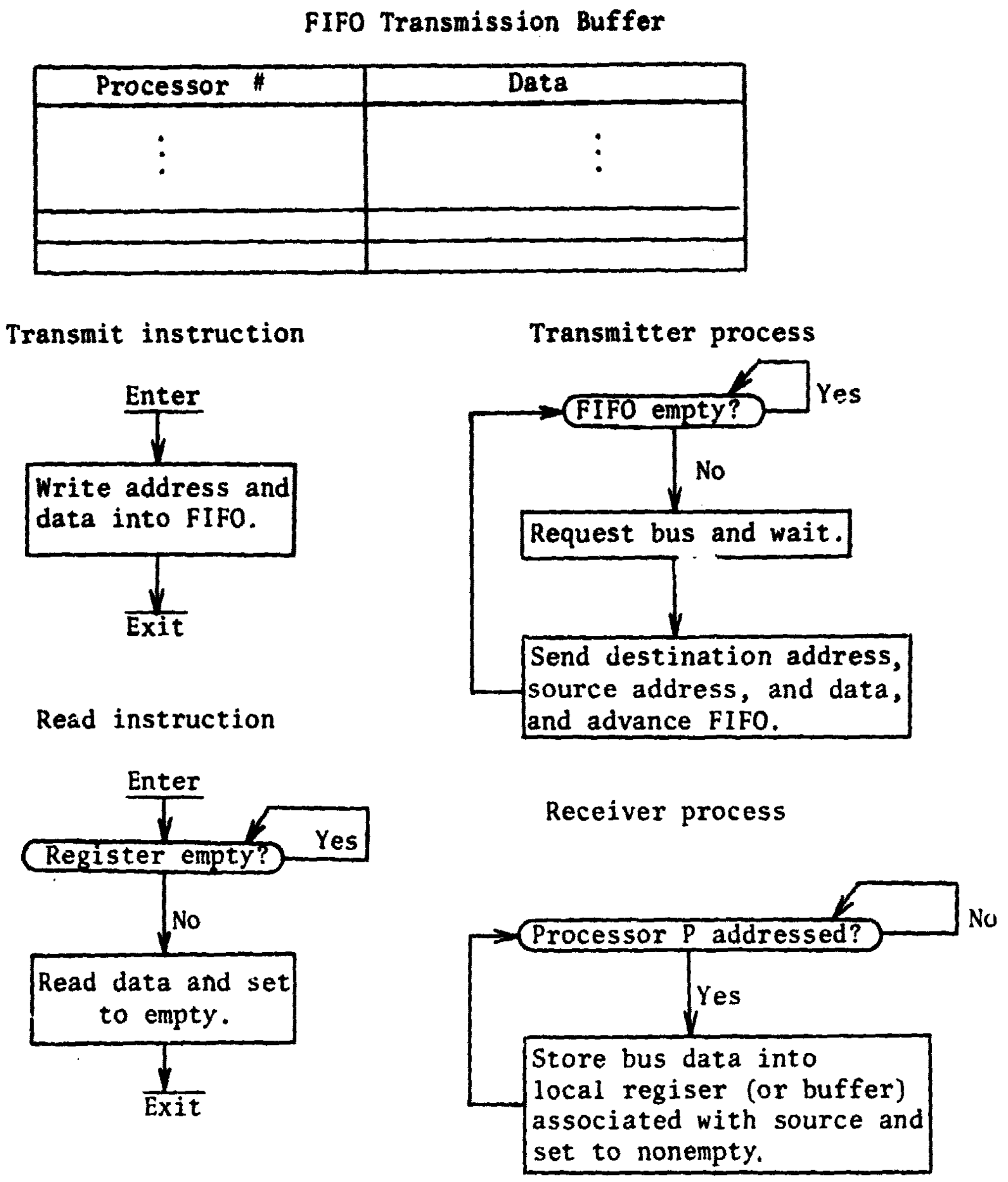

FIFO Transmission Buffer
Processor #
Data
Transmit instruction
Enter
Write address and data into FIFO.
Exit
Transmitter process
FIFO empty?
Yes
No
Request bus and wait.
Send destination address, source address, and data, and advance FIFO.
Read instruction
Enter
Register empty?
Yes
No
Read data and set to empty.
Exit
Receiver process
Processor P addressed?
No
Yes
Store bus data into local regiser (or buffer) associated with source and set to nonempty.

Figure 8: Bus Communication System.

Interprocessor Communication Primitives

Having considered all major system components we are ready to describe the system at the instruction set level. The instruction set of the nodal processors is assumed, but must be enhanced by communication and signalling operations. At the algorithm level the processor producing a result will broadcast this same result to all of its neighbors. The form of the broadcast operation at this level would be:

OUT (value, result id).

