AN EXPERIMENTAL MULTICOMPUTER WITH A REAL-TIME EVENT MONITOR

by
Richard B. Kieburtz
and
J. Mukerji, P. Sadayappan, & D. R. Smith

OGC Technical Report No. CS/E-83-004

An Experimental Multicomputer with a Real-Time Event Monitor

by
Richard B. Kieburtz
Oregon Graduate Center
Jishnu Mukerji, P. Sadayappan, and David R. Smith
State Univ. of New York
at Stony Brook

Abstract

The Stony Brook Multicomputer was conceived as an experimental vehicle to evaluate concepts for the design and use of distributed computing systems. It consists of a network of processing nodes and an operating system that provides services to experimental applications, including message transport, a distributed file system, and facilities for process management. It is also instrumented to monitor program-defined events that may occur at processing nodes of the Multicomputer. The monitor records event occurrences with such fine temporal resolution that it can effectively trace instantaneous state transitions in a distributed system. This paper describes the capabilities of the Multicomputer, and its use to obtain performance data in some applications.

1. Introduction

A testbed for evaluation of distributed system architectures should support certain essential services needed by any distributed application, such as the transport of messages. It should furthermore enable the performance of a proposed system architecture to be evaluated experimentally, even though the architecture under evaluation may differ substantially from the architecture of the testbed itself. Its requirements are those of a somewhat specialized system simulator.

The Stony Brook Multicomputer [1,2,3] was conceived as an experimental vehicle to evaluate concepts for the design and use of distributed computing systems. As it has evolved through its own design and prototype implementation, it is now suited to distributed simulation of a great variety of distributed system architectures. It consists of a processor network together with a distributed operating system called GOSSIP, that provides services to support experimental applications. It does not directly provide services to human users through a command language; these are provided by the UNIX operating system running on a PDP-11/60 which is integrated with the Multicomputer. Software development is done in the UNIX environment.

The basic design of the Multicomputer has been kept simple. A guiding principle in this design has been to avoid complex problems whenever possible, in preference to "solving" them. By following this dictum, GOSSIP exhibits a characteristic of the earlier versions of UNIX; it has a highly modular structure with cleanly defined interfaces. There has been no attempt to optimize its performance to cater for a particular class of applications. However, its basic interprocessor communications are reasonably fast, relative to the speed of the underlying hardware. This communication mechanism will be used in the simulation of any distributed computing system that is mapped onto the Multicomputer architecture.

We have been particularly interested in performance measurement in a distributed computing environment. The Multicomputer has been instrumented so as to allow detailed analysis of sequences of events in the system, with very fine temporal resolution. The instrumentation subsystem provides timed profiles of designated events occurring throughout the Multicomputer, without perceptibly loading the system as it is running an application. Designation of which events are to be monitored is programmable, and can be application dependent.

Before proceeding to describe the Multicomputer, it will be well to outline our concepts of distributed computing systems. A system is *distributed* when viewed at a level of detail in which individual components communicate with one another by exchanging messages. Of course, we recognize that by this definition every conventional computer system can be considered as distributed when viewed as a collection of individual component modules of electronics, which exchange messages in the form of signals on wires. So we want to further clarify what we mean by messages. We say that two (or more) system components *communicate by exchange of messages* when the communications medium delays information for a time sufficiently long that the interconnected system components may undergo state changes. Thus it is an inherent property, in our view of a distributed system, that the instantaneous global state of a system is not observable from any single point within the system. This is to be distinguished from the situation in which system components communicate via memoryless channels, or "wires", and the signal present on a channel can be thought of as a direct manifestation of the current state of the system component generating the signal.

In software systems, we also have good analogies. When processes communicate by direct procedure calls, or by common access to shared memory, the channel of communication is memoryless in this sense. A system that uses such communication is logically concentrated. On the other hand, if processes communicate *only* through buffers (such as the pipe mechanism of UNIX), the channels have memory and a system of processes is logically distributed. From now on, when we talk about messages, we shall mean messages communicated over channels with such a property.

Please note that the property of a system being distributed implies nothing about whether or not the system exhibits concurrency. Although distributed systems tend to exhibit concurrency unless specifically synchronized, concentrated systems can also utilize parallel computation units to achieve concurrent evaluation. And we are all familiar with the technique of multiplexing a purely sequential computer in order to run a system of processes which simulate concurrent activity.

In the Stony Brook Multicomputer, any system of processes will be distributed, as the basic communications channels have memory. However, the degree to which an application makes use of concurrency is not mandated by the system. The operating system allows an application to follow any model of computation that an experimentor has chosen, although support for that computational model must also be provided in the application.

1.1. Locality in a distributed system

The notion of a system as being distributed suggests by analogy to our physical world the concept of a neighborhood, or locality. However, as we understand from geometry, the concept of neighborhood arises only when we impose a measure of distance upon a space. For a distributed computing system as we have defined it, "distance" is traversed by messages. A suitable measure of the distance between two system components is the cost of exchanging messages between them. There are several such measures possible. A software measure is the number of machine instructions that must be executed in order to exchange a message with a given oorrespondent. Another useful measure is the mean time required for a round-trip message exchange between two correspondents. The time can, of course, be normalized to the average execution time of a machine instruction, to obtain a measure more nearly independent of a particular hardware technology.

Distributed computing systems, analyzed by neighborhoods, tend to have very simple topologies. Systems that use a common communications medium, such as Ethernet, consist of a single, unstructured neighborhood. So also do some store-and-forward networks, such as ARPANET, at the level of abstraction seen by the user. This is because actual routing, which might contribute to a cost measure, has been made transparent to the user (and in fact may be variable, with different packets of a message following different routes). In store-and-forward networks with more constrained routing, neighborhoods will become apparent.

In the Multicomputer, nodes are connected point-to-point, and an immediate neighborhood of any node consists of all nodes to which it is directly connected. Larger neighborhoods are those reached by forwarding messages through one, two, or more intermediate nodes, until the network "diameter" is reached, where this is defined as the number of node stages sufficient to reach any arbitrary node in the network from any other. We have chosen a hierarchical tree interconnection topology because, of all the networks whose diameter is logarithmically related to the node total, this seemed the closest match to the logical structures arising in most applications. This aspect is discussed further in section 5.

2. Evaluating a Distributed System

The primary purpose of a testbed is to enable the experimental evaluation of proposed system designs, prior to building a full-scale system prototype. With a distributed system, not only is prototyping likely to be expensive, but measurement also presents difficulties. A testbed should provide the ability to simulate the performance of a prototype system, but in a single laboratory setting, and without the demands on performance that may be characteristic of an operational system.

There are several aspects of the behavior of a distributed system design that should be evaluated, and not all of these are easily determined by experimentation. A system may be distributed in order to achieve ***reliability***, by the use of redundant components, isolated from one another to minimize the probability of simultaneous failure in consequence of some disaster. The reliability of a design cannot be measured directly, but the performance of a redundant backup mechanism can be tested experimentally.

A system may be distributed because it is responsible for managing a physically distributed database, or set of devices, or both. Systems that support asynchronous, possibly concurrent transaction activity have the responsibility to prevent independent transactions from interfering with one another. A testbed cannot effectively simulate such a system in actual operation, but it can test the effectiveness and performance of the interference avoidance algorithms used in a system.

Some systems are designed to allow the multiple tasks of a complex problem to be processed concurrently by a network of processing nodes. The motivation for this may be to enhance reliability, to manage transactions initiated by a set of geographically distributed customers, to utilize processors to meet the demands of time-critical processes, or just to increase throughput. In any of these cases, there is usually a design goal for system performance. While analytic performance estimates provide important data to designers, experimental performance evaluation remains highly desirable.

The Stony Brook Multicomputer has been designed primarily to permit experimental performance analysis. Thus it does not seek to provide high-level support for transactions, nor does it contain redundant backup mechanisms, but it does include a hardware-supported performance monitor capable of recording events in quasi-real time. The applications for which it is intended are experiments in which a distributed system design is installed on the Multicomputer. Parameters critical to a performance model will be measured experimentally, and scaled to the timing parameters of a full-scale design by an offline analysis.

Typical experiments planned for the Multicomputer include simulation of a distributed architecture for evaluation of functional-language programs, investigation of compilation as a pipelined process, and investigation of the performance of several divide-and-conquer algorithms in which the cost of interprocess communication can be taken into account. These applications are discussed in more detail in Section 6.

3. A hierarchical system architecture

The basic function of a distributed system testbed is to support a system of processes representing an application, and which interact with one another by exchanging messages. At first glance, it might seem that the most natural interconnection topology for a testbed intended to meet this general mandate would have only a single neighborhood, i.e. would make the locations of processes within the testbed network transparent to an application. This is not necessarily the case, however. The "application" to be run on an experimental system is not always the distributed system under evaluation, but is more frequently an *experiment* designed to evaluate a particular distributed system architecture. The experiment may require control, scheduling, test vector generation, etc. which are outside the immediate scope of the distributed application that is to be evaluated. These functions seem most naturally to be provided by a control process whose neighborhood should include those processes of the application that it controls. If the application involves subsystems, then the process hierarchy used in an experiment may be more than one level deep, as shown in Figure 1(a).

