THE DESIGN OF AN EMULATOR FOR A PARALLEL MACHINE LANGUAGE

by
Victor R. Lesser*
Computer Science Department
Carnegie-Mellon University

ABSTRACT

A paradigm is developed for structuring a complex emulator operating in a parallel hardware environment. This paradigm is based on the view that a complex emulator is best structured as of a set of microprocesses, each performing a small independent task, that interact in a closely-coupled manner. This is in contrast to the conventional method of structuring an emulator as a set of subroutines with a sequential flow of control among them. The design of an emulator for a parallel machine language (i.e. Adam's Graph Machine Language) using the new paradigm is discussed in detail, including the dynamic execution characteristics of the emulator in a parallel hardware environment. The analysis indicates that, given an appropriate microcomputer architecture, this structuring allows an emulator for a parallel machine language to be naturally and compactly coded and to fully map parallelism at the emulated machine language level into parallelism at the hardware level. In particular, it has been shown that an emulator can be structured so as to utilize well more than sixteen identical microprocessors. In addition, the emulator uses the idea of tailoring an emulator's control structure both to the emulated machine language and **dynamically** to the **specific** program being emulated.

I. INTRODUCTION**

The conventional structure of an emulator is a series of subroutines [TUC65, ROS69]. A typical decomposition of an emulator consists of a "control" subroutine that fetches the next instruction to be executed, a "decode" subroutine that determines the opcode of the next instruction to be executed and computes the effective address of the data operands, and a set of "semantic" subroutines which perform the operations specified by the opcode of the emulated instruction. This structure is appropriate for emulating a machine language which has a sequential control structure on a microcomputer which is not capable of asynchronous parallel operations. However, if either of these assumptions is relaxed the above decomposition of the emulation process is overly restrictive: in the framework of a subroutine control structure 1) the imbedding of the parallel control structure of the emulated machine is complex and inefficient, and 2) the parallel resources of the microcomputer cannot be effectively utilized.

The limitations of this subroutine approach to structuring an emulator will become an increasingly severe problem if the current trends continue towards more complex programming languages, language-directed machine design (i.e. the development of **intermediate machine languages (IML)** appropriate for execution of programming languages [MCK67, WIL72]), and highly parallel computer organizations. As programming languages become more complex in their control structures (e.g. parallelism, co-routines, monitoring, etc.), these complexities will be reflected in the control structures of the IML. In addition, there will be a desire to fully exploit the parallelism of the hardware 1) by imbedding, in the IML, control primitives that directly invoke hardware parallelism and 2) by using hardware parallelism to speed up the emulator of the IML.

To overcome the limitations of the subroutine approach to structuring an emulator for a parallel IML, an alternative structuring paradigm has been formulated. This paradigm decomposes an emulator into a set of microprocesses**, each performing a small independent task, that interact in a closely-coupled manner.

The microprocesses interact often, and therefore the their interaction patterns must be kept simple in order to minimize overhead. Further, there is no single interaction pattern (e.g. communication through a global shared memory, or ports, or message queues, etc.) that is most efficient and natural for all emulators, or even for a particular emulator. Rather, the interactions must be designed for each emulator.

This decompostion paradigm is similar in approach to those used in the design of sophisticated computer organizations such as the IBM 360/91 [AND67], CDC 6600 [THO64], and the SYMBOL machine [RIC71]. The internal organization of each of these computers consists of a set of independent, asynchronous modules: 1) the IBM 360/91

* This research was carried on while the author was with the Department of Computer Science, Stanford University, Stanford, Calif. and was partially supported under AEC contracts AT(04-3)326,P.A.23 and AT(04-3)515 Stanford Linear Accelerator Center, Stanford, Calif.

**This paper presents material from the third cnapter of [LES72].

** The relationship of a microprogram to a microprocess is analogous to the relationship (as described in [LAM68]) of a program to a process.

is structured so there are separate modules for the control, decode, and semantic phases of an emulation; these modules are interconnected and interact in a pipeline manner; 2) the CDC 6600 is structured so that there is a separate module for each "semantic subroutine"; these modules are organized so that the semantic phase of many instructions may be performed concurrently; and, 3) the SYMBOL machine, which directly interprets a higher-level language, is structured so there are separate modules for compilation, memory-management, execution of compiled code, etc.; each of these modules is capable of performing concurrently its task on a different user program.

