EVALUATION OF FINITE ELEMENT SYSTEM ARCHITECTURES

by

D.R. Rehak, WVT. Keirouz, Z.J. Cendes, C.T. Hendrickson

DRC-12-20-84

December, 1984

Evaluation of Finite Element System Architectures

Daniel R. Rehak[1]
Walid T. Keirouz[2]
Zoltan J. Cendes[3]
Chris T. Hendrickson[4]

Abstract

The finite element method is a powerful engineering analysis tool. However, the scope and size of the problems solved are limited by the capabilities and costs of computers currently used. New computer architectures with parallel processing capabilities can exploit the parallelism in the finite element method. Because of the large number of alternative hardware and software configurations and the high costs incurred in the development of such systems, a methodology is needed to compare proposed finite element systems without implementing either hardware or software. The paper presents such a methodology.

System architectures are evaluated by simulating the execution of the finite element software on the hardware. Hardware is represented as a set of processors and resources while the software is modeled as a set of computational tasks organized into an acyclic directed graph. The simulation is performed by scheduling routines. The result of the scheduling routines is a schedule consisting of task-processor assignments and task starting and finishing times. Also, processor and resource utilization levels are generated. These results provide a means of comparing proposed finite element systems.

---

[1] Assistant Professor, Department of Civil Engineering. Carnegie-Mellon University

Research Assistant. Department of Civil Engineering, Carnegie-Mellon University

[3]Associate Professor. Department of Electrical & Computer Engineering. Carnegie-Mellon University

[4]Associate Professor. Department of Civil Engineering. Carnegie-Mellon University

Evaluation of Finite Element System Architectures

Daniel R. Rehak[1]
Walid T. Keirouz[2]
Zoltan J. Cendes[3]
Chris T. Hendrickson[4]

1. Introduction

Development of the finite element method has been closely linked to advances in computer technology, allowing users to solve problems of increasing complexity. However, there remains a class of problems of a higher order of complexity with computational requirements exceeding the limits of current machines. Recent developments in hardware technology provide parallel processing capabilities which appear well suited to exploit the parallelism present in the finite element method. Alternative hardware architectures for parallel processing include:

- Tightly coupled networks where processors cooperate closely on the solution of the problem (e.g. *Cm** [6]).

- Loosely coupled networks where interprocessor communication is done on a message basis (i.e. an *ethernet*-like inter-processor communications network).

- Attached processors that range from array processors to special purpose finite element machines and *VLSI* systolic arrays [9, 15].

However, some issues must be considered to fully exploit the capabilities of multiprocessor architectures.

- Since most multiprocessor machines are still in the experimental stage, they systems lack user-friendliness and support for software development, thereby making programming costly.

- The selection of a finite element system (both hardware and software) is complicated by the large number of possible hardware and software configurations and the high costs associated with the development of any such system.

[1]Assistant Professor, Department of Civil Engineering, Carnegie-Mellon University

[2]Research Assistant, Department of Civil Engineering, Carnegie-Mellon University

[3]Associate Professor, Department of Electrical & Computer Engineering, Carnegie-Mellon University

[4]Associate Professor, Department of Civil Engineering, Carnegie-Mellon University

- Tuning the program to the hardware to optimize performance becomes harder with more complex architectures because additional issues must be considered (management of private and shared resources, data sharing between processors, interprocessor communications, etc).

Because of the wide range of options and the high costs incurred in the development of both the hardware and the software, the need to evaluate a proposed system without an actual implementation arises. A methodology for the preliminary evaluation of proposed system architectures (both hardware and software organization) is presented here. The methodology does not require that hardware be configured or software be written and makes use of a resource constrained, network scheduling framework to model and simulate a system's performance. Thus, proposed system architectures may be compared to determine which alternatives appear promising and deserve further study.

2. Parallelism and Parallel Processing in the Finite Element Method

2.1. Parallelism in the Finite Element Method

Increasing capabilities of microprocessors and *VLSI* technology make it feasible to build a machine with multiple processors cooperating on the solution of a problem. Special purpose processors that efficiently manipulate arrays are being developed. Such alternatives to the traditional von Neumann machine architecture can exploit the parallelism explicitly displayed in parts of the finite element method algorithms and may reduce computational costs.

The basic finite element analysis algorithm for a linear static problem consists of four major steps:

1. Generate the stiffness matrices and internal load vectors for all elements.

2. Assemble the element stiffness matrices and load vectors into a structure stiffness matrix and internal load vector.

3. Solve the system of equations for the structure displacements corresponding to the applied loads.

4. Recover element quantities (stresses and strains) for all elements from the displacements computed for the structure.

The generation of the stiffness matrices and load vectors for all the elements can be performed in parallel since the computation of one element quantity is independent from all other elements. Also, the recovery of element quantities for all elements can be done in parallel once the total system of equations has been solved. The process of structural stiffness matrix assembly can start as soon as the stiffness matrix from any one element is available. Parallelism also exists in the solution step:

• The solution of the system of equations governing the behavior of the structure can begin as soon as all contributions to a node have been assembled. Also, the recovery of element quantities can start as soon as the displacements for any element have been computed. Systems using marching solvers use this property.

• If the solution of the system of equations does not start until all elements have, been assembled, parallelism in the solver can be exploited through vectorization.

2.2. Parallel Processing Architectures

A hardware configuration consists of a set of processors and their associated resources. A processor may be a stand-alone unit like a *CPU* (Central Processing Unit), a functional unit of a *CPU* like a vector adder, or an attached processor or a customized *VLSI* chip. Resources include primary memory, secondary storage devices, communications channels or data buses, and input and output devices. In addition, resources can be private to one processor or shared between multiple processors. The performance of a program executing on a configuration depends upon the size, organization and type of available processors and resources. Hardware configurations may be classified according to the number of processors in the system and the nature of the network linking the processors. Four general classes of hardware configurations include:

• Uniprocessor systems consist of only one *CPU* and its associated resources. Examples are the *IBM 3083* or *Digital VAX 11/780.* A large number of finite element analysis programs have been written for this dass of machines [11]. However, sequential machines fail to exploit parallelism, and thereby constrain the performance of programs by the speed of the single processor.

• Dual processor machines consist of a uniprocessor, the host, linked to an attached processor by a very high bandwidth channel. The attached processor might be an array/vector processor [14,15] or a *VLSI* systolic array [8,9]. Two finite element analysis programs executing on dual processor systems have been reported, with speedup ratios of 5-*10* [14] and 72-T6[15]. The speedup ratio is the execution time of the program running on the host alone divided by the execution time on the dual processor system.

Since the attached processor acts as a slave to the host parallel processing does not occur.

- Loosely coupled networks consist of several autonomous processors that are not necessarily identical. The processors can access private or shared resources and communicate via a medium bandwidth (-70 *Mbps) ethernet-llke* network. Examples are the *Spice* network [12] and the *Apollo* Domain Architecture. No finite element programs using the capabilities of loosely coupled networks are known to the authors. Such a program would be composed of several processes that may execute on different processors.