The Multicomputer provides a tree of processing nodes on which to map a hierarchy of processes. For instance the tree illustrated in Figure 1(b) may be used to run the processes of Figure 1(a) by multiplexing a cluster of processes on a cluster of (possibly fewer) processing nodes. In this picture, the processes at the frontier of the process tree represent those derived from a system to be simulated; these processes may relate to one another in an arbitrary topology of neighborhoods. In the simulation environment provided by the Multicomputer, all communications between processes on this frontier are routed through a control process.

The Multicomputer system architecture has been designed to support process hierarchies of the kind just described. It consists of a tree of processing nodes, in this case DEC LSI-11 processors with 56-Kbyte memories, whose arcs are 16-bit parallel, bidirectional communications links with independent latch buffers in either direction. We refer to these arcs as *control links*.

This processing tree is augmented by a secondary tree which provides a distributed file system for use by processes running at nodes of the primary tree. The file systems allow processes to make use of secondary storage and to exchange data with one another. It also supports multiplexing of processes on the nodes of the primary tree by providing storage for suspended process images. Both the primary and secondary trees are shown in Figure 2. The nodes labelled T in the secondary tree are also LSI-11 processors, each with 256 Kbytes of memory. The node labelled G is a DEC PDP-11/60.

The one-level-deep subtrees of the primary processor hierarchy can be thought of as forming overlapping clusters of "leaves" of the secondary tree, as illustrated in Figure 3. Each of these clusters is served by a single T-node of the secondary tree, which implements a file system accessible only to the processes that run on nodes of the cluster. Inter-cluster communication can be achieved in two ways. The P-node at the root of each cluster is also a member of the cluster above it in the P-tree, and can relay messages that pass from the node above it to those below. The file system that serves each cluster also has access to the global file system which is supported by the G-node. Thus file images can be transferred between clusters by using the G-node as an intermediary. In most applications we have considered, inter-cluster communications are infrequent, however.

Arcs of the secondary tree support the movement of file segment images between local file systems and P-nodes within a processing cluster. These arcs are capable of two modes of operation. In one mode they emulate the control links of the P-tree, sending messages between processors a word at a time. In the other mode, they function as direct-memory-access (DMA) links, transmitting data in burst mode at close to the maximum access rate of the memories, and without processor intervention. The DMA links can make use of 18 address bits, and so can access the full 256 Kbyte address space of a T-node, even though the LSI-11 processor at each such node can access less than one-fourth of this address space. Thus the majority of the memory at each T-node is used only for storage of file segments.

3.1. Hierarchical control

The Multicomputer architecture has been designed to support a relatively flat process hierarchy. In this hierarchy, a superior process is always to be run at an interior node of the P-tree (called the control node), from which it never migrates. Subordinate processes, which as we have said may be the processes of an experimental application controlled by the superior, can be run interchangeably on any of the P-nodes in the cluster rooted at the control node.

Process multiplexing has been kept simple; a P-node supports only one process at a time. The control process which runs at its superior node can preempt the execution of the process running at a subordinate P-node, ordering that process to be suspended and replaced by another to be run in its place. When this happens, a *process suspension record* (PSR) is created and stored in the local file system that serves the cluster. The control process can designate a file containing the PSR of another process to be resumed, or can call for the creation of a new process. By allowing only a single process to be active in a P-node at any time, the resident portion of GOSSIP is relieved of the task of supporting a general purpose interprocess communication facility. It needs to do little more than to enable the exchange of messages between processors. Interprocess communication is a duty of a control process, which also controls scheduling within a cluster, and must maintain a complete data base on the activities of an entire neighborhood of communicating processes.

3.2. Message based communication

GOSSIP supports the transport and acknowledgement of messages between adjacent nodes [4]. These messages are intended for the functions of control, status reporting, and support of higher-level protocols, so it has been considered more important to make the message facility simple, efficient and reliable than to make it very general. Accordingly, messages are bounded in length (less than 32 bytes) and in number of message types. There are eight message types, of which seven are interpreted by GOSSIP, and one type denotes messages to be interpreted by applications processes. An application can, of course, define higher levels of protocol which impose additional structure upon messages.

The format of a message is illustrated in Figure 4. A header word defines the message type and its actual length, in words. The header also contains a field to identify the message channel over which a message is to be sent, or has arrived. Note that channel identifiers give only the relative direction of a neighboring node within the Multicomputer, and are not process identifiers. When a message is actually transmitted over a control link between two processors, the channel identifier becomes redundant, and this field of the header is replaced by eight bits of redundant code for error detection. All one- and two-bit errors in header words and acknowledgements are detectable.

A cyclic redundancy code (CRC) word was originally provided at the conclusion of each message. However, it was found that the incidence of occasional errors in message transmission was so small as to be practically unobservable. Intermittent but frequently recurring errors, such as those that occur due to noisy cable connections, are detected by the coding of header words and acknowledgements. After observing the system in continuous operation over a period of time, the software-implemented CRC check on the bodies of messages was removed in order to reduce the overhead of message transmission. To send a message and receive its acknowledgement currently has a cost of about 140 instruction executions.

Messages are used in GOSSIP to support process scheduling within a cluster, to transmit capabilities for the use of shared files, and to support higher-level protocols, such as one that defines point-to-point message channels to connect processes running at nodes beyond the immediate neighborhood of a processor [4]. Messages are also used to request file operations of the local file system serving the nodes of a cluster, and to receive status information indicating the outcome of a service request.

4. A distributed file system

Processes active in the Multicomputer can communicate by means of messages, but the message facility is primarily intended to carry control and status information. Data and programs are moved and stored by using the facilities of a file system. Each cluster of processors in the P-tree (and therefore the processes that run on this cluster) is served by an individual local file system. A process can directly access only the files within the local file system serving the cluster to which it belongs. A process that has been assigned to the superior P-node of a cluster (such as 1, 2 or 3 of Figure 3) which is not at the root of the P-tree can have access to two file systems, since the node on which it runs belongs to two overlapping clusters. As we have said, a process assigned to the superior node of a cluster will normally perform a control function, rather than an application defined task.

The use of independent, local file systems to serve process clusters provides an effective interprocess communication facility capable of handling data objects of arbitrary size, and avoids the complexity inherent in supporting file migration, location-independent files, or files which exist in multiple copies. Of course, any or all of these concepts may be built into an application to be run on the Multicomputer, but they are not characteristic of all applications.

The Multicomputer also supports a single, global file system whose contents are indirectly accessible from every P-node cluster. A P-node cluster never uses a global file directly; instead it may ask for a copy of a global file to be made in its local file system. The operating system, GOSSIP, does not guarantee that copies made of a global file will be kept consistent. If consistency of copies is required by an application, it is up to the application to define and implement the algorithms required to maintain consistency. A global file may also be updated by overwriting it (or segments of it) with a file from one of the local file systems. Again, it is the responsibility of an application to ensure that multiple updates do not interfere with one another.

The global file system is supported by the PDP-11/60 which runs at the G-node of the Multicomputer. The components of GOSSIP that implement the global file system are embedded as processes in a UNIX environment on the PDP-11/60. UNIX files may be copied to global GOSSIP files, and vice-versa. Each local file system will also make use of a global file to provide backing store for file segments that might otherwise overflow the address space available at a T-node.

4.1. Segmented files

Each file consists of an indexed set of zero or more segments. A segment is a variable-length sequence of bytes, and is the granule of data transfer between a P-node and a file system. The segments which constitute a file can be created or deleted dynamically, and the length of any segment is the number of bytes that were last written into that segment.

The contents of a file are accessed by transferring segment images, one at a time, between a file system and a P-node, or between a T-system and the G-system. A segment transfer is accomplished in response to a request from a P-node. A request names the file system (unique within the Multicomputer) the file, and the segment index. Operations on segments include **create segment, delete segment, read segment, write segment,** and **copy segment** where the copy operation takes place between local and global files.

Segmented files would offer no profound advantage over the use of a single-level name space, except for the fact that files in the Multicomputer are also subject to capability-based protection. A file, not a segment, is the granule to which protection applies. We anticipate that it will often be the case that several segments will share the same access restrictions. The use of segmented files allows a process to have access to a large number of individual data objects, represented as segments, but allows the number of capabilities that the process must retain to be much smaller.

4.2. File protection

The protection system on the Multicomputer can be viewed as defining a *virtual protection machine* which provides to the user a protected file system. System functions are executed on this machine in a privileged mode. This permits them to gain indirect access to the contents of capabilities, analogous to the way that privileged processes can gain access to specific hardware registers in a conventional computer. A process running in unprivileged mode on this virtual machine does not have access to the actual capabilities for files; capabilities are transparent to a user process.

The concept of the virtual protection machine was found advantageous to provide enforcement for locally centralized control in the Multicomputer, for use of a local file system to support interprocess communication, and for process multiplexing on the P-nodes of the Multicomputer.

Since the protection system occupies the same physical address space as does an applications program, the LSI-11 processors used in the prototype implementation of the Multicomputer have been modified by adding primitive two-mode memory protection hardware. This allows the address space to be partitioned into two regions. The protected region includes all I/O bus addresses, and a designated, low-address region of memory. The resident code of GOSSIP and its tables and buffers reside in the protected region; applications code and data are loaded in the unprotected region. Processor interrupts are always masked when executing in the privileged mode, unmasked when executing in the unprivileged mode.