Just as the level of a virtual machine language (i.e. an IML) was introduced into the interpretation process to more efficiently execute higher level languages, it is felt that to construct efficient and natural emulators for parallel IMLs a new level of variablity must be introduced into the emulation process. This new level, which is called the **virtual process-memory-switch** [BEL70] **environment**, allows the designer to structure an emulator in terms of an arbitrary configuration of microprocesses and their associated interaction patterns without directly dealing with how microprocess activity is mapped into microprocessor activity (see Figure 1).

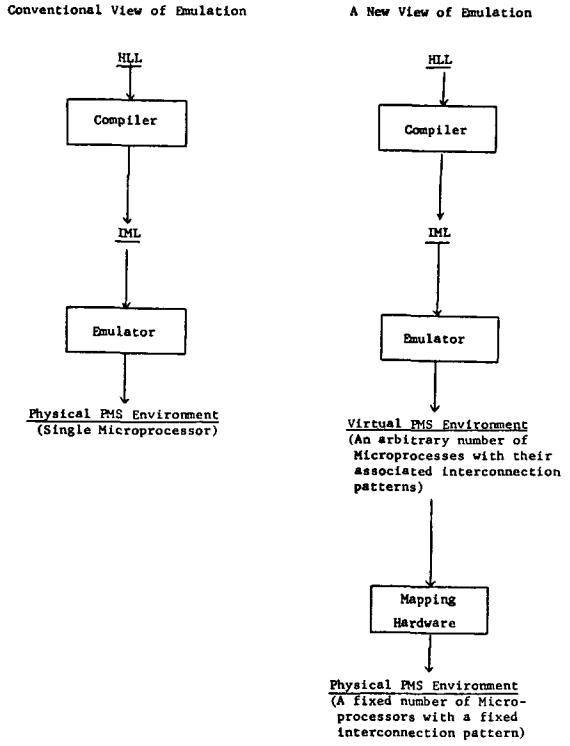


Figure 1. A New View of Emulation

A parallel microcomputer architecture that implements the concept of a virtual PMS has been developed by adding a new level of hardware control which performs the mapping of microprocess activity into microprocessor activity [LES72]. This control level can be thought of as a simple hardware operating system which manages the scheduling of and interactions among microprocessors. Microprocess activity is specified through a data base called the **control data structure** (CDS). The CDS consists of a collection of state vectors; each state vector defines a microprocess. A particular set of interaction patterns among microprocesses is dynamically defined by varying the number of and relationships among these state vectors. In a conventional computer the analogous structure for control contains a fixed set of data elements (e.g. program counter, interrupt register, etc.) whose relationships are predefined. Thus, in a conventional system, control can be modified only by changing the value of data elements in the CDS (e.g., by changing the program counter). The ability added here to modify the syntax of the data structure for control is the key to creating the virtual PMS environment appropriate for a particular IML emulator.

The remainder of the paper discusses a case history of the design and emulation of an IML for a highly parallel programming language in the framework of the microcomputer architecture discussed above. In particular, the following topics are discussed: 1) the structure of the parallel programming language to be emulated and the conceptual problems in designing its IML, 2) the decomposition of the emulator into a large set of small, independent tasks (microprocesses), 3) the specification of the interaction patterns among these microprocesses, and, 4) the dynamic execution characteristics of the emulator as the number of microprocessors is varied.

II. Adam's Graph Machine Language

The parallel machine language that was chosen to be emulated is based on an asynchronous parallel programming schema (language) developed by Adams [ADA68] called the Adams' Graph Machine Language (**AGML**). The AGML was chosen as a test case not based on its practicality as a machine language but rather because its emulator can be designed to employ the following control structure concepts: distributed parallel control, pipelining, recursion, finite resource scheduling and message queueing.

The AGML is based on a data flow model [KAR66, ROD67] for representing the sequencing aspects of a computation. The instructions of the AGML can be thought of as nodes of a graph; the nodes are connected to each other through links which are FIFO (first-in - first-out) queues. These links are uni-directional data paths in which one terminal point of the link is denoted as an output link of a node while the other terminal point is denoted as an input link of a node. An instruction (node) is executed when each of its input links contains a data item; a node executes by removing the input data from each of its input links, performing a calculation on these data, and storing the output of the calculation on zero or more of the output links. After the node has stored the results of the calculation on its output links, the node can be re-executed when each of the input links again has data items. An example of a graph program is shown in Figure 2.