If result numbers are produced in a fixed order then the result identifier may be deleted. Of course, results for separate iterations are produced in a fixed order, but within any iteration several numbers may be produced, i.e. displacements for several degrees of freedom.

At the same level, a processor which requires a result from one of its neighbors must identify that neighbor. Including the identification of the neighbor and the character (degree of freedom, etc.) of the number required, the input primitive operation at the algorithm level would have the form:

IN (value, input id, neighbor).

Each processor (structural node in the finite element problem) will input k numbers in order to produce one result, where k is the number of neighboring nodes in the problem topology. This will be done $f \times k$ times per iteration where f is the number of degrees of freedom.

At the implementation level, neighbors are of two types: those with local connections to the reference processor (called local neighbors) and those only connected to the reference processor by the global bus (called global neighbors). At the algorithm level a neighbor is identified by an absolute node number where the numbering scheme refers to the problem topology. At the implementation level, physical neighbors are identified in two different ways depending on their local or global relationship to the reference processor. Local neighbors are identified by a relative designator, which for planar connectivity could be specified by a compass direction, North, Northwest, Southeast, East, etc. This relative designator will map directly into a reference processor input port address. A global neighbor would be referenced by a processor number determined by a hardware-fixed numbering scheme established by the processor interconnection topology. Of course, for any given reference processor, numbered i in the global numbering scheme, the global number of any local neighbor can be determined from i and the relative designator.

j = h(i, direction), where "direction" is the relative designator.

Because of the character of the iterative scheme, results for distinct iterations will be accessed sequentially by the processor using them. Thus no iteration number need be associated with a result number. The flow of results from source to destination can be viewed at the algorithm level as taking place via a group of FIFOs associated with the receiver. There is one conceptual FIFO associated with each neighbor in the problem topology and with each of the neighbor's degrees of freedom. The total number of conceptual FIFOs for processor i is $N_F = \sum_{\substack{\text{neighbors} \\ j \text{ of node } i}} f_j$, where f_j is the number of degrees of freedom of processor j. The number of degrees of freedom is 6 for a node at which all 3 translational and 3 rotational displacements in 3 dimensions are allowed. The nature of the parallel iterative scheme guarantees that each of these FIFOs need only have length 2. Data synchronization requires a receiving processor to hang up when attempting to read an empty FIFO. Providing the ability for a processor to wait for an information item (message) also provides hardware support for the highly message oriented type of operating system described by Brinch Hansen [6].

These conceptual FIFOs are impractical to implement in hardware as a result of their limited length and large number. If, however, a fixed order is adopted for producing numbers associated with different degress of freedom, then one FIFO could be associated with each neighbor, j, with a length on the order of $2 \times f_j$ floating point numbers. For the local neighbors, at least, it might be practical to implement such FIFOs in hardware. The FIFOs for all local neighbors should not be combined for two reasons:

1) Some locally connected processors may not be neighbors in the problem topology.

2) The varying nature of the iteration completion speeds would make it difficult to establish a sequential order on results from different neighbors. For global neighbors, numbers arrive over the same global bus from different source processors in random order (though the order of numbers from any given processor may be fixed by the degree of freedom to which it corresponds). Thus the manipulation of a FIFO for each global neighbor will probably be a software function if it is desirable.

Let us adopt the strategy of a single input FIFO associated with processor i for each of its local neighbors. Processor i is then responsible for determining the character of an input number from the order of arrival; thus a result id is not necessary. Figure 9 shows the local transmission scheme with 12 neighbors allowed to take into account the 3 dimensional case. Processor i places words for local neighbors into the local output register for simultaneous transmission to all local neighbors which are enabled. A non-empty input FIFO with its interrupt enable bit set will cause an interrupt of processor i whenever it is not empty. The non-empty status can also be tested directly. An input FIFO full condition must be reported to the processor writing into the FIFO and thus affects this processor's output operation. A set of primitive operations for local communication is given in Table 1.