- Tightly coupled networks consist of several processors where interprocessor communication is hardware supported and is routed via data buses. Examples of such machines are *Cm*^{0} [6], the *NASA* Finite Element Machine [13] and *ILLIAC IV*[2]. Tightly coupled networks may be subdivided into four classes according to the number of instruction and data streams concurrently processed:

    o An *SIMD* (Single Instruction stream, Multiple Data stream) machine consists of one control or master processor and several slave processor which simultaneously execute the same instruction stream on independent sets of data as in *ILLIAC IV.*

    o *MISD* (Multiple Instruction stream, Single Data stream) machines are also known as pipelined or vector machines. A hardware pipeline consists of a set of functional units (i.e. fixed point arithmetic, floating point arithmetic, index, data fetch, etc.) linked together in series where multiple instructions simultaneously act on the data stream as it goes through the pipeline as in the *CRAY-1,* or the *CDC STAR.* Finite element programs developed for uniprocessor systems can be transported to *MISD* machines.

    o *MIMD* (Multiple Instruction stream, Multiple Data stream) machines consist of at least two general purpose processors linked together by data buses. Multiple instruction streams acting on multiple data streams execute concurrently in *MIMD* machines. Examples of such machines are *Cm*^{0} and the *NASA* Finite Element Machine. No commercial finite element software is available for this class of machines. Using *Cm*^{0} for the solution of electrical power networks [4] (which is similar to finite element analysis), performance improved as the number of processors was increased, but leveled off with a system of nine processors. Also, using the *NASA* Finite Element Machine, the speedup ratios for the various steps of the solution algorithm (element stiffness generation, assembly, solution of equations, element recovery) varied between *2.82* and *4.00* on a four processor configuration.

o Systolic arrays [8,9] are computational cells connected in a regular network. A ceil is a processor with limited capabilities and resources. Data flows continuously through the network and each processor performs simple computations on the data. In a finite element system, they can be used as special purpose processors that perform matrix operations such as multiplication, symmetric triple product, *l*.(Afactorizaiion and backsubstitution.

Architectures offering parallel processing can exploit the parallelism displayed by the finite element method. However, only loosely coupled networks and *MIMD* machines maximize the use of parallelism throughout the finite element analysis algorithm. Effective programming becomes harder with increasing architectural complexity. In particular, scheduling of solution steps on particular processors and allowing for necessary data communications will greatly influence system performance. Developing experimental systems is extremely costly and no formal guidelines exist to determine which alternatives are most cost effective.

3. Modeling and Scheduling a Finite Element System Architecture

3.1. Problem Description

The objective of the methodology described is to provide a comparison of various software and hardware designs which exploit parallelism to improve performance. Designs are evaluated by simulating the scheduling or assignment of computational tasks to particular processors. Alternative designs are compared based on the estimated costs of sample runs developed from the simulation. As a first estimate, the cost of a run is taken as its total execution time.

To study the behavior of algorithms on various architectures, the representations of hardware and software must be independent Furthermore, the software representation must support parallelism in the algorithm and be independent of the size of the problem being solved. Representations satisfying these conditions treat the hardware as a set of processors and their associated resources, while the software is modeled as a directed acyclic network where nodes stand for independent computational tasks in the program and branches represent data flow paths and precedence relationships between tasks.

The finite element program is modeled as a generic non-iterative program structure independent of the finite element problem size. Nodes, representing tasks, can be *subscripted* and as such, stand for the computation of all the variables generated from the node by varying each of its subscripts over its range. Network links represent data precedence relations between computational steps. The size independent generic network model is then expanded to build an explicit network including all the tasks for a given problem, where each task computes one quantity. This expansion is performed using a set of rules for expanding the subscripted nodes and generating the appropriate precedence relations and resource constraints [7].

The simulation of the program on a hardware configuration is performed by solving an integer programming problem whose objective is to find a feasible mapping, satisfying precedence and resource constraints, from the set of tasks onto the set of processors while minimizing the objective function (i.e. total execution time). The solution results are the task-processor assignments and the starting, finishing and execution times for all the tasks. Also, processor and resource utilization levels are generated by the scheduling procedures.

Because of the deterministic nature of the solution technique used, the programs which are modeled are restricted to be non-iterative. This forces non-linear systems to have a predetermined number of iterations per loading step and a predefined number of load steps. Representing the hardware as a set of processors and resources imposes some limitations on the modeling of interprocessor communications because some of the information concerning the physical connections is lost. For example, the cost of communicating between two processors depends on how the two processors are connected. Furthermore, it has been shown that the problem of finding an optimal assignment resulting in minimum total execution time of a set of non-preemptible tasks to a set of processors subject to resource and precedence constraints is WP-complete in almost all cases [5]. To reduce the complexity of the general scheduling problem, some simplifying assumptions are made, although the simplified problem is still /VP-complete. These assumptions are:

- The *resource requirements* of a given task executing on a certain processor are assumed to be independent of all other tasks and processors. An example that violates this assumption is the case of two tasks that must execute in a sequential manner with the second task using data computed by the first. The data must be communicated over the network connecting the two processors executing the two tasks. The resource requirement of the communication will vary from minimal if the two tasks execute on, the same processor to substantial if the data must be transferred over many physical links of the networks. Task-processor uncoupling also assumes that resources utilized by a task during execution are released after the completion of the task. In certain cases, data generated by a task must be saved for later use. This data utilizes a memory resource that is released onfy when the data is no longer needed.

- The *time domain* is assumed to be an integer space. Although time is usually conceived of as a continuous space, computer time is discrete because it can be represented by cycles, where each cycle cannot be decomposed.

- The tasks are assumed to be *non-preemptible.* A task is said to be preemptible if its execution can be interrupted by another task and then resumed at a later time. The non-preemptibility assumption eliminates the need to compute the cost of saving the status of a task when it is preempted and of setting up the task when it is resumed. To provide a degree of preemptibility, a task can be decomposed into a set of sequential tasks, however this eliminates the constraint of having all subparts of a task executing on the same processor.

The scheduling and computer configuration model as formalized below is capable of representing a wide variety of analysts algorithms and all of the architectures described above.

3.2. Notation and Formal Model

The description of the computer system consists of the specification of all of its components. The hardware consists of:

- the set P of processors

$$P = \{p_j : j = 1,\ldots,m\}$$

where *m* is the number of processors. A processor is a *CPU,* a functional unit of a *CPU* (i.e. an adder) or a special purpose chip.

- the set *R* of private and spooled resources in the system

$$R = \{R_k : k = 1,\ldots,l\}$$

where *l* is the number of resources in the hardware system and f_i is the number of units of resource R_i. Resources include, but are not limited to, memory, disk and communications channels.

The algorithm is decomposed into:

• the set *T* of computational tasks.

$$T=\{T_i : i = 1 \dots n\}$$

where n is the number of tasks. The execution of the tasks obeys the precedence relations of the tasks based on data inheritance. This is represented by a binary relation 9 on T^2 that induces a partial order on the set T. The relation $(T_i, T_j) \in 9$ means that task Tj cannot start executing before task T_i has finished. By definition, T, is the starting task (with a starting time of *zero)* and $7'_n$ is the last task to execute.

Three sets of variables are used to describe the scheduling problem. These are the *resource requirements* variables, the *time* variables and the *task-processor* assignment variables.