The data flow model for sequencing allows the implicit expression of parallel activity because if there exists no data dependencies among a group of nodes, then these nodes may be executed simultaneously. For example, the two multiplication nodes in Figure 2 can be executed simultaneously whereas the plus node must await the completion of both multiplication nodes. A data flow model can also be thought of as a distributed control system since each node can independently decide, based on local information, whether it can execute.

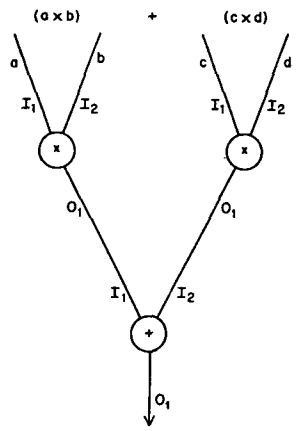


Figure 2. A Simple Graph Program

There are three types of nodes in the AGML: parallel, procedure, and sequential nodes. The **parallel node** allows for the expression of pipeline (vector) parallelism. The parallel node is defined so that it may be immediately re-executed rather than waiting for the computation to complete. This re-execution of the parallel node can occur once the input data items have been removed from their links provided there is another set of input data items on the input links. Thus, multiple instances of a node may be concurrently executing, giving the effect of a pipeline. The **procedure node** allows for the expression of recursive control structures. The procedure node, instead of invoking a primitive arithmetic operation, causes the invocation of a graph procedure. The input parameters of the invoked graph procedure are the input data items of the procedure node. The invoked graph procedure may, in turn, contain procedure nodes, thus leading to a recursive, parallel control structure. Finally, the **sequential** node allows for the expression of data-dependent sequencing of the graph, e.g., loop-control, etc; it is defined so that its semantics are affected by its previous execution history. In particular, the execution history is used to select, for the next execution of the sequential node, a subset of its input data links from which input data will be accepted.

A graph procedure is terminated when there is a data item on each of the external output links. An external output link is a link of the graph procedure that is not connected as an input link to any node in the graph procedure. These external output links are used to transmit the output of the graph procedure to the procedure node that invoked the graph procedure. This termination condition differs from Adam's original formulation which is based on all nodes of the graph procedure being inactive.* This new termination condition was introduced so that the AGML could be emulated in a highly parallel manner. Monitoring for Adams' original formulation of the termination condition is very difficult to do in a highly parallel manner. This is especially true if no assumptions are made about the actual number of physical microprocessors. In essence, the monitoring process overlays the highly parallel distributed control structure of the graph machine with a control structure which requires sequential accessing of a shared, global data base. In contrast, the monitoring process for the new termination condition does not affect the basic distributed control nature of the AGML. In fact, the process of monitoring for this new termination condition is precisely the same as the process of monitoring for whether a node is ready to execute.

* The new termination condition makes the AGML only output-determinate rather than completely determinate.

III. An Emulator for the Adams' Graph Machine Language

The main emphasis in the design of this emulator has been the exploitation of the implicit parallelism of an AGML program. The exploitation of this implicit parallelism is not worthwhile when node operations are simple arithmetic operators (e.g., +, *, etc.) because the overhead costs of determining whether there is parallelism and then invoking parallel microprocessors is significant with respect to the computation performed by a node. However, the AGML control structures could just as well be used to sequence nodes whose primitive operations were larger units of computation (e.g., matrix multiply, exponentiation, etc.) in which case the overhead costs of extraction of implicit parallelism would be tolerable. This balance between the computational grain of the most primitive operations of a language that can be parallelized and the overhead cost of emulating the control structures of the language which specify this parallelism is a crucial design parameter in developing and programming parallel languages. The major focus of this paper is, however, not on determining this balance point but rather on the development of techniques for compactly and simply coding emulators which exploit the parallelism of a parallel IML.

The AGML emulator exploits the parallelism of an AGML program by:

1) making parallel, whenever possible, the overhead functions required to sequence an AGML program;

2) dynamically tailoring the CDS, not only to the structure of the AGML emulator, but also to the structure of the **specific** AGML program to be emulated.