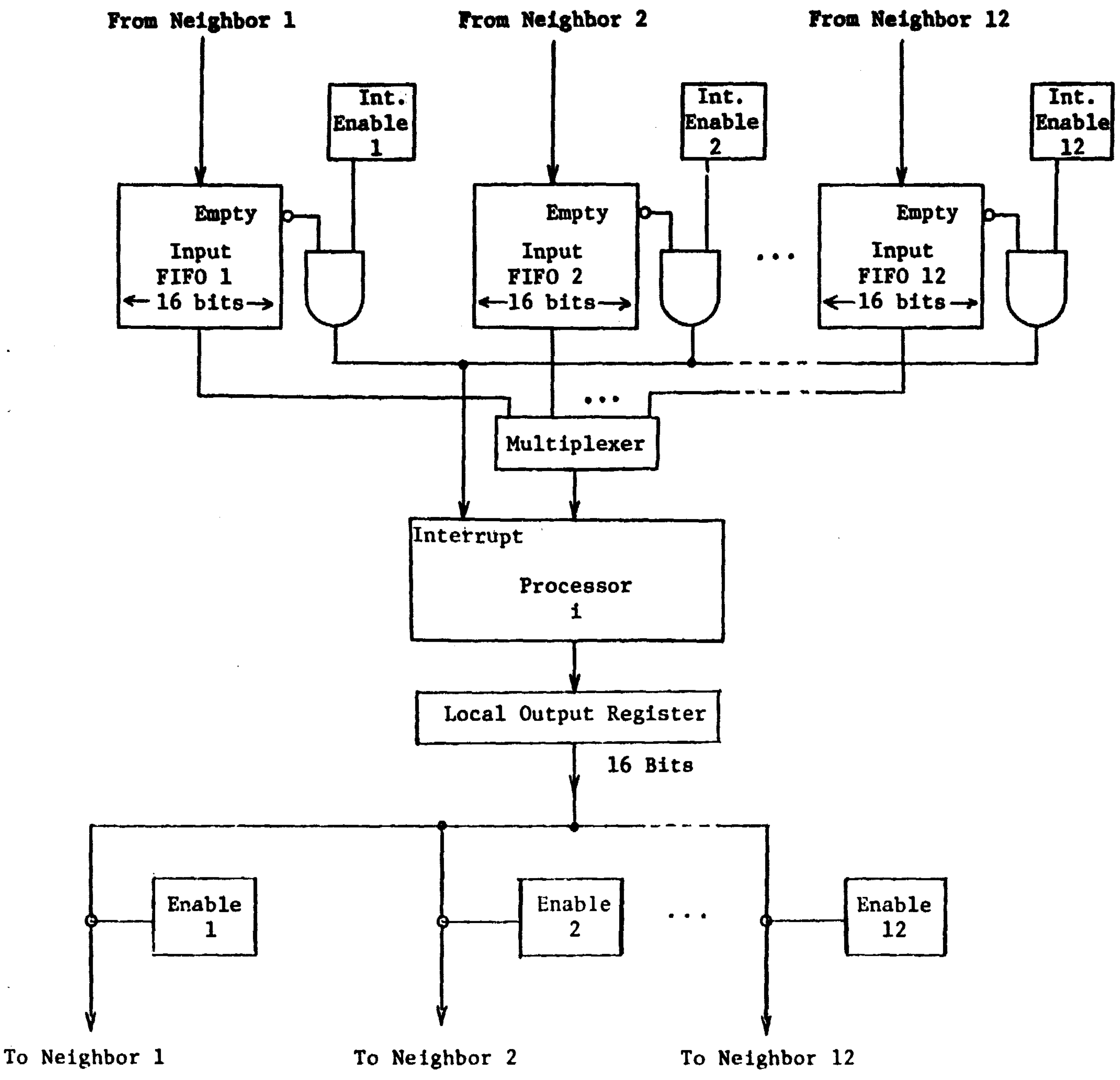


Figure 9. Conceptual Framework for Local Transmission

Operations:

Output (word)	- Broadcast the 16 bit word to all enabled neighbor processors.
Enable (j)	- Set the flag to enable output to local neighbor j.
Disable (j)	- Disable output to local neighbor j.
Input (j, word)	- Input one 16 bit word from the input FIFO for neighbor j.
Interrupt enable (j)	- Enable interrupt when input FIFO j becomes nonempty.
Interrupt disable (j)	- Disable interrupts from input FIFO j.

Tests:

Output busy?	- Is this processor's input FIFO in any enabled neighbor full?
Input ready (j)?	- Is input FIFO j nonempty?

Table 1. Local Communication Primitive Operations

For global transmissions we will also adopt the policy of having a receiving processor, say k, determine the character of a word received from processor i by the order in which i transmits words. Any FIFO buffering must be done by software since it is impractical to supply a hardware FIFO for every other processor in the system and only the software can determine which connections are active in a given problem. Thus words coming in over the global bus must be associated with a source processor number i and will interrupt processor k which will determine what to do with them (place them in a software FIFO associated with global neighbor i). The FIFO output queue prevents a sending processor from hanging up on an output request until as many as 1023 bus transactions are completed. Processor i records a word w and destination processor k in the FIFO and a hardware controller requests the global bus and performs the (w,k) transmission when it is at the head of the queue and the bus is available. The transmission (w,k) performed by processor i is received by processor k as the pair (w,i). Thus information transmitted on the global bus takes the form of information packets (source, word, destination) where "destination" plays the role of an address and "source" and "word" are the data received. A diagram of the global bus interface is shown in Figure 10. The global bus is provided with an input