• The *resource requirements* variables specify the number of units of a particular resource which a task will require while executing on a specific processor. Since the number of resources in the system may be greater than one, the resource requirements of a task T_i executing on processor P_j is represented by a vector R_{ij}. For simplicity in the notation, it is assumed that the resource requirements vector FL_{ij} (task T_i executing on processor P_j) is given by a function /? over *TxP.* Thus,

$$R_{ij} = \{r_{ij1},\dots, r_{ijl}\} = \beta(T_i, P_j)$$

where r_{qk} is the number of units of resource *f* required by task T_i executing on processor Pp

• The *time* variables denote the execution, starting and finishing times of the various tasks.

o t_{ij} is the execution time of task T_i on processor P_j. The execution time is given by a function a over *TxP.*

$$t_{ij} = \alpha(T_i, P_j)$$

o ST_i is the starting time of task Tj.

o FT_i is the finishing time of task T_i.

o *ST^* is the starting time of the schedule, so SIT, = 0.

o t_{max} ^ the total duration of fre schedule, which is equal to the finishing time of task T_n $(t_{max} = FT_n)$.

- The *task-processor assignments are* represented by two sets of variables. The first set of variables ({V_{ij}}) are time independent and indicate if a task T_i executes on processor P_j.

$$Y_{ij} = \begin{cases} 1 & \text{if task } T_i \text{ executes on processor } P_j \\ 0 & \text{otherwise} \end{cases}$$

The second set of variables ({$*_{ijt}$}) are time dependent and indicate if a task *T_i* is executing on a processor P_j at a time *t.*

$$X_{ijt} = \begin{cases} 1 & \text{if task } T_i \text{ is executing on processor } P_j \text{ at time } t \\ 0 & \text{otherwise} \end{cases}$$

Two sets of variables are used because it is easier to express the precedence constraints in terms of the time independent variables, while the resource constraints can be most readily expressed in terms of the time dependent variables.

The set of time dependent variables ({$\hat{}_t$}) completely defines the state of the system at any time *t.* The same statement can be made about the set of time independent variables ({ *Y_{ij}}*) and the starting and finishing times *({ST_i, FTJ).* Therefore, the two representations must be related to avoid inconsistencies.

Direct relations compute one set of variables once the other set has been determined. The definition of the time independent variables ({I_{ij}}) and the starting and finishing times ({ST_i, *FTJ)* in terms of the time dependent variables are:

- *Y_{ij}* is *one* if and only if task T_i executes on processor P_j at some time.

$$Y_{ij} = 1 \iff \exists\, t;\ X_{ijt} = 1 \iff \sum_{t=1}^{t_{max}}$$

- ST_i is the minimum time *t* at which task T_i is executing on some processor.

$$ST_i = t \quad \sim=\gg \quad 3\ y;\ X_{st} = 1\ \&\ X\hat{}_{-1\}} \gg 0 \tag{2}$$

- FT_i is the maximum time *t* at which task T_i is executing on some processor. Task T_i is executing somewhere at time *{t-1)* but not at time *t.*

$$FT_i = t \quad \sim \quad 3\ y;\ X_{g(t-1)} = 1\ \&\ X\hat{} = 0 \tag{3}$$

The time dependent decision variables can be determined from the time independent variables by noting that X_{ijt} is *one* for all times *t* between ^ and *FT^* if task T_i executes on processor P_j.

$$X_{ijt} = \begin{cases} Y_{ij} & \text{if } ST_i \leq t < FT_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Additional relationships can also be derived by using the properties of the starting and finishing times, precedence relationships, resource constraints and the properties of the starting and finishing times.

- For a task Tj, the finishing time *FT!* equals the starting time *ST!* plus the execution time *t* of the task on the processor to which it is assigned.

$$FT_i = ST_i + \sum_{j=1}^{m} Y_{ij} \cdot a(T_i, P) = ST_i + \sum_{j=1}^{m} Y_{ij} t_{ij} \tag{5}$$

- The starting time of a given task is greater than or equal to the finishing time of its predecessors.

$$ST_j \geq FT_i \qquad \forall\ (T_i, T_j) \in \mathcal{P} \tag{6}$$

9 denotes the precedence relation acting on 7*. (T_i, T_j) G ^ means that task T_j cannot start execution until T_i has finished execution.

- The number of units of resources used by a task must be less than or equal to the number of available units at any time *t*.

$$\sum_{1*1}^{m} \sum_{1*1}^{n} X\text{^}p(T_{It}P.) \leq \{f_1^{\circ},\ldots, rf\} \forall f$$

or

$$S\ \sum_{Js\,1}\ \sum_{Is\,1} X_{-}.r_{a} < fi \quad \forall f, \forall k = 1,\ldots,l \tag{7}$$

- Each task will execute on one and only one processor.

$$\sum_{j=1}^{m} Y_{ij} = 1 \quad \forall i = 1,\ldots, n \tag{8}$$

- At any time *t*, at most one task will execute on a given processor.

$$\sum_{i=1}^{n} X_{ijt} \leq 1 \quad \forall t, \forall j = 1,\ldots,m \tag{9}$$

The objective function to be minimized is a function representing the cost of executing the program on the given hardware configuration. As a first estimate, the cost function is taken as the total execution time f_{max}. Thus, the scheduling problem defined above will yield a set of task-processor assignments subject to all resource and precedence constraints which minimize the total execution time t_{max}. Summarizing, the integer programming (*IP*) problem defining the scheduling problem is:

IP:	Minimize	t_{max}
	subject to	(1), (2), (3), (4), (5), (6), (7), (8) and (9).
	where	Y_{ij}, X_{ijt} are *0-1* variables.

Because this integer programming problem is hard to solve exactly, heuristic techniques are used to obtain good, feasible, but not necessarily optimal solutions.

3.3. Scheduling Procedure

The scheduling procedure used to determine the total execution time is an extended form of the *Critical Path Method* (*CPM*) [1, 16]. *CPM* is a two pass network traversal algorithm that guarantees optimality in the case of unlimited resources and an unlimited number of identical processors. The method is modified to handle both a limited number of processors (not necessarily identical) and limited resources. The result is a heuristic set of scheduling rules which do not guarantee optimality but should produce relatively good and feasible schedules.

The first pass of the scheduling process generates a feasible earliest starting schedule by assigning earliest starting times to tasks that become active. A task becomes active when all of its predecessors have finished execution. It is then appended to the active tasks queue. The scheduler assigns the task at the front of the queue to an available processor. Since the task may execute on more than one processor, a set of rules is needed for the purpose of conflict resolution in task-processor assignments [1, 16]. The rules are tried, in turn, until an assignment is made.

1. Select a processor to minimize task finishing time (processor resulting in earliest finish of task),

2. Select a processor to minimize processor waiting time (last processor to become idle),

3. Select a processor to minimize resource utilization (processor resulting with least resource requirements of task),

4. Select a processor at random.

The scheduler proceeds with a second backward pass to generate *slacks*. The slack or float time of a task is the amount of time by which its start may be delayed without increasing the total execution time of the program. Slacks are generated by shifting the tasks forward in time *{right shifting)* in a continuous manner (tasks may not jump over other tasks). Because the order of right shifting of the tasks can affect how far a task will be shifted, another set of rules is needed to resolve the ties if more than one task is to be shifted at the same time [16]:

1. Right shift tasks in descending order of their earliest finishing time.

2. Calculate for each task a latest starting time assuming that only one task is shifted at a time, then right shift in order of decreasing latest starting times.

3. Right shift the tasks in order of increasing resource requirements.

4. Right shift a task at random.

The slack of a task is the difference between the latest and earliest starting times. The slacks can be used as priorities to generate a new improved schedule by an event-driven scheduler. The resulting schedule provides an estimate on total execution time and a set of task-processor assignments.