With memory protection for kernel code in a P-node, and physical separation of the file server processes (at T-nodes) from the applications processes (at P-nodes), it is possible to have capability protection for files. Only the application code executed at a P-node is regarded as untrustworthy. This untrustworthy code has no means of direct access to a file system, because of the physical separation of file systems onto separate nodes. File access requires the use of messages passed to a file system over a control link, which cannot be directly accessed by applications code executing in the unprivileged mode. A file system will only respond to messages of a specific header type (TFILE). The file protection subsystem, which is part of the resident kernel of GOSSIP at each P-node, prevents forgery of such messages by an applications process. A file server process, executing at a T-node, can safely assume that every request it receives has been validated before being sent.

The file protection subsystem maintains, on behalf of the applications process running at a P-node, a list of file capabilities (C-list). Each capability contains

i) a file name, and the name of a local file system to which it belongs;

ii) a set of *rights* for use of the file;

iii) a set of segments of the file, for which access is authorized

A process acquires a capability either by

a) creation of a file (to which the applications process at this P-node has the rights of an owner), or

b) a grant from a neighboring P-node.

A capability grant is made in the form of a control link message whose header type is CAP, and which is interpreted by the protection subsystem rather than directed to the applications process. An applications process refers to a file by use of an index into the C-list, which has been furnished to it by the protection subsystem. When an applications process makes a request for a file operation, the request is interpreted by the protection subsystem, and validated against the rights contained in the capability to which it refers. If it is valid, a file transfer request is formatted as a message of type TFILE, and addressed to the appropriate local file system.

The original motivation for implementing memory protection in a P-node was in order to isolate the effect of software errors; to help in distinguishing between bugs in an untried operating system and bugs in the code of an experimental application. However, we believe it will be equally important to be able to prevent misuse of the file systems by potentially errant applications. Protection is of greater importance in a distributed system than in one which is concentrated, because of the extreme difficulty of surveillance of the whole system.

The incremental cost of file protection, whether measured in terms of operating system complexity or in bytes of resident kernel code, is very modest. Protection is applied only to operations that require interpretation through calls to the operating system, and not to operations that might ordinarily be carried out by in-line code. Furthermore, no additional levels of indirection are introduced in order to implement protection. File naming is a service that would be required in any case, as would formatting of file access request messages. The system overhead directly attributable to protection enforcement is only that of checking the types of messages that originate with an applications process, and checking the rights held in a capability.

4.3. File access rights

The rights associated with a file are divided into two classes. *Generic* rights refer to the file access operations that can be performed by a local file system, **read, write, delete, create segment, delete segment.** These rights authorize the transfer of file segment images between a file system and the address space of a process running on a P-node, or authorize operations that affect the file directory in the local file system. Other rights, called *auxiliary* rights [10] may restrict the direct use of a file image to specific procedures of higher levels of the operating system.

4.4. Process multiplexing in the P-tree

In the Multicomputer, auxiliary rights are used to protect the primitive operations that implement process switching on a P-node. For instance, when a control process orders the process currently running at a P-node to be suspended, a **suspend process** primitive will be invoked, naming as an argument a file into which a PSR is to be stored. This file must be one for which the active process holds the auxiliary right **suspend process** but on which no generic rights are held. Thus, the unprotected code of an applications process may not read or write directly to this file.

When the primitive operation **suspend process** is invoked, GOSSIP does not immediately attempt to access the file that is named as an argument in the call. Instead, its first action is to format a **segment read** request to the local T-system, naming as an argument a system-defined file that contains the load images of a set of system-defined procedures. These are the non-resident portions of GOSSIP. The segment named in the read request is one that contains the code of a procedure to implement the creation of a PSR. This segment is read into unprotected memory locations in the P-node, and overwrites the code of the applications process being suspended.

Before giving control to this procedure, however, the resident kernel of GOSSIP must first establish for it a capability to write into the PSR storage file. To do this, a copy is made of the capability given as an argument in the original primitive call, but the rights of the copied capability are modified from a *rights amplification table.* This table contains a set of generic rights which are to replace the auxiliary right that has been invoked on the named file. The system procedure which has been loaded is then passed control, giving it as an argument the index of the new capability to the PSR file which has been created for its use. The procedure gets control with memory protection disabled, allowing it direct access to the C-list of the suspended process so that images of these capabilities can be incorporated into the PSR being created. After the PSR is written to a new segment of the PSR file, the procedure returns control to the resident kernel of GOSSIP, which re-initializes the C-list storage, and prepares to interpret the next directive from a control process. The inverse operation, using a PSR to reactivate a suspended process, takes place in a similar manner, but also makes use of a file from which the process' code segment is loaded.

5. Instrumentation

Analytical performance measurement of a distributed system is a challenging task. Performance bottlenecks typically show up as excessive synchronization of the components of a system. That is, components that might otherwise be expected to execute concurrently are observed to spend a large fraction of available time waiting for notification of events from other system components. Although it is not difficult to observe waiting by system components, it can be quite difficult to analyze the cause of synchronization, and to infer how unwanted synchronization might be reduced. This is because it is not possible, in a distributed system, to observe the global system state with precision from any observation point within the system.

Our solution to this problem, within the framework of the experimental Multicomputer, has been to superimpose upon it an essentially ***concentrated***, real-time event monitor which makes use of dedicated hardware attached to nodes of the Multicomputer. It is possible to do this because the raw processing speed of the hardware components used in the Multicomputer is not very great. Therefore, by using a fast microprocessor, it is possible to multiplex observation channels rapidly enough to obtain a "snapshot" of selected components of the system state, over a limited number (up to 16) of nodes of our network. Of course, not very much state information can be extracted from each node in such a snapshot. Therefore it is important that the particular state component to be monitored can be selected, for each experiment to be run on the Multicomputer, under programmed control. The monitoring strategy is simple, but effective.

At each P-node of the Multicomputer there has been added, on the circuit board containing the memory protection hardware, a three-bit latch called an ***event register***, which is addressable on the processor's external bus (Q-bus). Any three-bit code can be stored into this event register by a single MOVE instruction executed by the LSI-11 processor. The event registers from up to 16 P-nodes are connected into designated bit positions in a 48-bit data accumulator register, whose entire contents are read as a sequence of 8-bit bytes on a 50 microsecond duty cycle. Strobe circuitry prevents indeterminate results from occurring because of a race condition between reading and writing to bit locations in the 48-bit register. A block diagram of this real-time event monitor is shown in Figure 7.

Within each 50 microsecond reading cycle, each of six eight-bit bytes of the data accumulator are read by a Signetics 8X300 microprocessor, and compared with the value of the accumulator byte from the previous read cycle. If any byte has changed in value from the preceding read cycle, that byte is pushed into a hardware FIFO queue, along with the value of a 16 bit counter which is incremented once each read cycle. Thus, changes in the global state of the Multicomputer, as represented by the three-bit code selected for an individual experiment, are stored in the FIFO queue along with a time stamp of 50 microsecond precision.

This temporal precision is close enough to instantaneous observation to suit most conceivable experiments that could be run on the Microcomputer. It can be compared with the average instruction execution time of approximately 8 microseconds. In 50 microseconds, an LSI-11 processor can execute a sequence of approximately six instructions. If an observable state change occurs as a consequence of executing the six-instruction sequence, then one of these instructions will write to the event register of the local P-node.

We can further constrast these timings with those of events which are likely to be of major consequence in an application. Significant events in a distributed system tend to be marked by the transmission of messages. Recall that the average number of instruction executions (by a processor at one end of a control link) required to send a message and receive its acknowledgement is approximately 140, or roughly 20 times the number occurring during a basic monitoring cycle.

The rest of the process of recording global state change data is straightforward. The PDP-11/60 processor is interrupted whenever the FIFO queue becomes half full, and its contents are recorded on a sequential file, along with a low-precision (16.67 ms.) time stamp, for later, off-line analysis. Provision has also been made to dump the FIFO queue often enough that the high-precision counter does not overflow between times that the queue is dumped, for otherwise, the two-level time stamp could become ambiguous.

5.1. Use of the real-time event monitor

The real-time event monitor enables a small number of event types to be chosen for analysis, and the occurrence of these events at nodes of the P-tree to be recorded with 50 microsecond precision during the course of an experiment on the Multicomputer. The location of an event within the P-tree is determined by the position of its code in a data accumulator image. Analysis of a time-stamped event trace consists in computing time intervals that separate occurrences of pairs of events of interest, and statistical analysis of these time intervals. From such analysis can be obtained processor utilization ratios, statistics of arrivals of service requests at a file system, expected waiting times, etc. The causes of observed synchronizations of supposedly concurrently executing processes can be inferred from a well planned experiment by using this technique.

It will help the reader's intuition if we describe in some detail the plan of an experiment, using the real-time event monitor. Suppose we want to measure the activity of processes in the P-tree, to determine an *effective* processor utilization ratio, and to determine expected service times for file segment transfers and the effectiveness of scheduling, when running some unspecified application. We begin by giving a state diagram (Figure 6) of the process states of interest in this experiment, and of the events which induce state transitions.

The significant events are all associated with the transmission or receipt of control messages by processes running on nodes of the P-tree. A process can send a message to another process (or to the kernel of GOSSIP) on an adjacent P-node. The sender might either await a reply to this message, or it might proceed without waiting. These two classes of events are distinguished, as they have different implications insofar as processor utilization is concerned. If a process must await a reply, we can assume that its processor is busy-waiting until that reply arrives. Otherwise, we may assume that the process is active, and its processor is effectively employed. We have also assumed in this experiment that when a process sends a request for transfer of a file segment, it will be forced to wait until a confirming message arrives to indicate that the requested service is complete.