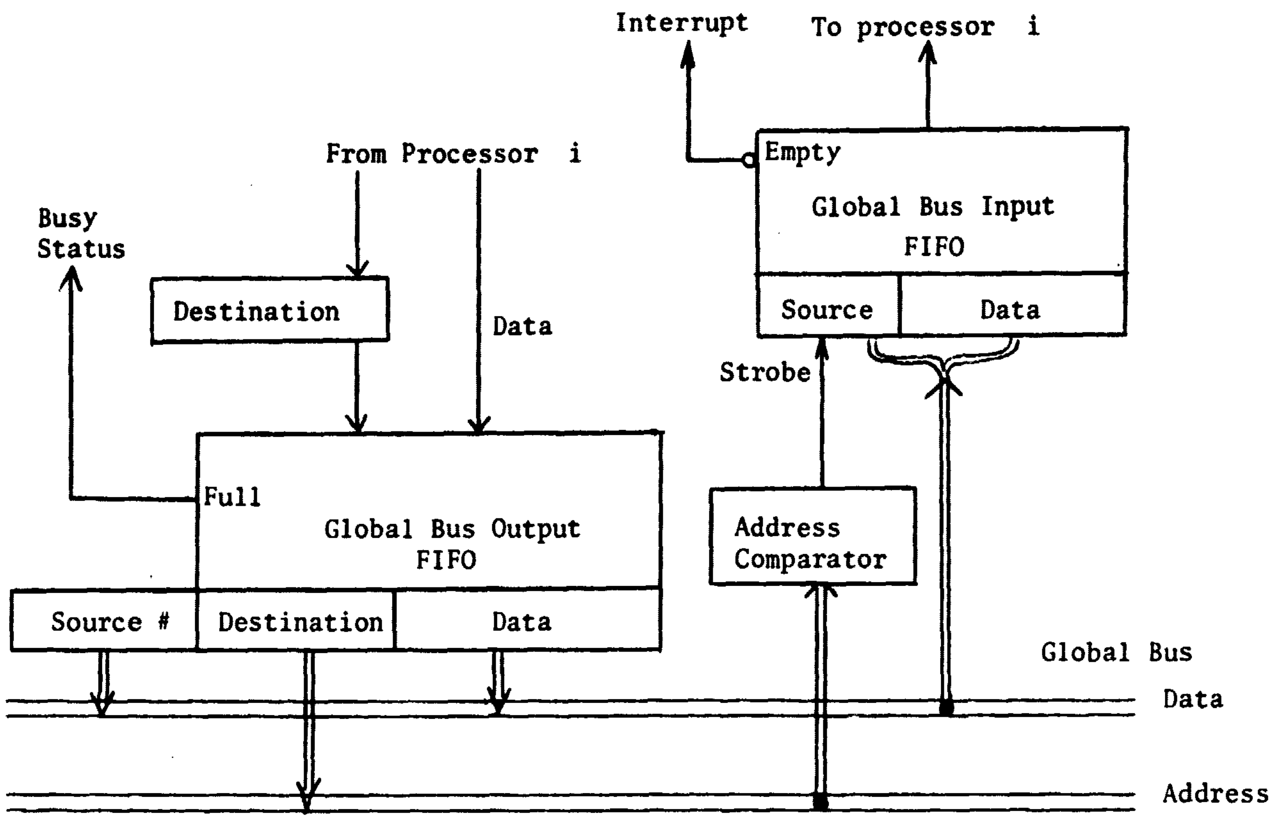


Figure 10. Outline of Global Bus Interface

FIFO because processor i could receive transmissions (from different sources) as often as once every bus cycle in the worst case. This is faster than the data can be handled by software.

The primitive operations and tests associated with global bus communication are listed in Table 2. A special destination number is provided for broadcast mode transmissions.

Operations:	
Set destination (i)	- Write processor number (i) into the global bus destination register.
Send word (w)	- Write w and destination register contents to FIFO initiating transmission.
Set broadcast	- Set the global bus destination register to the special broadcast address.
Read source (s)	- Read the source processor number at the head of the FIFO.
Read data (w)	- Read the data word and advance the FIFO to the next source and data pair.
Receive broadcast	- Set address comparator to respond to broadcast address.
Ignore broadcast	- Set address comparator to respond only to this processor's number.
Test:	
Global input ready?	- Is there a (source, data) pair in the FIFO?