4. Examples

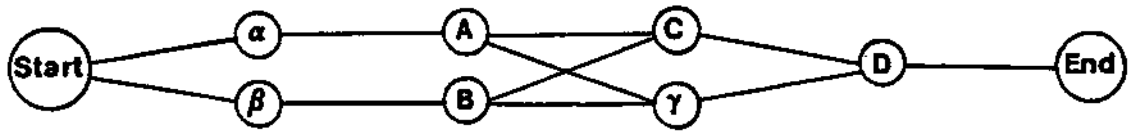
This section presents a small illustrative example of the use of the methodology, followed by a more comprehensive finite element example.

4.1. Illustrative Example

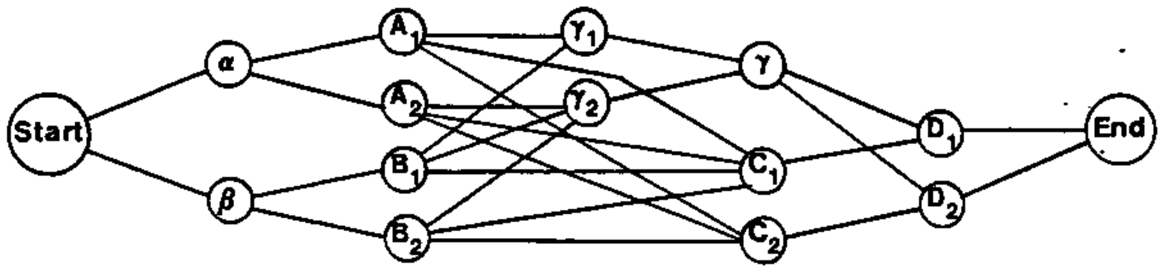
Consider the program shown below, where a, */?* and *y* are scalar variables, A, B, C and D are n-dimensional vectors and Fan n-dimensional function of a scalar. The tasks are the computations of A, B, C (the cross product of A and B), *y* (the dot product of A and B) and D.

```
Begin
    A   :»  F{a);
    B   :*  lifi);
    y   :»  A * B;         {dot product of two vectors}
    C   :=  AxB;           {vector product of two vectors}
    D   :»  y C;
End.
```

The generic network representing the program appears in Figure 1. Since some of the nodes represent the computations of vectors or sum of products, these nodes can be expanded so the parallelism inherent to the computation can be fully exploited. By expanding the generic network of Figure 1 to act on the individual components of the vectors, the network in Figure 2 is derived, assuming vector length *n* is 2.



Figu re 1: Generic Network of Sample Program



Figu re 2: Expanded Network of Sample Program

A hardware configuration is shown in Figure 3. It consists of three identical *scalar/vector* processing units (PU) with a common shared memory (M). The task duration and memory resource requirements for the program tasks are shown in Table 1, along with the task-processor assignments, starting and finishing times. The processor utilization graph for the sample problem is shown in Figure 4. A horizontal bar is associated with each processor. Blank areas represent idle periods while filled areas represent active periods. The resource level plot for the memory is shown in Figure 5. The horizontal axis represents time while the vertical axis represents the percentage of memory used. Such a graph may be produced for each resource in the system.

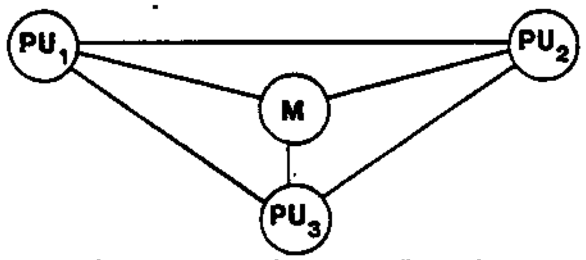


Figure 3: Hardware Configuration

Number	Process	Duration	Memory Reauirement	Starting Time	Finishing Time	Assigned Processor
1	Start	1	0	0	1	3
2	a	3	10	1	4	3
3	*fi*	3	10	1	4	2
4	A_1	15	20	4	19	3
5	A_2	15	20	4	19	2
6	B_1	15	20	4	19	1
7	B_2	15	20	19	34.	3
8	T_1	10	30	19	29	2
9	Y_2	10	30	34	44	3
10	*y*	3	30	44	47	3
11	C_1	23	50	34	57	2
12	C_2	23	50	34	57	1
13	D_1	10	50	57	67	3
14	D_2	10	50	57	67	2
15	End	1	0	67	68	3

Table 1: Task Data and Results

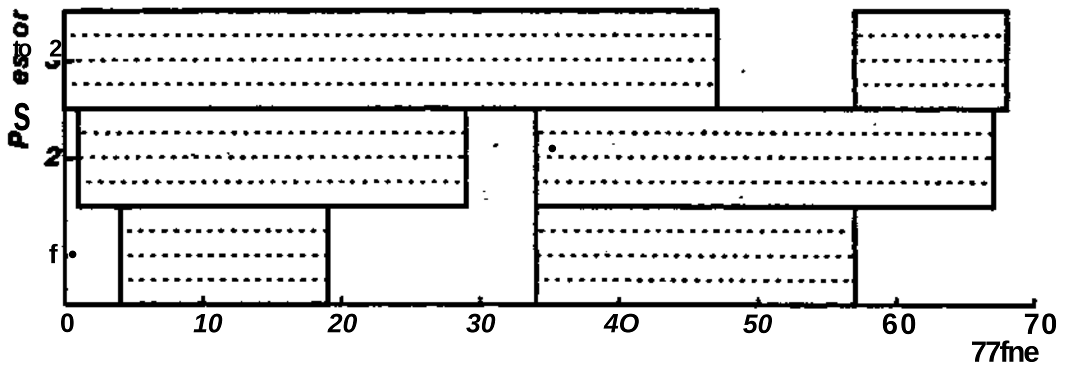


Figure 4: Processor Utilization

4.2. Finite Element Example

The second example deals with the execution of a non-linear substructured analysis of a pressure vessel on a network of processors. The finite element model of the pressure vessel to be analyzed is reproduced in Figure 6 [3]. The pressure vessel is composed of a cylindrical part and a spherical part which are welded together. The pressure vessel is modeled as three substructures: cylinder, sphere and junction. Both the cylinder and the sphere substructures have a linear behavior and are condensed to their interface with the non-linear junction. A pseudo-code listing of the non-linear finite element algorithm, based on that used in FINITE [3,10], is shown in Figure 7. The network representation of the algorithm is shown in Figure 8, where the subscripts C, J and S refer to