The event "processor reset" indicates a preemtive interrupt from a superior control process in the P-tree, which aborts the activity of the currently active process. The state transition in this case is (arbitrarily) made to state 0, because the processor is assumed to be active in preparing to initiate another process. However, in a different experiment, one might wish to designate another process state by this event.

As can be seen from the transition diagram, it may be the case that several different events can induce the same state transition. For the purpose of the experiment we have outlined, there will be no need to distinguish among these events.

The next step is to encode the state transition events into three-bit event codes that can be stored in the data accumulator of the event monitor. A possible encoding is illustrated in Figure 7. Note that from the codes, it is possible to infer what transition has occurred without having traced previous states of the process. We consider this desirable from the point of view of robustness of the experimental data, because it does not rely upon synchronization of a sequence of observations in order to infer the state of a process. However, many other encodings are possible, including multi-word code sequences, if it were really necessary to provide a fine-grained characterization of events.

Once an experiment has been planned, it is necessary to embed event-recording instructions at the appropriate locations within the program that is to be executed. In the case of the example we have considered above, all of the significant events involve use of the message transport facilities provided by the operating system, and the event-recording code is easily inserted into an "instrumented" version of GOSSIP at the levels that support file access and message transport. For other experiments, it might be necessary to embed the event-recording instructions into an applications program.

In the example considered above, it would be very easy to obtain gross statistics on the utilization of each monitored processor; these would simply be the relative fractions of time the process spent in each of the three states, active, awaiting a control message, or awaiting service from a local file system. If we had found from an initial analysis of the data that processes seemed to be spending an inordinately long time awaiting service from file systems, we might wish to determine whether contention for that service had become a significant factor. To make this determination, the same experimental data would be reevaluated, but now, the traces of processors within a common cluster would be analyzed to determine the degree to which they overlapped one another in occupying state 3,awaiting a file operation. Since the P-nodes in a cluster are served by a common file system, overlapped service times indicate possible contention for use of the processor at a T-node.

A yet finer-grained analysis of the data could determine the distribution function of service request interarrival times, and could approximate the distribution of service times as a function of mean arrival rate. A software package for analysis of the raw traces produced by the real-time event monitor has been prepared by M. Hamza [6].

The real-time event monitor is the most recently engineered component of the Multicomputer, and no experiments using it had yet been run at the time this paper was written.

6. Experimental applications

The Multicomputer has been running in part since the fall of 1980. Since then, additional software components have been developed, and more copies of its hardware components have come on line. In its current operational configuration it runs with the P-tree and the P-Kernel while the T system software is being debugged. The GOSSIP code to support the local and global file systems is complete, as is the P-system kernel through the levels of message-based communication, file access, and protection. The process management level is not yet complete.

The initial experiments run on the Multicomputer did not require use of the file systems, but involved asynchronous concurrent execution of tasks distributed over the P-tree. These experiments included an algorithm to find the least cost Hamiltonian circuit of a graph (travelling salesman problem), a game-playing program (Othello) which used a logically parallel alpha-beta pruning algorithm [7] to limit search, an experiment based on the 8 queens problem, and a tree search problem. This last application will now be described in order to appreciate the capabilities of the multicomputer as a research tool.

In [8] Keller, Lindstrom and Patil at the University of Utah have proposed a distributed architecture for concurrent evaluation of programs written in a purely applicative programming language called FGL [9]. This architecture has been evaluated initially by running a simulator on a conventional, sequential computer. Several aspects of the distributed architecture that may have a profound effect upon performance are difficult to evaluate in a sequential simulation, however. It is particularly hard to obtain good estimates of the dynamic behavior of task scheduling and of the degree of concurrency that could be achieved if execution took place on an actual distributed architecture. One experiment presently being programmed for the Multicomputer is a distributed interpreter for the FGL language, and an essential component of the interpreter is a scheduler that distributes tasks among the P-processors in the network.

The process of evaluation of a functional program can be viewed in terms of dynamic transformation of a function dependency tree (FDT), in which leaf nodes represent values and interior nodes represent applications of user defined functions. Under one evaluation scheme, the computation of a function application node may proceed when all of its subordinate nodes are either leaf nodes (corresponding to primitive values), or have been previously evaluated. These computations may take place concurrently if processing resources are available. One of the beauties of the applicative method is that such concurrency will synchronize itself automatically.

To illustrate, Figure 8 assumes that $p(r,s)$ is a function defined in terms of functions f, g and h, and that evaluation of an application of p to arguments x and y is required as a part of a larger program. Attempting to evaluate this application induces a transformation of the FDT which reflects its dependency on the subexpressions $g(x,y)$ and $h(x,y)$. The function of the scheduler is to map this FDT tree dynamically onto the physical processor tree in a manner that yields a good speedup for the application on the Multicomputer. A simple, direct scheme will be to map the FDT so that its subtrees map onto subtrees of the physical processor tree. This scheme offers the advantage that a task's subordinates always reside either on the same processor or on an immediately subordinate one, and communication overhead is minimized. A disadvantage of this strategy is that if two tasks assigned to sibling processors require widely differing amounts of computation, the entire subtree rooted at the lightly loaded processor may wait idle for a significant amount of time while the other subtree computes. Such "unbalanced" task trees could result in poor average utilization of the available processors.

The following strategy attempts to alleviate the unbalanced loading problem. At any processor, instead of assigning tasks to free subordinate processors as soon as possible, distribution is delayed, and some of the demanded tasks are evaluated on the same processor. This may result in the generation of further tasks. A list of demanded tasks is maintained, and whenever the size of this list increases beyond a predetermined threshold value, extra tasks are distributed to subordinate processing nodes. Meanwhile, the parent processor continuously attempts to generate more tasks to maintain the size of the task list above the threshold value. If a subordinate processor evaluates its assigned task well ahead of other siblings, another available task from the task list is immediately assigned to it. The expectation is that this "delayed distribution" strategy will improve processor utilization with unbalanced task trees.

An experimental scheduler incorporating this scheme has been implemented with a two-fold objective:

1) To serve as a means of testing the effectiveness of delayed distribution for a variety of problems with varying threshold values. If good processor utilization is obtainable for a variety of application situations representing varying degrees of load imbalance and a range of computation and communication requirements, this can then form the basis for the scheduler of a distributed FGL interpreter.

2) To create an environment in which application programs for the Multicomputer could be written in the programming language "C" (enhanced with a higher order mapping function and with restrictions on use of globals).

A higher order "map" function permits the user to specify parallel evaluation of several functions and/or simultaneous evaluation of the function for different argument values. The form of the system call is:

```
map(n,
        f1,arg1,len1,res1,
        f2,arg2,len2,res2,
        ..
        fn,argn,lenn,resn);
```

where

arg1 - is an array; arg1[0], arg1[1], ..,arg1[len1-1] are
    arguments to the "len1" parallel invocations of
    function f1.

res1 - is the result array; the result of f1(arg1[0]) is
    to be put in res1[0], f1(arg1[1]) -> res1[1], etc.

This specifies the parallel evaluation of functions f1,f2,...,fn with the specified argument and result locations. The functions f1,f2;..,fn have to be defined as procedures of the form:

```
f1(arg,res)
struct argtype *arg;
struct restype *res;
{
  ..
}
```

The scheduling strategy of deferred task distribution has been partially evaluated in a recent experiment with the Multicomputer. Nodes of the P-tree of the Multicomputer were configured in two ways, as balanced binary trees of depth two and of depth three, as shown in Figure 9. The experiment involved running the same application on both configurations, using deferred task distribution with task distribution thresholds varying from two up to nine. At a threshold value of two, each newly demanded task is allocated to a subordinate node as soon as possible, while at a threshold of nine, a superior processor attempts to maintain a list of eight demanded but unevaluated tasks before distributing excess tasks to subordinates. The variables measured in the experiment were overall processing time and average processor utilization factor.

Fig.10 shows the experimentally measured speedup for a tree-search problem on the Multicomputer. The "balanced problem" involved the search of a complete tree, while the corresponding "unbalanced problem" involved the search of an incomplete tree of the same fanout and depth. The incompleteness resulted in an imbalance of compute requirements of different branches of the tree at various points in the search.

From the results, the following observations may be made:

1) The "balanced" case shows no increase in speedup resulting from the use of delayed task distribution for either the 3-processor or the 7-processor configuration. In fact, the 7-processor case shows a significant loss of speedup with increasing threshold.
2) The "unbalanced" case displays a peaking behavior of the attained speedup. As the threshold for task distribution is increased, there is an increase in speedup; this is followed by a decrease as threshold is further raised. The decline in speedup beyond the peak is more rapid for the 7-processor case.