Table 2: Global Bus Communication Primitive

The broadcast mode is primarily intended for loading the same program or initial data into all, or almost all, processors. It will thus probably be used only with the control processor as a source. It is easy, however, to give broadcast capability to all processors. Transmission of a block of words to a single destination is done by following one Set destination operation by several Send word operations. Thus the software can approximate a block transmission operation without complicating the bus structure.

Two timing problems were mentioned in connection with transmission over the global bus:

1) A processor sending information may have to wait as long as 1023 bus cycles before its request for the bus is granted.

2) Packets of information may arrive at a destination processor as fast as once per bus cycle.

The problem of a source processor waiting to obtain the bus can be solved by providing a FIFO buffer for output to the global bus. Transmission requests, consisting of destination processor number and word, are posted in the FIFO and a hardware controller transmits a word whenever the bus becomes available. The problem of a destination processor not being able to handle information packets as fast as they arrive in the worst case could be solved by having the destination processor refuse the transmission, in which case the source processor relinquishes the bus leaving the information packet in its FIFO.

The problem with allowing the destination processor to refuse a transmission is that it introduces a non-zero probability that a processor attempting to transmit will be refused arbitrarily often. Thus a particular kind of deadlock becomes possible in which the system as a whole progresses but one or more processors are blocked. This sort of deadlock is not possible if the transmitting processor holds the bus until the transmission is complete. Bus holding, however, allows for complete hang up of the system as a result of one processor's failure to empty its global bus input register.

In either case the prompt emptying of global bus input registers by receiving processors is essential to efficient use of the bus. Thus the interrupt associated with a full input buffer should have very high priority.

Perhaps it should be non-maskable. The probability of deadlock in either case (bus holding by the source or transmission refusal by the destination) can be statistically reduced by adding a global bus input FIFO. The possibility of deadlock still exists, however, since the FIFO may become full.

Cooperative Signaling and Synchronization

Several aspects of the algorithms to be executed by the finite element machine require synchronization of the nodal processors. Consider, for example, the transition from formation of the stiffness matrix to solution of the linear equations defined by it. No processor should proceed to the solution process until all are finished with the stiffness matrix formation. Let us call such a synchronization a barrier synchronization. Such an operation can be performed by designating a control processor and requiring each processor to report arrival at the barrier to the control processor by way of the global bus. The control processor waits until all reports are in and then sends a broadcast message for all to pass the barrier. For 1024 processors this requires 1025 bus transmissions to do barrier synchronization. As bus transmissions are sequential, this is quite time consuming. Faster synchronization might be performed using a reporting network built from local interconnections, but, since software is involved in rebroadcasting a locally received message, longer basic transmission times and nontrivial control software are involved.

It is proposed to address this and similar synchronization problems by providing hardware support, other than the local and global communication schemes, for cooperative signaling. The cooperative signaling interface to one processor is shown in Figure 11. It consists of a set of k identical interfaces forming k independent networks over the set of all processors. That is, the ith signaling flag interfaces on all processors form the ith signaling flag network. The flag is the central item of the ith interface

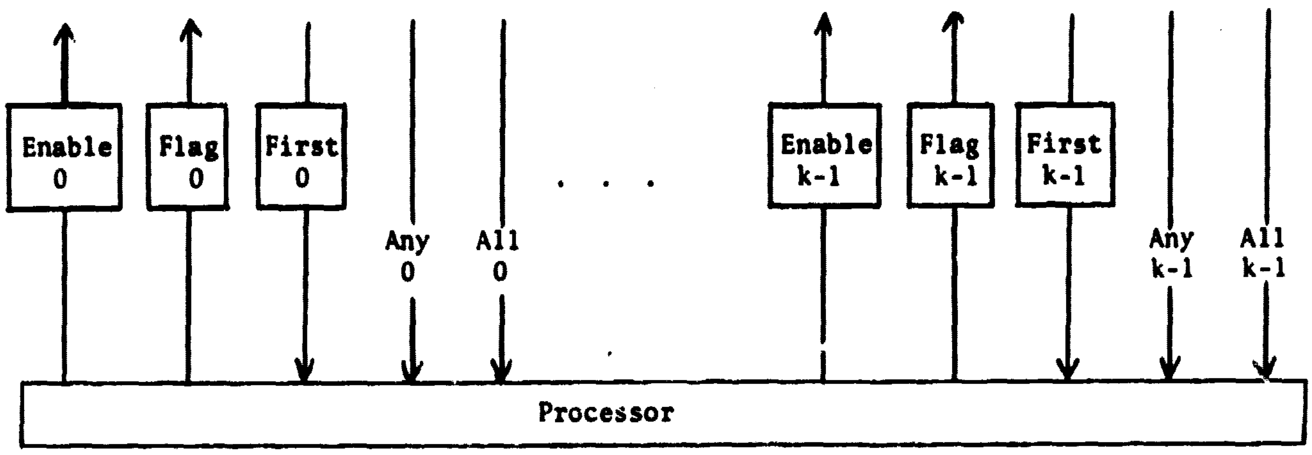


Figure 11: Cooperative Signaling Interface.