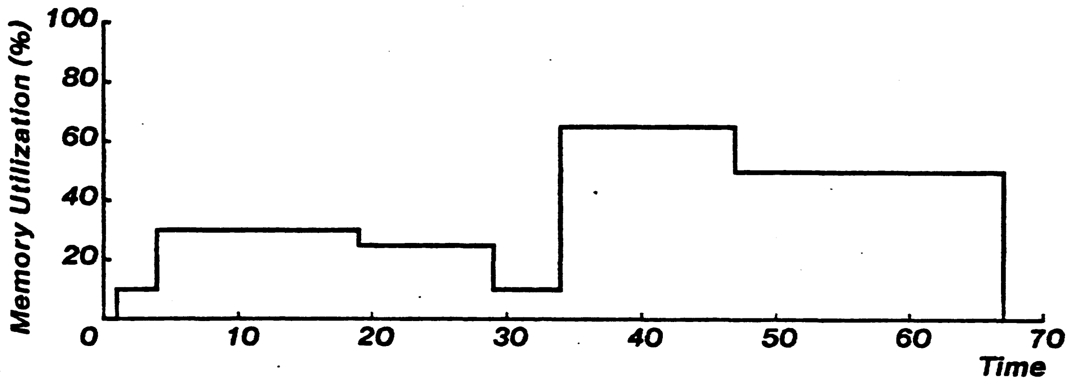


Figure 5: Memory Utilization Level

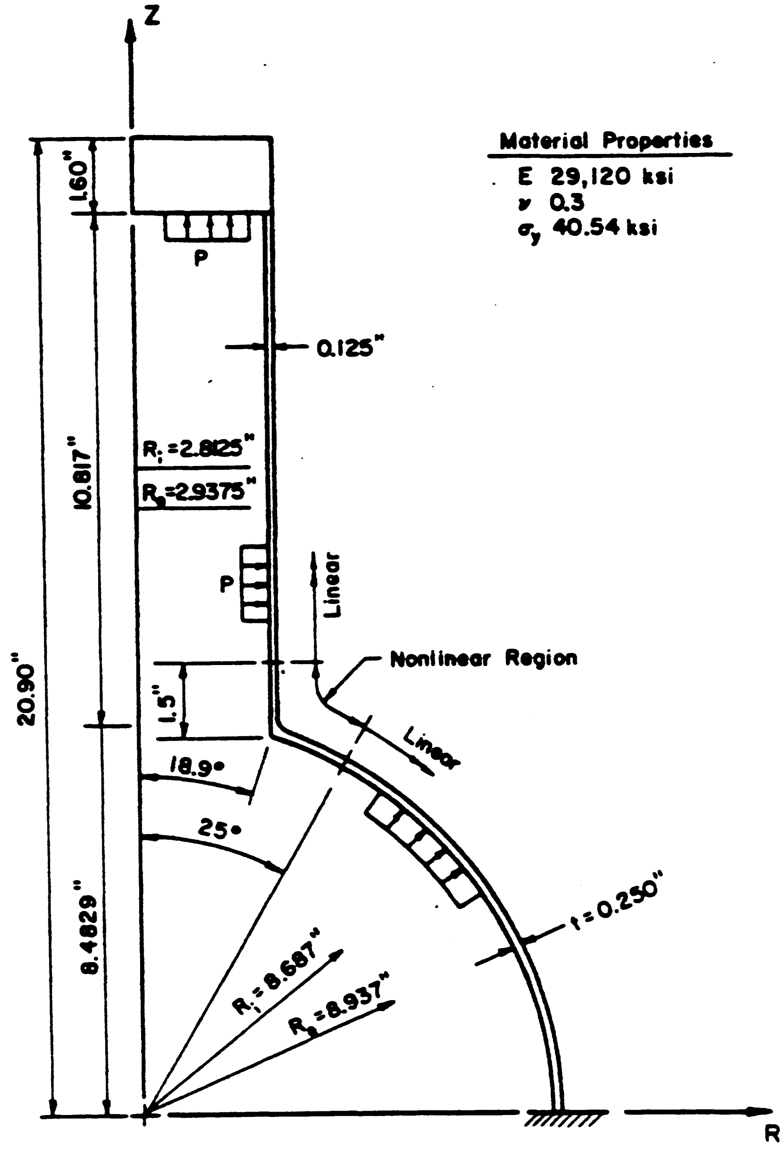


Figure 6: Axisymmetric Pressure Vessel Finite Element Model [3]

the cylinder, junction and sphere substructures respectively. All tasks generating element quantities (i.e. stiffness matrix, internal forces and nodal loads) for all the substructures, can be processed in parallel. Although the condensation of the cylinder and sphere substructures can execute in parallel, the condensation of a substructure cannot start until all element quantities in it have been computed. Furthermore, the reduction of the structure stiffness matrix follows the condensation of the cylinder and sphere substructures and the computations of element quantities in the junction. Finally, the recovery of element quantities for the three substructures can be performed in parallel. In the analysis, the stiffness matrix of the junction is updated and reduced only once per load step. In this example, only one iteration per load step is allowed (all iterations have the same structure and will scheduled in the same manner). One load step is shown in the graph representation in Figure 8 because the modeling of additional load steps is identical to the first load step.

The graph representation of the finite element algorithm is a generic network which is independent of the problem size and element types. As such, it cannot be used directly to determine the durations and resource requirements of tasks dependent on element types and problem size such as computing the element stiffnesses of different types of basic elements and assembling the structure stiffness matrix. The task data for this example were arrived at by performing operation counts (multiplication, addition, square roots, sines and cosines) for each task and using the durations of these operations on an *Amdahl 470 V/8* computer (each time unit represents 500 machine cycles). A complete description of the example is presented in [7].

The problem was scheduled on a variety of hardware configurations. To prevent the simultaneous updating of a structure matrix during its assembly, only one element is assembled at a time. To model this action, fictitious resources restricting access to the structure matrices are used. A unit fictitious resource is attached to each structure matrix and all tasks updating this matrix require this resource. There are seven such resources in the system, one for each stiffness matrix and load vector. Schedules were generated for hardware configurations consisting of a variable number (one to ten) of identical processors. Table 2 presents the schedule lengths and processor utilization levels for the ten schedules. The charts in Figures 9 and 10 show the variation of the speedup ratio and the average processor utilization versus the number of processors in the system. The speedup ratio of a program executing on a system of identical processors is the execution time on a single processor to the execution time on the multiprocessor machine. Processor utilization for a system with ten identical processors is shown in Figure 11. As can be seen from Figure 10, the average processor utilization dropped from 93% in the case of two identical processors to 50% in the case of ten identical processors. This low processor utilization indicates that the algorithm cannot exploit the full

```
1.      Initialize the three substructures.
2.      FOR ALL substructures sub, WHERE sub: = cyl, jun, sph, DO          Assemble Substructures
2.1         FOR ALL elements m, WHERE m : = 1,..., NE^, DO                 Element Properties
2.1.1           Compute [y^                                  {Rotation Matrix}
2.1.2           Compute  [k^                                 {Local Element Stiffness Matrix}
2.1.3           Compute {p 1 ^                               {Local Element Nodal Load Vector}
2.1.4           Compute {ffF}m                               {Local Element Internal Forces Vector}
2.1.5           Compute [k1gm : = MmTfrJm [yl^               {Local to Global transform}.
2.1.6           Compute {p^:» M^p'^
2.1.7           Compute {f>F}m: = M^{^F}m
22          FOR ALL elements m, WHERE m : = 1.....A/E^, DO                 Substructure Quantities
2.2.1           Assemble [kJm into [K]^
22J2            Assemble {p^ into {P}sub
a       FOR ALL substructures sub, WHERE sub: = cyl, jun, sph, DO          Condense Cylinder and Sphere
ai          Reduce [KJ^
3.2         Compute [k]^: = [KJ^ - [Kfe£|b [Kg^[KJ^          {Condensed stiffness matrix}
3.3         Compute {p}-^ •= ^Psub " ^iJsTub ^eJs-1ub ^e^sub  {Condensed load vector}
a4          Compute {f}^: = {P^}^ - [KJ^[KJ^ {P1!^b          {Condensed Internal Forces Vector}
4.      FOR ALL substructures sub, WHERE sub: = cyl, jun, sph, DO          Assemble Structure
4.1         Assemble [k^ into [K]
4.2         Assemble {p}sub into {P}
4.3         Assemble {pF}sub Into {P^}
5.      {A}:=[K]-1({P}.{PF})                                               Solve
a       FOR ALL substructures sub, WHERE sub:* cyl, jun, sph, DO           Element Quantities
6.1         Select {u}sub from {A}                           {Substructure Displacements}
&2          Compute{Ae sub :* [Kee r1 sub ([KJ sub {A^sub -{PJf sub + {PFe sub)
6.3         FOR ALL elements m, WHERE m :« 1,..., NE^, DO
6.3.1           Select {u}m from {A}a|b                      {Global Element Displacements}
6.3.2           Compute {u'}m:=Mm{u}m                        {Global to Local transform}
6.3.3           Compute {e}m: = spatial derivatives of {uf}m {strains}
6.3.4           Compute {a}m:=[EIm{e}m                       {stresses}
```