No speedup is to be expected in the "balanced" case, since very uniform distribution and good speedup should result without having to resort to delayed distribution; increased thresholds merely result in greater idle times for subordinate processors. The greater deterioration of speedup for the 7-processor case seems to result from poorer utilization of leaf processors at high thresholds. The leaves of the 3-processor network experience idle delay during task list build-up at their superior processor only once; after the initial build-up of tasks at the superior, it continuously generates tasks while the leaves compute on their tasks. In the 7-processor case, however, the superiors of the leaf processors are themselves subordinate to the root processor, and are started off with a fresh task as soon as they complete their current task. Each time the processor is started off with a new task, it must build up its task list afresh, and the subtree below it is idle for that duration. Thus the leaf processors accumulate a significant amount of idle time. This also explains the peaking behavior of the unbalanced case. The balancing effect of delaying distribution results in an initial rise in speedup. At higher thresholds, the poorer utilization of the leaves results in a decrease in average utilization, and hence of attained speedup. The observed results suggest better performance can be attained through the incorporation of a "pipelining" mechanism into the scheduling scheme, so that generation of the task list of a succeeding task is done in parallel with the evaluation of a preceding one.

The continuation of these experiments and the incorporation of the results into the design of the distributed FGL interpreter for the Multicomputer will be reported on at a later date. Other measurements in conjunction with the FGL interpreter relate to the volume of message traffic generated, the frequency with which task suspensions occur, and a distribution function of their lifetimes, as well as concurrency measures and a measure of the performance of a distributed garbage collector. Because communication events in the system can be monitored with a high degree of temporal precision, it should be possible to scale performance data to the channel capacity of communications links, as well as to processor and memory capacities. This will provide data for systems designers that would be extremely expensive to obtain by the alternative means of sequential simulation.

Another experiment being planned for the Multicomputer views compilation as a pipeline process, in which procedure-sized packets of program are passed through various phases of translation. The processing stages will be realized by tasks (processes) dynamically assigned to nodes of the Multicomputer, and the pipes connecting these stages will be realized by use of the local file systems (T-systems). The purpose of the experiment will be to assess the feasibility of using a scheduled pipeline for processes that involve a high degree of variability in the per-stage processing time. Again, the ability to account specifically for the cost of moving data images will enable the results to be normalized to systems whose design parameters differ from those used in the Multicomputer.

Other types of experiments for which the Multicomputer is suited include performance measurement of higher-level protocols by distributed simulation, performance measurement of transaction systems, and investigation of rates of convergence of distributed voting algorithms.

7. Acknowledgement

Many persons have been involved with the design and implementation of the Stony Brook Multicomputer, over a period of more than three years. In particular, we wish to thank Devesh Bhatt, who did most of the hardware design for the control links, Andrew Chang, who worked on the P-node communications kernel, on file protection and process management, Arch Harris, who contributed to the original concept and did system simulations, Mahmoud Hamza, who has developed both the hardware and software for the real-time event monitor, and Roger Lam and Brad Nohejl, who implemented the basic internode communications software.

8. References

[1] Harris, J.A. and Smith, D.R., "Simulation experiments of a tree-organized multicomputer", *Proc. 6th Sympos. on Computer Architecture*, IEEE, April, 1979.

[2] Kieburtz, R.B., "A hierarchical multicomputer for problem-solving by decomposition", *Proc. 1st Internat. Conf. on Distributed Computing Systems*, IEEE, Oct., 1979, pp. 63-71.

[3] Mukerji, J. and Kieburtz, R.B., "A distributed file system for a hierarchical multicomputer", *Proc. 1st Internat. Conf. on Distributed Computing Systems*, IEEE, Oct., 1979, pp. 448-458.

[4] Kieburtz, R.B. "A distributed operating system for the Stony Brook Multicomputer", *Proc. 2nd Internat. Conf. on Distributed Computing Systems*, IEEE, Apr. 1981, pp. 67-79.

[5] Mukerji, J. "Design of a distributed file system for a hierarchical multicomputer", Ph.D. thesis, Dept. of Computer Science, SUNY at Stony Brook, April, 1982.

[6] Hamza, M., and Smith, D. R., "Design of the performance monitor for the Stony Brook Multicomputer", submitted to IEEE Transactions on Computers.

[7] Lindstrom, G., "Alpha-beta pruning on evolving game trees", Tech. Rept. UUCS-79-101, Dept. of Computer Sceince, Univ. of Utah, 1979.

[8] Keller, R.M., Lindstrom, G., and Patil, "A loosely-coupled applicative multiprocessing system", *AFIPS Conf. Proc.* vol. 40, 1979 NCC, June, 1979, pp. 613-622.

[9] Keller, R.M., *et al.* "FGL programmers guide", Dept. of Computer Science, Univ. of Utah, Feb., 1980.

[10] Wulf, W., *et al.* "HYDRA: the kernel of a multiprocessor operating system", *Comm. ACM* vol. 17, June, 1974, pp. 337-345.

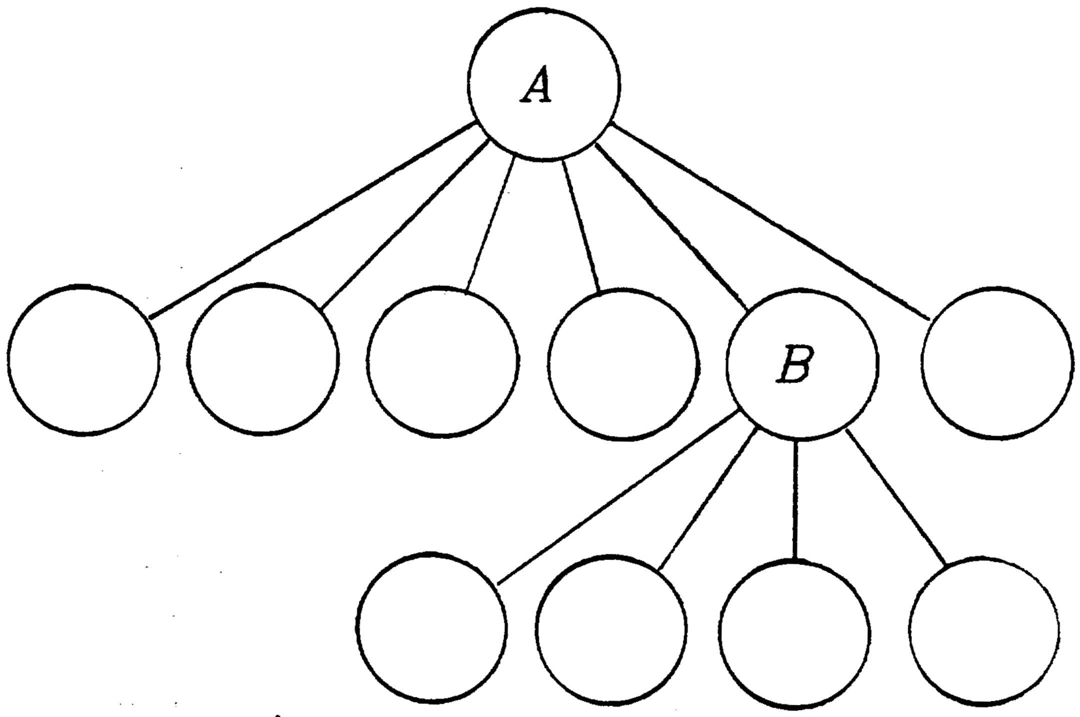


Figure 1(a) -- A system of processes which forms a rooted hierarchy. The node labelled A is a control process. Node B is a the control process of a subsystem.

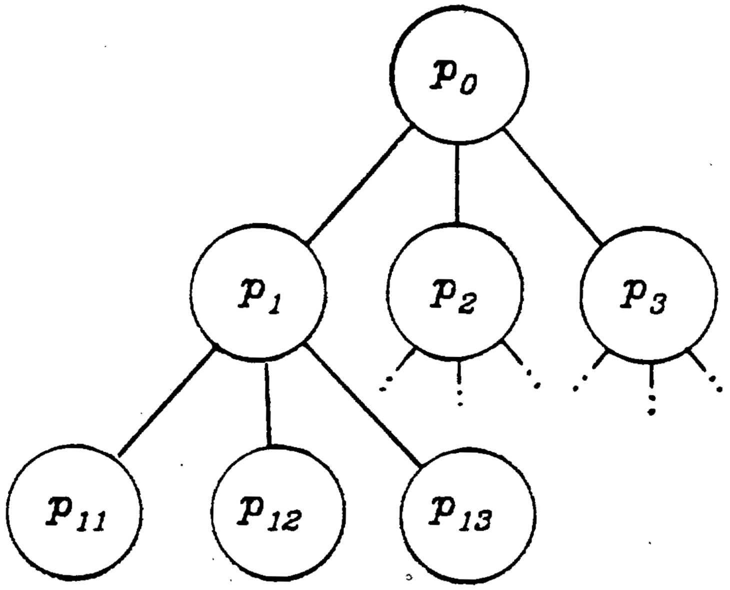


Figure 1(b) -- A processor tree. Nodes represent processors with local memory. Arcs represent communications links for control messages.