Enable serves to connect this processor's Flag into the ith signaling network. Any and All are cooperatively determined signals which tell each processor if any enabled Flag is set and if all enabled Flags are set, respectively. To support unique selection, each time a Flag is set a priority network decides if the Any line was initially zero and this is the leftmost (in an arbitrary ordering) processor to set its flag at this time. The unique "first" processor to set its flag on a given signaling cycle has its First bit set. First is cleared when the flag is cleared. The primitive signaling operations available to each processor are summarized in Table 3.

Connect (i)	- Set the ith Enable bit.
Disconnect (i)	- Clear the ith Enable bit.
Set (i)	- Set signaling Flag i.
Clear (i)	- Clear signaling Flag i.
Any (i)?	- Is the ith Flag set in any connected processor?
All (i)?	- Are the ith Flags set in all connected processors?
First (i)?	- Did the last Set (i) cause First to be set?

Table 3: Cooperative Signaling Primitives.

With these signaling primitives the barrier synchronization can be performed. One processor is chosen as controller and all are connected to two signaling flags. Flag 0 is the barrier and is initialized to one in the controller and to zero in all other processors. Flag 1 is the report flag and satisfies the same initial conditions. Upon reaching the barrier, each processor sets its report flag and performs the Any test on the barrier, waiting as long as Any is true. The controller repeatedly performs the All test on the report flag and clears the barrier flag when this test succeeds. Using the unique selection feature the barrier synchronization can be accomplished with an identical program in all processors as shown in Figure 12.

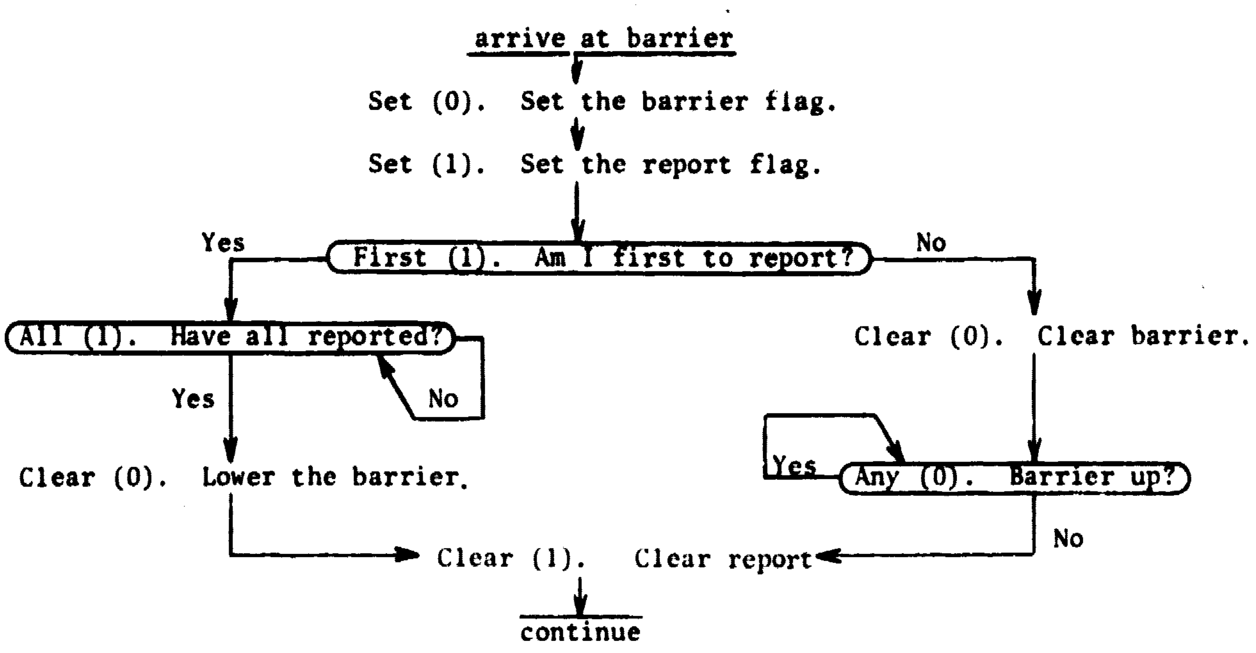

arrive at barrier
Set (0). Set the barrier flag.
Set (1). Set the report flag.
First (1). Am I first to report?
Yes
No
All (1). Have all reported?
Yes
No
Clear (0). Lower the barrier.
Clear (0). Clear barrier.
Any (0). Barrier up?
Yes
No
Clear (1). Clear report
continue

Figure 12: Barrier Synchronization with Unspecified Controller.