Figure 7: Non-Linear Substructured Finite Element Analysis Program

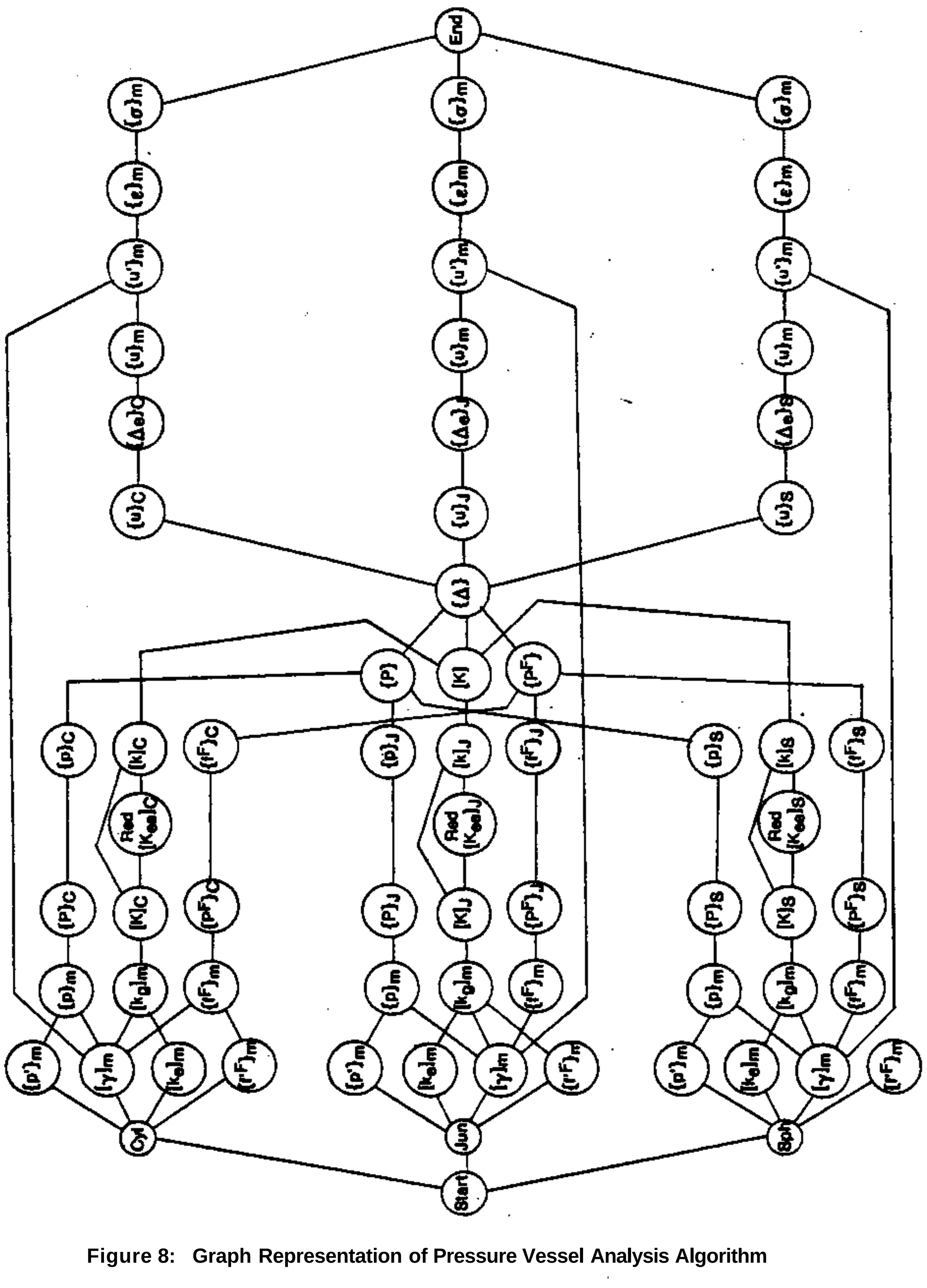


Figure 8: Graph Representation of Pressure Vessel Analysis Algorithm

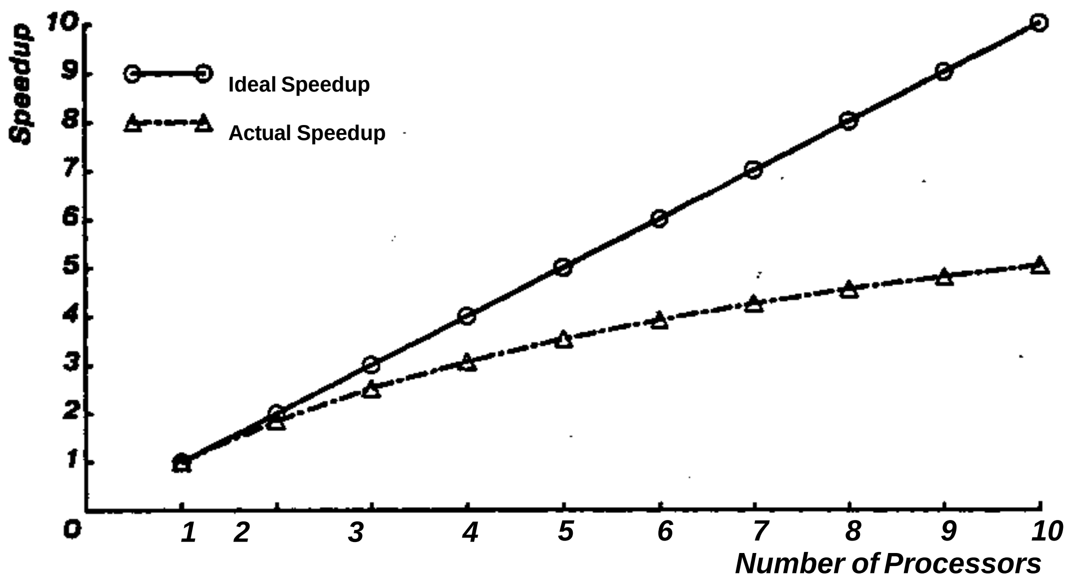
	Schedule	Processor Utilization											
#Proc	Length	AVQ.	1	2	3	4	5	fi	7	S	a	is
1	30,688	100	100	•	•	-	•	-	•	•	•	•
2	16,573	93	100	85	-	-	•	-	-	•	-	-
3	12,196	84	77	75	100	•	•	•	-	•	•	-
4	9992	77	71	66	100	70	•	•	•	-	•	•
5	8689	71	61	67	60	100	65	•	-	•	-	•
6	7820	65	63	61	56	55	100	57	•	-	-	•
7	7203	61 •	52	60	58	52	52	100	53	-	•	-
8	6736	57	100	48	57	57	48	49	50	43	-	-
9	6379	53	46'	55	45	45	45	45	100	47	52	•
10	6085	50	53	43	50	43	43	43	100	43	43	44
9+AP	2314	88	90	85	87	86	85	86	85	87	87	100

Table 2: Processor Utilization Levels

capabilities of the hardware system. The same conclusion can be reached from Figure 11. For a significant portion of the total execution time, only processor 7 is active. This is so because the duration of one critical activity (reduction of the stiffness matrix of the total structure) consumes 29% of the total execution time during which nine out of ten processors are idle.