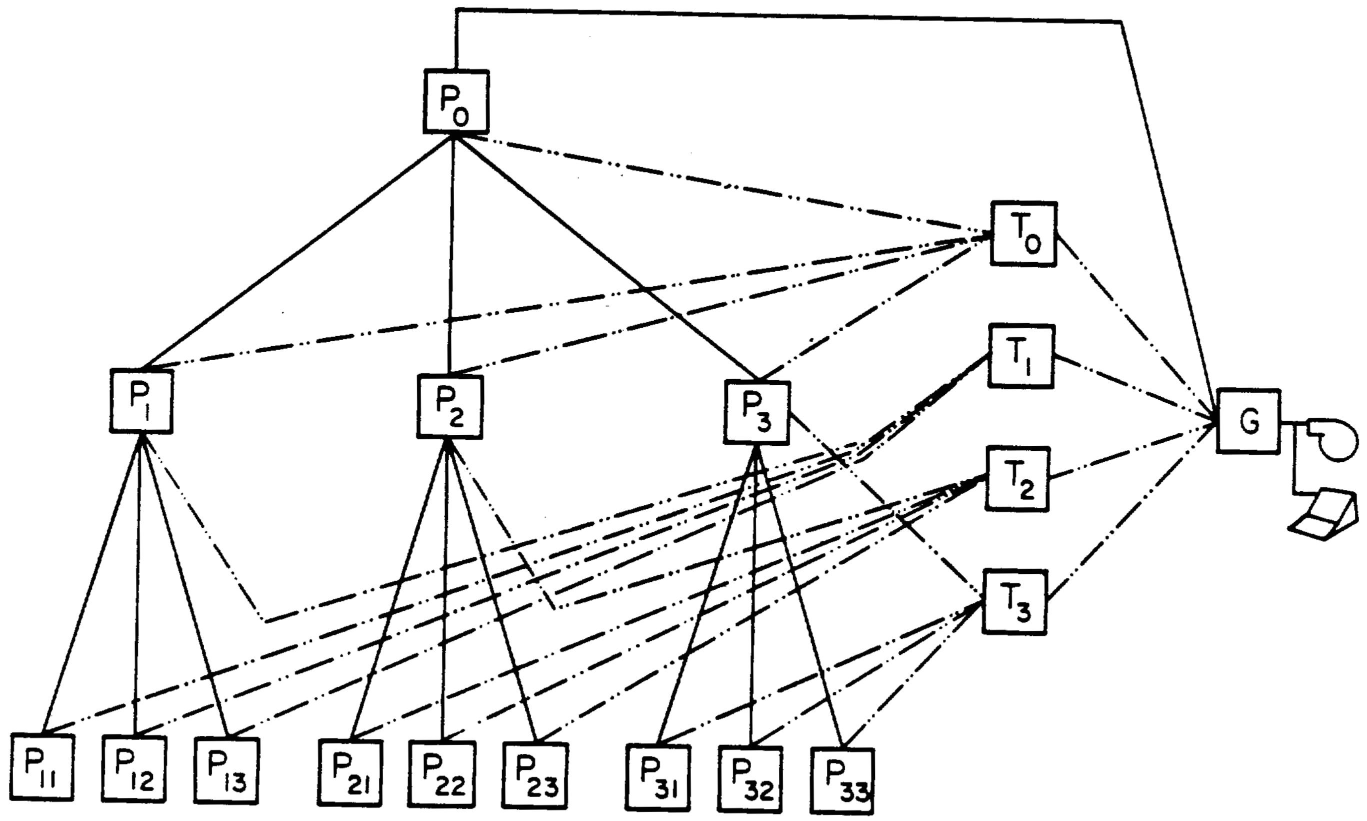


Figure 2 — Interconnection architecture of the Multicomputer. Control and applications processes are run on the processing nodes labelled P_i. File systems are implemented at the T-nodes and the G-node. Solid arcs, called control links, are used for short control messages. Dashed arcs have DMA capability, and are used for transport of data and code segments.

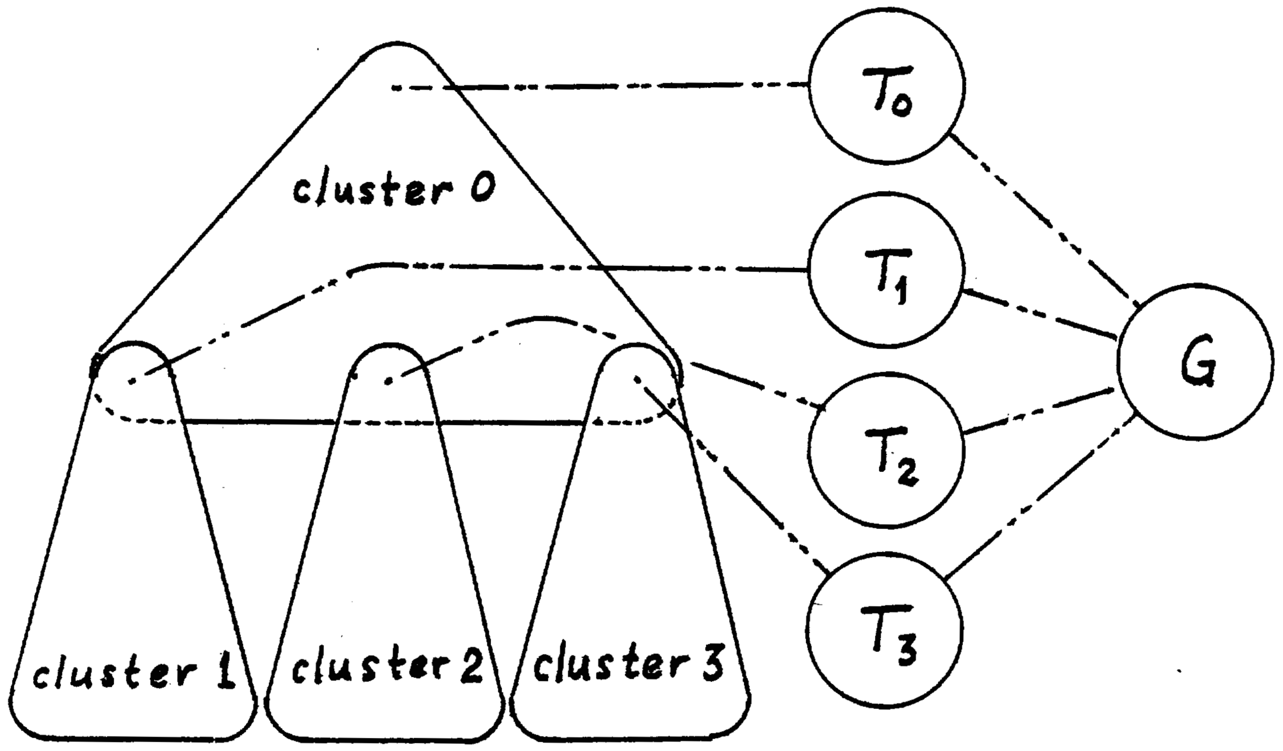


Figure 3 – Processor clusters lie at the leaves of the file systems hierarchy.

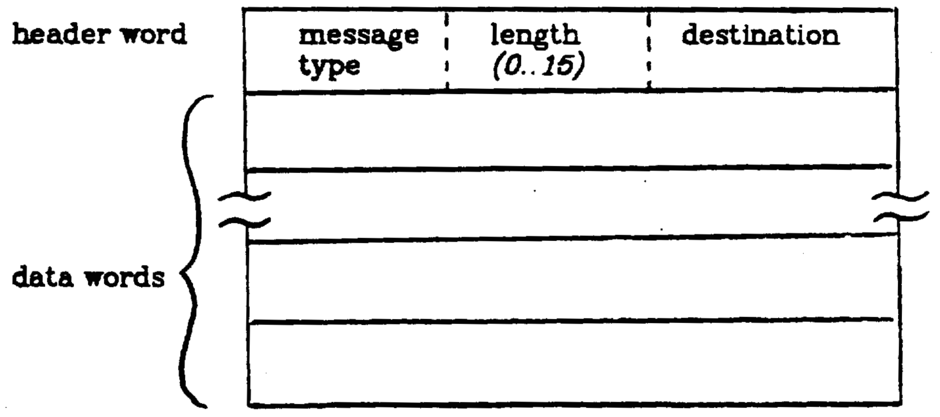


Figure 4 -- Format of a control message. The length of a control message is variable, up to a maximum of 15 data words. The destination field of a header carries a channel identifier.