Under the reasonable assumption that each primitive operation requires one instruction time, an 8 instruction loop can be used to fully synchronize a multiple item broadcast transmission as would be used to load a common program into all processors.

The number of signalling flags required to perform the synchronization needed in the finite element machine remains to be determined. It appears that many useful synchronizations can be performed in a time independent of the number of processors with only a few flags. For example, two flags are sufficient to imitate the 3 wire handshake of the IEEE standard interface bus [7]. No active synchronization mechanism such as semaphores or critical section support [8] is proposed for this machine. Such mechanisms are most helpful in arbitrating a shared facility. There is no shared facility in the present architecture other than the global bus which is handled by its own arbiter.

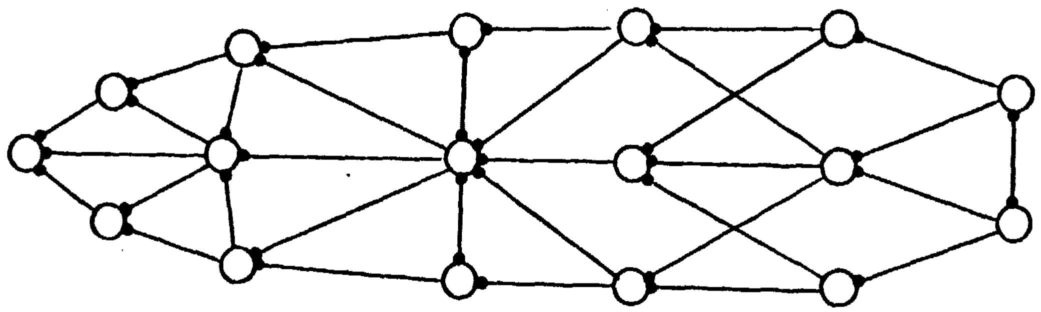
Performance of the Finite Element Machine

It is now important to evaluate the performance of the architecture described above over the problem class for which it was designed. The following comments give an indication of the direction of the work to be done in this area. A few of the crucial performance issues are:

1. The development of effective procedures for mapping the finite element model topology onto the array topology with minimal use of the bus.
2. Effective overlapping of time multiplexed bus transmissions with computation.
3. Effectiveness of global control of the array and global I/O operations.
4. Speed of the linear equation solution,which reduces to rate of convergence for an iterative scheme.

Of particular concern is the relatively slow convergence rate which can be expected for the Jacobi iteration (see e.g. [9]) since a Jacobi method seems most natural for the type of microprocessor array described.

The fact that a receiving processor waits for its input registers (local and global) to be marked full before performing the next iteration step means that data synchronization can be used to get away from Jacobi iteration to a "wave type" interation scheme which is similar to Gauss Seidel in that the latest available information at each node is used in the computation (see e.g. [9]). The iteration could be started with an initial δ value guess, but the source registers for only a few processors marked full. These processors would proceed to calculate results and fill the registers of their neighbors. By correctly arranging the initial full/empty marking the updated values could be made to proceed in waves through the structure. In Figure 13, input cells initially set full are indicated by black dots at each node circle.



Figu e 13: Initia' ıditions for Iteration

Computation starts at the vertex node, since only this node has all input cells marked, and, when complete, the result is sent to the 3 nearest neighbors to the right, marking their corresponding input cells as full. Thus the new values propagate from left to right through the structure in a wave-like fashion. The vertex node can start its second iteration cycle as soon as all 3 neighbors have completed their first iteration.

Some rough timing considerations for the execution of one iteration step, as outlined in the flowchart of Fig. 7, can be made. Each complete iteration step involves some results from processors with local connections to processor j (left hand branch in flowchart) and some results sent over the bus (right hand branch). We assume that all local connections are listed first in the routing table of Fig. 6. We can then view processor j as proceeding from left to right computing an expression of the form:

$$\delta_j = C_j - a_1 R(q_1) - a_2 R(q_2) - \dots - a_f R(q_f)$$

$$- b_1 R(P_1) - b_2 R(P_2) - \dots - b_g R(P_g).$$

The number of local connections for processor j is f (or f_j) and the number of global, or bussed, connections is g_j. Let t(m) be the time required to compute the constant term through the mth product term in the expression. Then processor j has $t(f_j)$ time units at the beginning of each iteration cycle during which it requires no results sent via the global bus. If $f_{min} = \min_j(f_j)$ is the minimum number of local connections over all processors, then $t(f_{min})$ time units at the beginning of each iteration step are available for bus transmissions with no possibility of delaying any processor. In the worst case, it is possible that the last bus result transmitted is the first one needed, though this case is not likely. In this case, one or more processors would have to wait for completion of all bus activity before proceeding to calculate the part of the expression depending on global results.