This tailoring of the CDS for a specific graph program is accomplished by creating a distinct control structure for sequencing each node of the graph. This control structure for sequencing each node is tailored to the particular type of node and the node's input and output requirements. Thus, the CDS for the AGML emulator closely mirrors the distributed control structure of the particular AGML program. In addition, the CDS may be dynamically modified during the execution of a graph program so as to take advantage of the potential parallel activity that is generated when a graph procedure is dynamically invoked.

III.1 The CDS for AGML Emulator

The crucial aspect of the design of the emulator is the specification of the emulator's CDS. The CDS provides a syntactic framework within which the emulator can be conveniently microcoded. The CDS of the AGML emulator

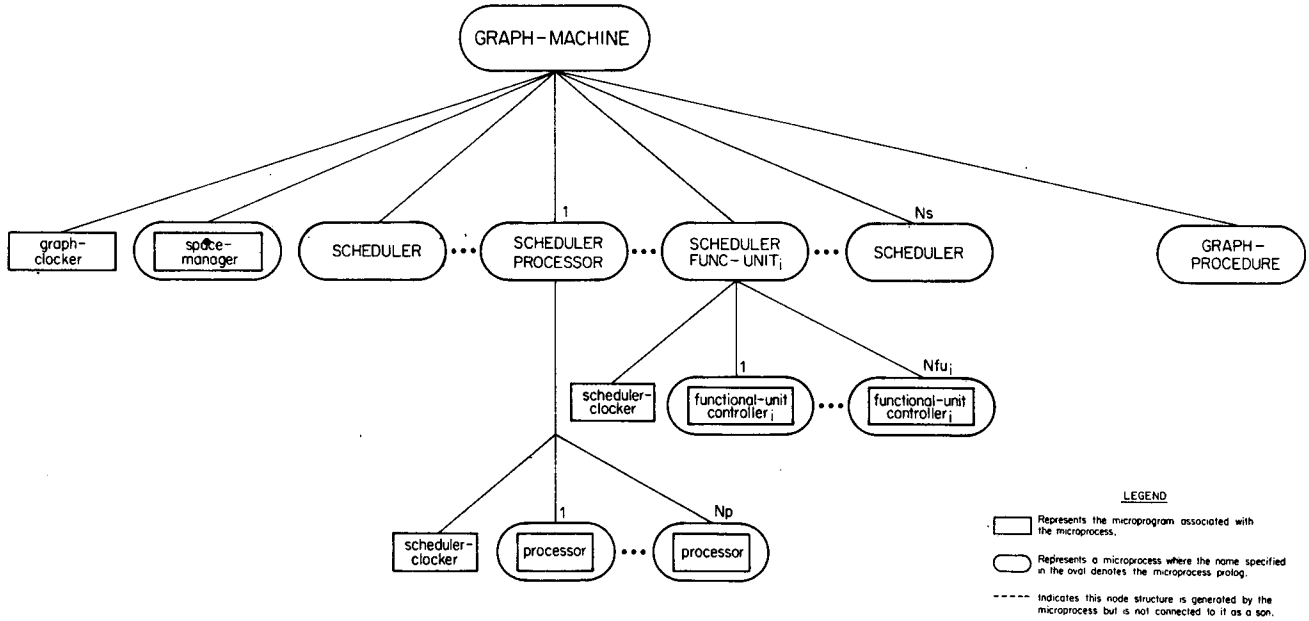


Figure 3a. Control Data Structure for Resource Management

can be thought of in terms of two parts: 1) a CDS for resource management (e.g., the dynamic allocation of memory for link queue space) and 2) a CDS for sequencing of a graph program. These two parts of the emulator's CDS form a two level hierarchy in which the CDS for resource management is at the top level. The CDS for resource management is implemented as a fixed structure which is independent of the particular AGML program being emulated, whereas the CDS for sequencing of the graph program has a dynamic structure which is dependent on the particular graph procedures currently being executed.