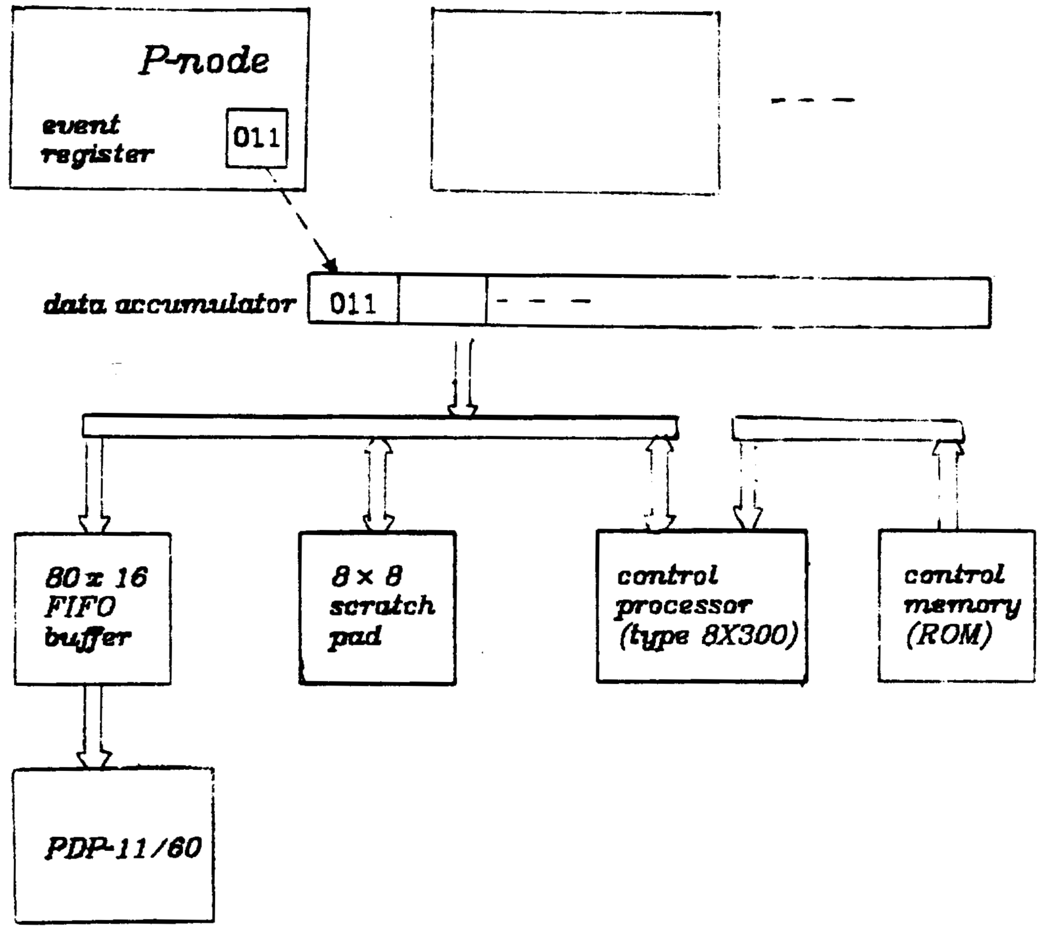


Figure 5 – Block diagram of the real-time event monitor, showing connection of the event register at one P-node.

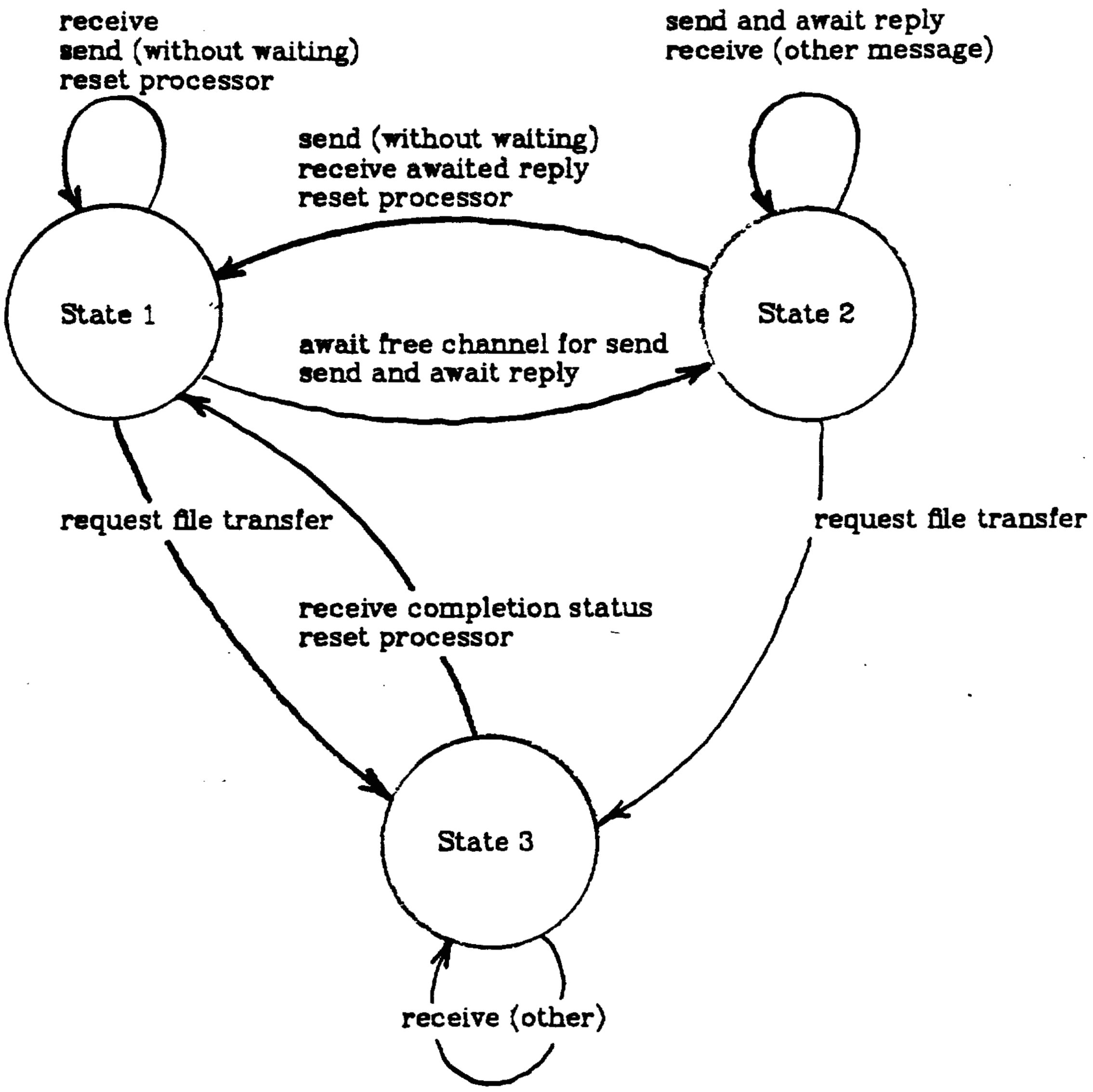
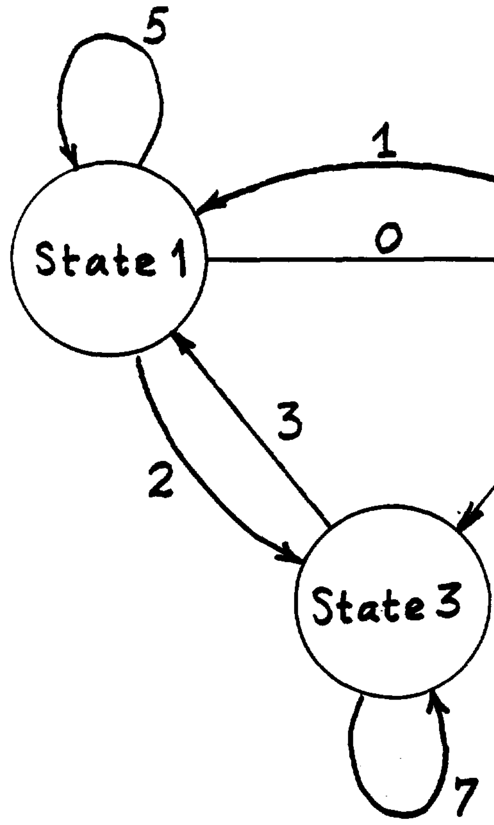


Figure 6 -- Activity states of a process running at a P-node. Arcs are labelled by the names of events that induce state transitions. In state 1 the process is active; in state 2 it is awaiting a reply from a message previously sent, or is awaiting a free channel on which to send a message; in state 3 it is awaiting notice of completion of a requested file operation. Every event except "send and await reply" corresponds either to an entry point of the P-node kernel of GOSSIP or to an entry point of the interrupt handler.



Code	Transition	Events
0:	1 → 2	attempt to send, channel busy / send and await reply
1:	2 → 1	send (without waiting) / receive awaited reply / reset
2:	1 → 3	send request for file transfer
3:	3 → 1	receive notice of file transfer completion / reset
4:	2 → 3	send request for file transfer
5:	1 → 1	receive message / send (without waiting for reply)
6:	2 → 2	receive unawaited message / send and await reply
7:	3 → 3	receive (other than file transfer completion notice)