To explore the timing associated with the hardware of the proposed system, we assume that each processor in the array consists of a 16 bit, single chip microprocessor (for example the TMS 9900 from Texas Instruments) enhanced by a hardware floating point unit. It is proposed to allow for 32 bit floating point operations although the precision requirements are not yet firmly established. The average instruction time for the 9900 is about 6 microsec and we assume that the multiply-subtract operation required for each R value can also be performed in 6 microsec by the added hardware. With these assumptions a sketch of the code needed to accumulate terms in the expression and optimistic timing estimates would be as shown in Figure 14.

```
      Get constant term.                  6 microsec
 ┌──►Reference routing table. ┐
 │    Get multiplier.         │
 │    Get result value.       │
 │    Multiply-subtract.      ├           36 microsec
 │    Increment index.        │
 └─── Test for completion.    ┘
```

Time to accumulate constant and m product terms:

$t(m) \simeq 6 + 36\ m$ microsec

If floating point operations are done by software then:

$t(m) \simeq 12 + 250\ m$ microsec

Figure 14: Program Timing for an Iteration Step

To see how the computation time interacts with bus transmission time, we must consider the speed of bus transfers. Contention for use of the bus among N processors can be done by a request tree technique so that the total delay involved in determining the next processor to get control of the bus is $\log_2 N$ times the network gate delay associated with a single processor. Assuming 50 nanosec delay for gates in one processor and the transmission line to the next processor in the network, 0.5 microsec would be required for bus acquisition. Transmission of data by the processor curently in control of the bus can proceed in parallel with the priority computation for the next controller, so 500 nanosec are available for transmission of address and data bits over the bus. With a careful bus design, probably of an hierarchical nature, this seems within reach of current technology.

If n_{max} is the maximum number of connections to any one processor, then the total time for all processors to complete one step of the iteration has the form $t(n_{max}) + t_{bus}$, where t_{bus} is the time spent waiting on bus results. If the number of bussed connections times the time for one bus transmission is less than $t(f_{min})$, then t_{bus} is guaranteed to be zero. If all bus transmissions cannot be done in $t(f_{min})$, t_{bus} may still be zero depending on the detailed scheduling of bus transmissions. Assuming a bus capable of 2 transmissions per microsec, if each processor has at least one local connection $(f_{min} = 1)$, then 84 results can be transmitted over the bus without any possibility of a processor waiting for bus results.

Conclusion

The architecture described above is in the process of being evaluated to determine its effectiveness for finite element analysis. This evaluation involves a combination of prototype design, simulation and algorithm development. The approach taken is essentially iterative, in that knowledge gained from constructing algorithms for the initial (small) version of the processor array will be used to influence the direction of future designs. An initial logic design for the nodal processors, along with interfaces for local and global communication as well as cooperative signalling, is complete and construction of a system with a restricted number of processors has started.

[1] Zienkiewicz, O.C. and Cheung, Y.K., Finite Element Method in Structural and Continuum Mechanics, McGraw Hill, 1972.

[2] Weiman, C.F.R. and Grosch, C.E., "Parallel Processing Research in Computer Science: Relevance to the Design of a Navier-Stokes Computer," Proceedings of the 1977 International Conference on Parallel Processing, IEEE and Wayne State Univ.

[3] Flynn, M.J., "Some Computer Organizations and their Effectiveness," IEEE Trans. on Computers, C-21, No. 9, Sept. 1972.

[4] Opferman, D.C. and Tsao-Wu, N.T., "On a Class of Rearrangeable Switching Networks" Bell System Technical Journal, 50, No. 5, May-June 1971.

[5] Bell, C.G. and Newell A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[6] Brinch Hansen, P., "The Nucleus of a Multiprogramming System," Comm. ACM, 13, No. 4, April 1970.

[7] IEEE Std 488-1975, IEEE Standard Digital Interface for Programmable Instrumentation, IEEE, Inc., 345 E. 47th St., N.Y., N.Y. 10017.

[8] Brinch Hansen, P., "Concurrent Programming Concepts," Computing Surveys, 5, No. 4, Dec. 1973.

[9] Varga, R., Matrix Iterative Analysis, Prentice Hall, 1962.