III.1.1 The CDS for Resource Management

The CDS for resource management is pictured in Figure 3a. The resource management functions are carried out by the SPACE-MANAGER, and SCHEDULER(1) ... SCHEDULER(Ns) microprocesses. The SPACE-MANAGER microprocess dynamically allocates fixed length blocks of storage for link queue space. This storage allocation cannot be done statically since graph procedures can be dynamically invoked during the execution of a graph program. In addition, there can be many graph procedures which are simultaneously requesting storage for their links. Thus, the storage allocation has to be done dynamically in a central place. The SPACE-MANAGER microprocess, by appropriate manipulation of its execution-state, can sequentialize the acceptance and processing of communications which either request the allocation of storage or specify the release of previously allocated storage.*

The scheduling function of the resource manager is implemented through a set of SCHEDULER microprocesses such that each type of primitive node operation could conceivably have its own SCHEDULER microprocess. A SCHEDULER microprocess is used to assign, depending upon the type of operation, either a functional unit or a microprocess to carry out the primitive operation of a node. Each SCHEDULER microprocess has a fixed length queue to hold requests for a device (i.e., functional unit or microprocess) that cannot be currently honored. If this queue becomes full, then the SCHEDULER microprocess employs the "waiting" execution state that permits selective listening rather than the suspended execution state. In this selective listening state, a communication to the SCHEDULER that requests a device is not consummated, whereas a communication to the SCHEDULER that specifies the termination of a device is consummated.

* The microcomputer's interprocessor communication primitives allow for the specification of different classes of communication. A microprocess, through manipulation of its execution state, can specify which class (possibly no class) of communications it will currently accept. Through this mechanism, a microprocess can 1) sequentially accept and process multiple communications, 2) selectively accept only certain types of communications, and 3) asynchronously accept requests for communication.