Figure 7 -- An encoding of process activity state transition events

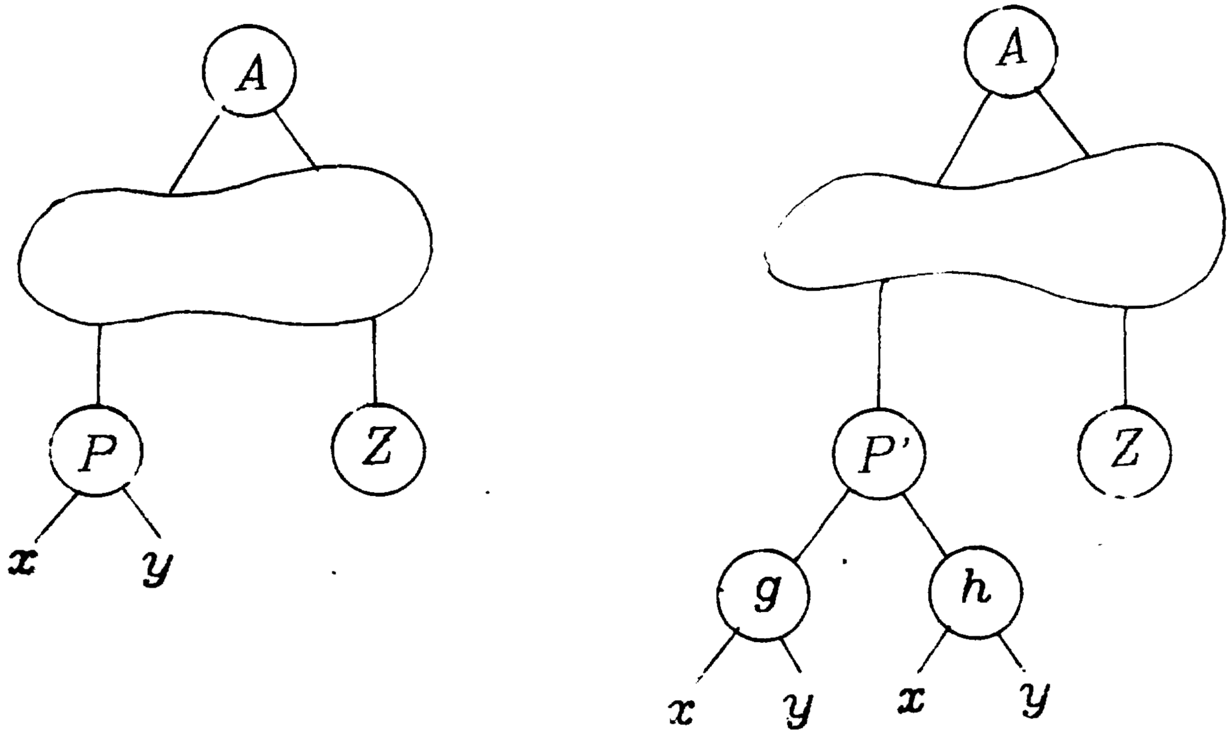


Figure 8 — A simple example of a functional program *A*, defined in terms of functions .. *P*, *Z*. *P* is itself defined in terms of user-defined functions *g*, *h*, which in turn are defined in terms of *x*, *y*, assumed to use no further user-defined functions in their definition. As the program dependency tree (PDT) evaluates, the definition of *P* is first invoked, causing evaluation of *g*, *h*. The PDT reflects this and *P* is changed to *P'*, signifying a partially evaluated function.

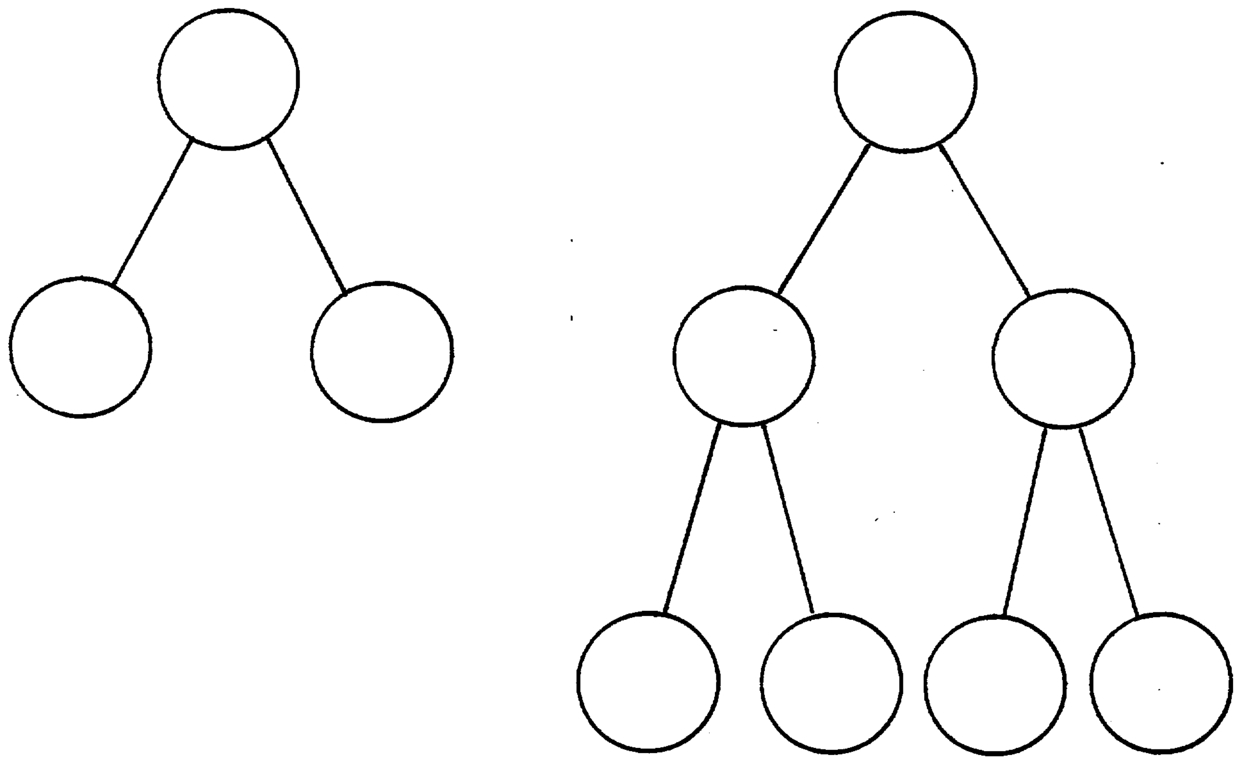


Figure 9 -- Two configurations of the P-tree of the Multicomputer

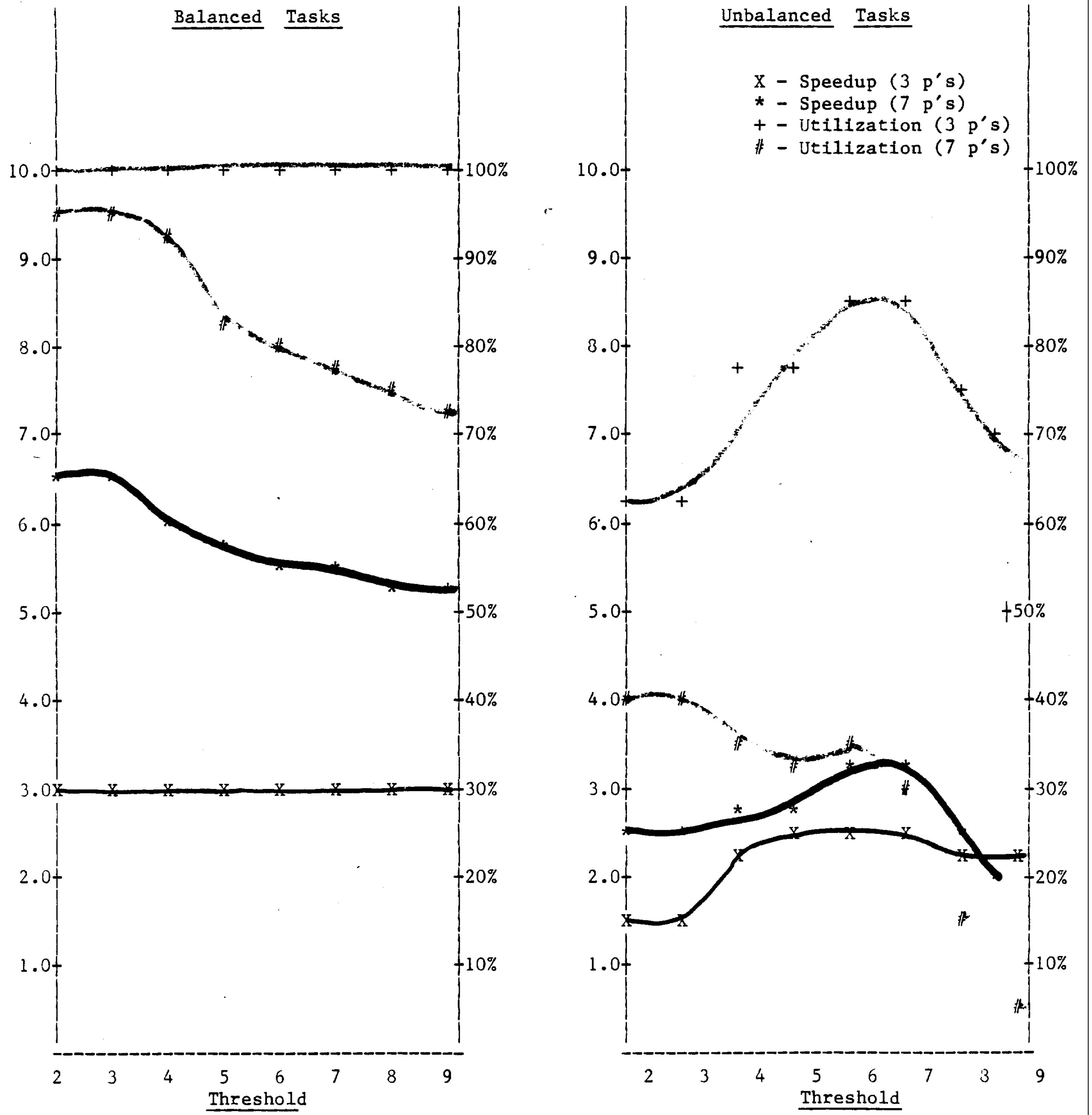


Fig.10 Variation of speedup/leaf utilization with threshold value.