The pressure vessel problem was also scheduled for a hardware configuration consisting of nine identical processors and one array processor. The array processor is four times faster in performing scalar operations and sixteen times faster in performing matrix operations. The schedule length and the processor utilization levels are given in the last row of Table 2 (the number of processors is 9 + Array Processor). The processor utilization chart is given in Figure 12. Compared to the schedule generated for the ten identical processors, the schedule length was reduced by 62% while the average processor utilization jumped from 50% to 88%. The changes occurred because the durations of the critical and time consuming tasks have been reduced by a factor of sixteen, thus reducing the schedule length and the processor idle time.

Developing a complete set of schedules for a given hardware base and program structure for a variety of problems will illustrate such bottlenecks such as those described above. Other alternative program organizations for this problem which reduce this bottleneck include the development of a multiprocessor solver and the development of an asynchronous finite element analysis algorithm.



Figu re **9:** Speedup versus Number of Processors

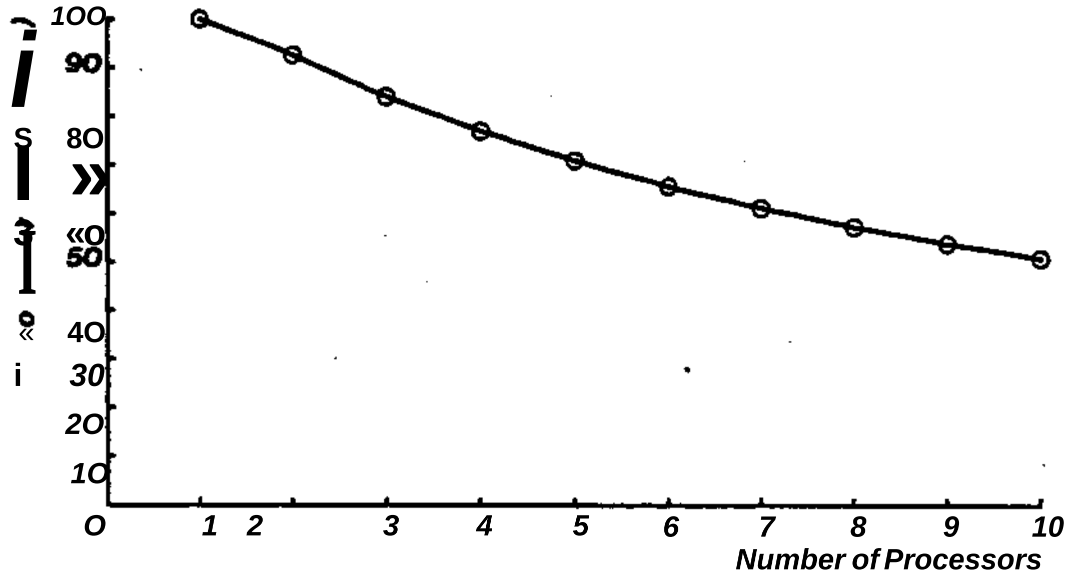


Figure 10: Average Processor Utilization versus Number of Processors

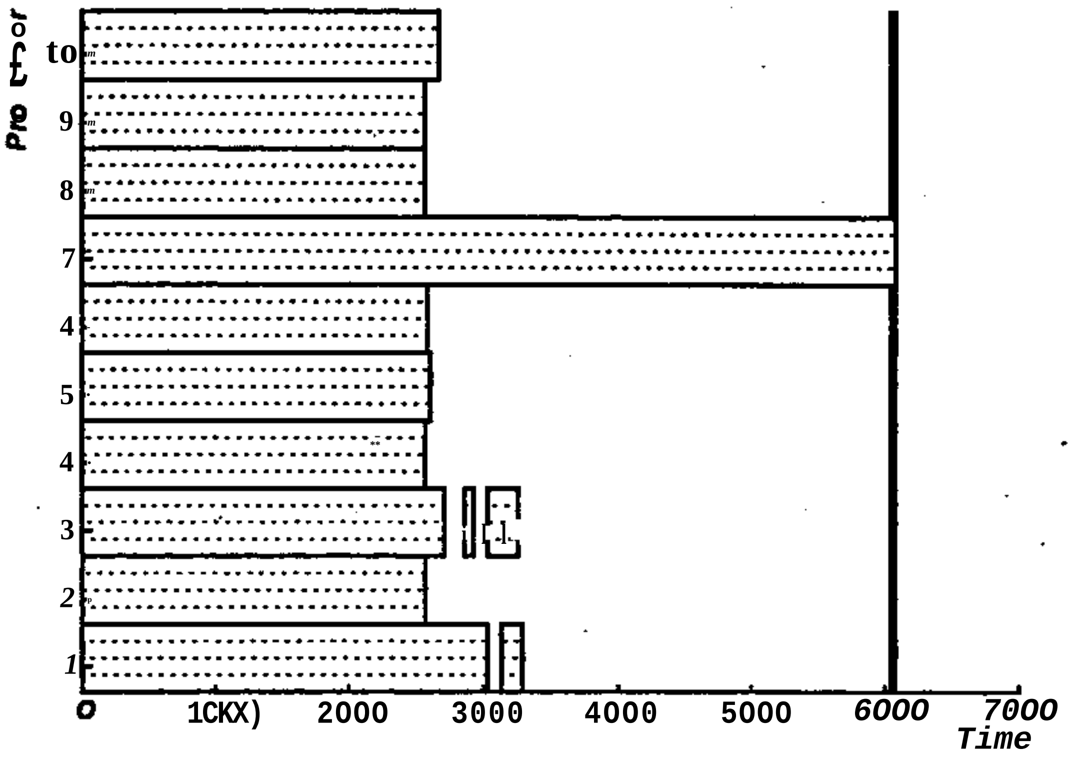


Figure 11: Processor Utilization (10 Identical Processors)