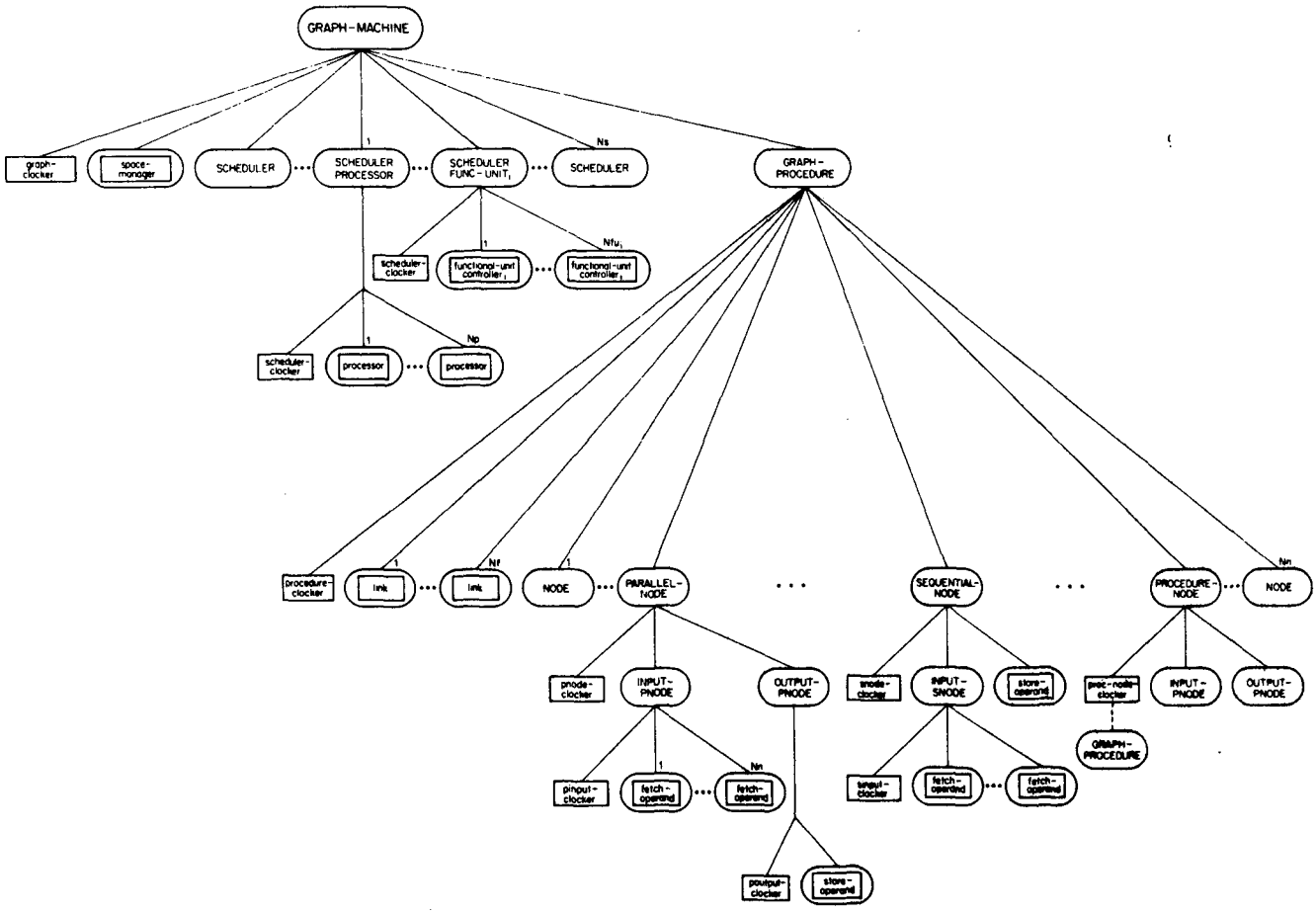


Figure 3c. Control Data Structure for AGML

procedure, initializes and allocates storage for the links of the graph procedure, and monitors for the termination of the graph procedure. The CDS for sequencing of a graph procedure, as previously discussed, is tailored to the particular graph procedure being emulated. The template for a tailored CDS is pictured in Figure 3b. This tailored CDS contains, for each link and node of the graph procedure, a corresponding LINK microprocess and NODE microprocess, where there are three types of NODE microprocesses: PARALLEL, SEQUENTIAL and PROCEDURE. For instance, the graph procedure specified in Figure 2 results in a CDS containing 7 LINK microprocesses and 3 NODE microprocesses. This CDS for sequencing of a graph procedure has been designed so that, once generated, its structure need not be modified. Thus, the structure-building overhead is only incurred once and consequently is not a function of the number of node executions. In addition, the generation of the CDS for all NODE microprocesses can be done in parallel.

The LINK microprocess is responsible for retrieving and storing data from a link's queue space and updating the queue pointers. The LINK microprocess acts as a semaphore process for controlling access to the link's queue space. A semaphore process is required for controlling access to a link queue because at the same time, one node may desire to place data on the queue while another node may desire to remove data from the queue at the same time. The LINK microprocess is also used to avoid "busy waiting" when a node desires a data item and the link queue is empty. In this case, instead of the node repeatedly querying the LINK microprocess whether input link data is available, the LINK microprocess accepts a request for data from the node and then, when the data is available, transfers the data to the node which is in a waiting execution-state. Thus, as will be seen in more detail later, the LINK microprocess allows the trigger function of a node (i.e., deciding when a node is ready to execute) to be monitored in a non-busy way. A similar handshaking mechanism is used to avoid a node "busy waiting" until there is room on the link to store output link data. In addition, the LINK microprocess allows the updating of queue pointers to go on in parallel with a node's further processing.

The NODE microprocess implements the following overhead operations required to sequence a node: 1) fetching the input data items from the appropriate input links, 2) deciding when the node operation is ready to be executed, 3) transferring the input data items to the

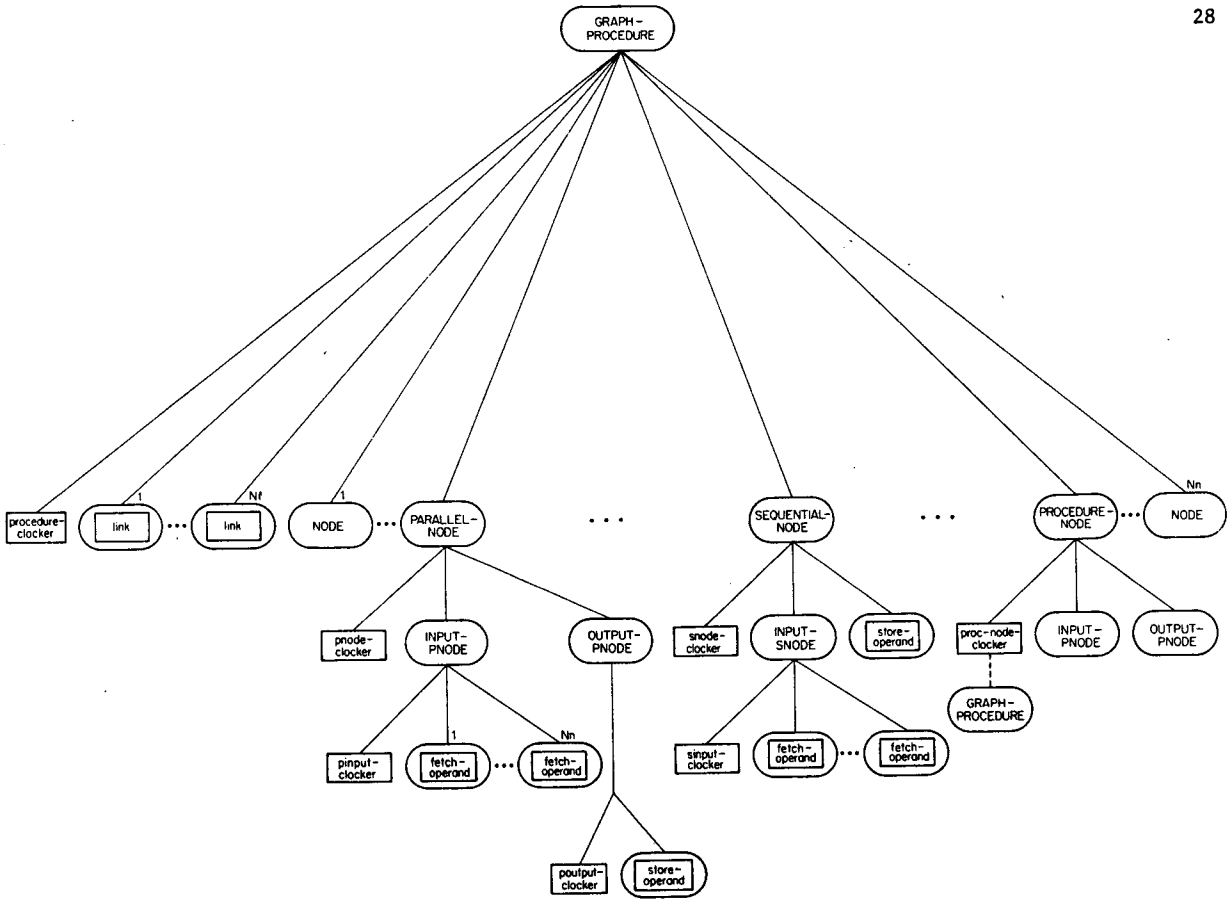


Figure 3b. Control Data Structure for GRAPH-PROCEDURE

The AGML emulator could have been organized without this centralized scheduling function. In essence, the centralized scheduler is scheduling virtual microprocesses which are in turn being scheduled on physical microprocessors by the built-in hardware scheduler. Thus, the emulator could have been organized so as to use the built-in scheduler alone. There are two main reasons for taking the centralized scheduler approach. The first reason stems from the simplicity of the built-in hardware algorithm for scheduling. Specifically, the two level scheduling approach allows the design of a sophisticated graph scheduler which takes into account the structure of the graph procedure so as to utilize available microprocessors** more efficiently [NEL72]. The second reason stems from the semantics of the parallel node that permit the concurrent initiation of an arbitrary number of primitive operations for each parallel node. In order to take advantage of this potential parallelism of the parallel node in a non-centralized scheduling approach either 1) each time a primitive node operation is initiated, the sequencer of a parallel node would have to dynamically generate the state vector of a microprocess to carry out the operation; or 2) the fixed CDS structure of the appropriate SCHEDULER microprocess would have to be duplicated for each parallel node in the graph procedure. Thus, either the structure building overhead involved in sequencing of the graph procedure would greatly increase or the size of the CDS for the graph procedure would greatly increase. On the other hand, a centralized scheduler has a fixed CDS structure which does not vary during the execution of the graph, and there is only one state vector for each device that can be scheduled. For these reasons, a centralized scheduling approach is used.

** The virtual scheduler can query the hardware system to find out the number of physical microprocessors and use this information as a parameter in the scheduling function.