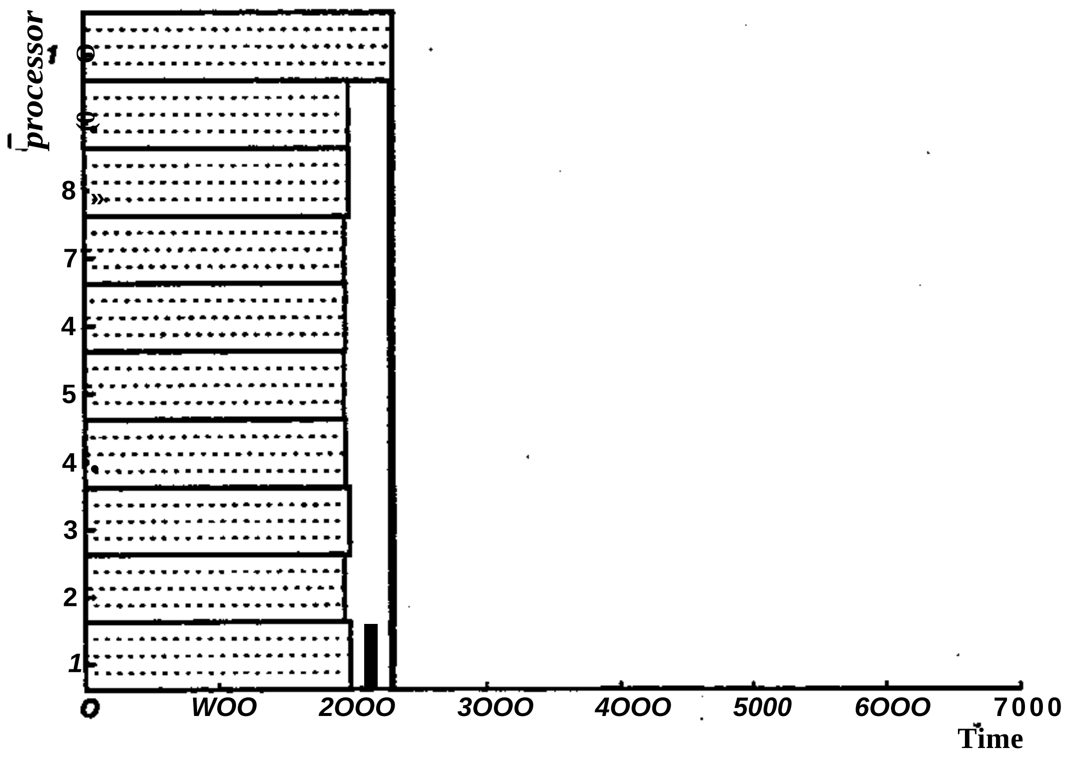


Figure 12: Processor Utilization (9 Identical Processors + Array Processor)

Alternatively, the problem could be modelled with more than one substructure in the junction. Analysis of a variety of such alternatives provides a rational basis for comparisons of alternate system configurations.

5. Conclusions

This paper has presented a methodology for the preliminary evaluation of the performance of finite element system architectures. This methodology provides a tool for the selection of a system for further study without initially incurring the costs of developing such a system. The evaluation is performed by simulating the execution of the program on the proposed hardware configuration. The physical system is modeled as a set of processors and resources while the finite element program is represented by a directed network where the nodes are independent computational tasks and the links represent precedence constraints on the order of computation. The simulation is based on a heuristic scheduling algorithm representing a modification of the critical path method. The result of the scheduling is an estimate of the cost of solving a problem on a proposed hardware and software configuration. Based on a set of such simulations, alternatives can be ranked and those deserving further study or implementation can be selected.

The methodology provides one approach to the evaluation of finite element systems. It provides a flexible mechanism to represent a wide variety of algorithms and hardware configurations. Regrettably, the current methodology does not place sufficient importance on the issues of inter-processor communications and data sharing which have been identified as potential bottlenecks in actual system performance. However, these reservations can be alleviated by extending the hardware and software models and the scheduling procedure.

Possible extensions to the methodology include better models for the hardware and software and extensions to the scheduling problem and the scheduling technique. The hardware model can be improved to include information describing the communications network linking the processors. An extension to the software model is needed to allow the representation of iterative, conditional and asynchronous algorithms. The scheduling problem can be modified to allow the preemption of tasks, to handle resource requirements which depend on the processor used for another task and to model global resource usage. Also, a more elaborate cost function can be used, including items such as resource utilization and resource costs. Finally, the scheduling technique can be extended by developing more sophisticated selection rules and using slacks as priorities to generate an improved schedule.

References

1. Adam, T. L., Chandy, K. M. and Dickson, J. R., "A Comparison of List Schedules for Parallel Processing Systems," *Communications of the ACM,* Vol. 17, No. 12, December 1974, pp. 685-690.

2. Barnes, G., Brown, R., Kato, M., Kuck, D., Slotnick, D.and Stokes, R., "The ILLIAC IV Computer," *IEEE Transactions on Computers,* Vol. C-17, No. 8, August 1968, pp. 746-757.

3. Dodds, R. H., Jr., Lopez, L. A. and Pecknold, D. A., *Numerical and Software Requirements for General Nonlinear Finite Element Analysis,* Technical Report SRS-454, Department of Civil Engineering, University of Illinois at Urbana-Champaign, September 1978.

4. Durham, I., Dugan, R. C, Jones, A. K. and Talukdar, S. N., "Power System Simulation on a Multiprocessor," *Text of Abstracts, Summer Power Meeting,* IEEE Power Engineering Society, July 1979.

5. Garey, M. R. and Johnson, D. S., "Complexity Results for Multiprocessor Scheduling Under Resource Constraints," *SIAM Journal of Computing,* Vol. 4, No. 4, December t975, pp. 397-411.

6. Jones, A. K. and Gehringer, E. F., *The Cm^0 Multiprocessor Project: A Research Review,* Technical Report CMU-CS-80-131, Department of Computer Science, Carnegie-Mellon University, July 1980.

7. Keirouz, W. T., *Preliminary Evaluation of Finite Element System Architectures,* Master's Thesis, Department of Civil Engineering, Carnegie-Mellon University, November 1983.

8. Kung, H. T., "Why Systolic Architectures?," *IEEE Computer,* Vol. 15, No. 1, January 1982, pp. 37-46.

9. Law, K. H., *Systolic Schemas for Finite Element Methods,* Technical Report R-82-139, Department of Civil Engineering, Carnegie-Mellon University, July 1982.

10. Lopez, L A., "FINITE: An Approach to Structural Mechanics Systems," *International Journal for Numerical Methods in Engineering,* Vol. 11, No. 5,1977, pp. 851-866.

11. Noor, A. K., "Survey of Computer Programs for Solutions of Nonlinear Structural and Solid Mechanics Problems.,[11] *Computers & Strucutures,* Vol. 13,1981, pp. 425-465.

12. Spice Project *Spice User's Manual,* Department of Computer Science, Carnegie-Mellon University, 1983. " .

13. Storaasli, S. S., Peebles, S. W., Crockett, T. W., Knott, J. D. and Adams L, "The Finite Element Machine: An Experiment in Parallel Processing," *Research in Structural and Solid Mechanics -1982,* Housner, J. M. and Noor, A. K., eds., NASA, 1982, pp. 201-217.

14. Strohkorb, G. A. and Noor, A. K., *Potential of Microcomputer/Array-Processor System for Nonlinear Finite-Element Analysis,* Technical Report NASA TM 84566, NASA, June 1983.

15. Swanson, J. A., Cameron, G. R. and Haberiand, J. C, "Adapting the Ansys Finite-Element Analysis Program to an Attached Processor," *IEEE Computer,* Vol. 16, No. 5, June 1983, pp. 85-94.

16. Wiest, J. D., "Some Properties of Schedules for Large Projects with Limited Resources," *Operations Research,* Vol. 12, No. 3, May-June 1964, pp. 395-418.