III.1.2 The CDS for Sequencing of a Graph Procedure

The sequencing of a graph procedure is implemented through the microprocess GRAPH-PROCEDURE. This microprocess generates the CDS for sequencing of a graph

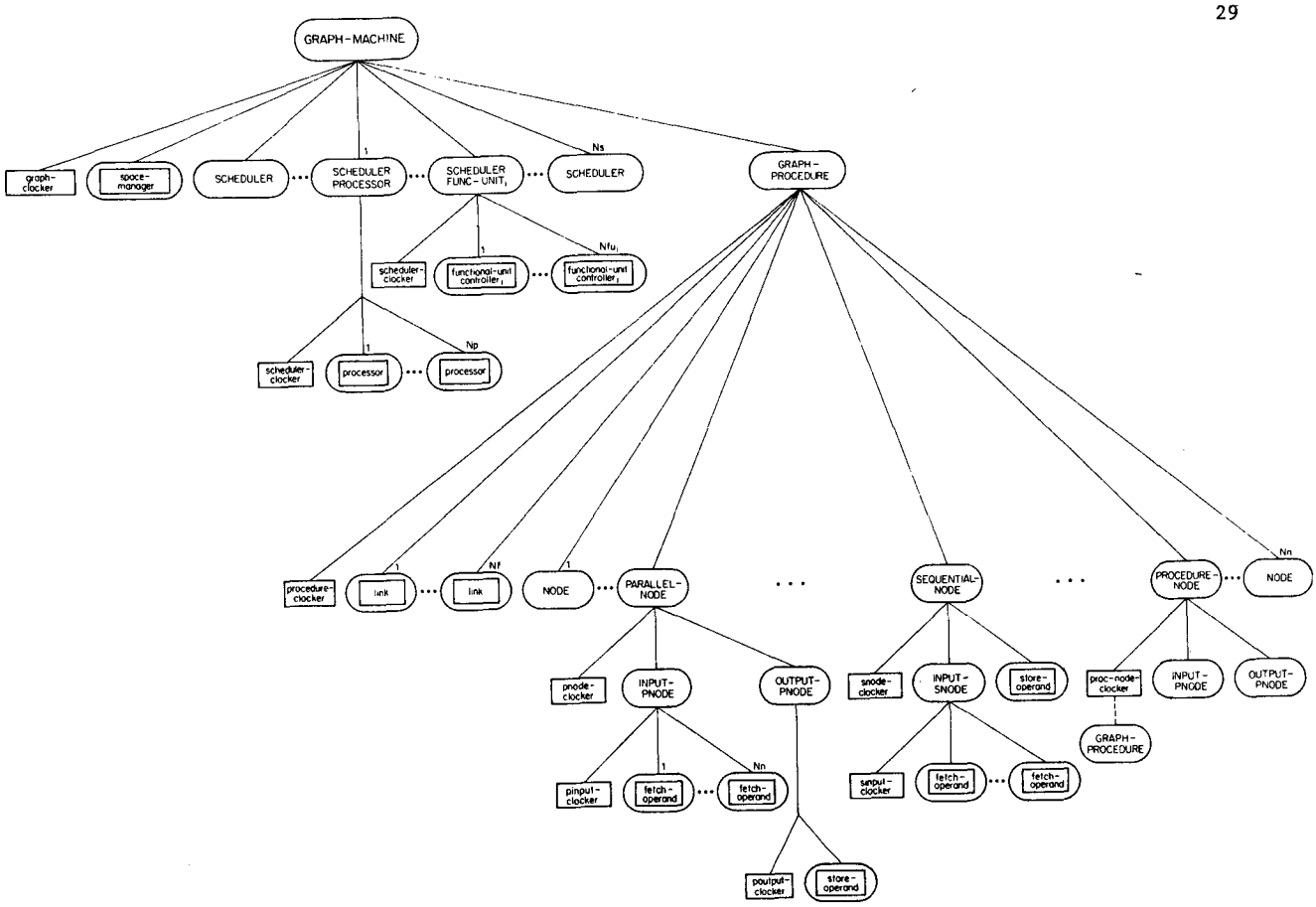


Figure 3c. Control Data Structure for AGML

procedure, initializes and allocates storage for the links of the graph procedure, and monitors for the termination of the graph procedure. The CDS for sequencing of a graph procedure, as previously discussed, is tailored to the particular graph procedure being emulated. The template for a tailored CDS is pictured in Figure 3b. This tailored CDS contains, for each link and node of the graph procedure, a corresponding LINK microprocess and NODE microprocess, where there are three types of NODE microprocesses: PARALLEL, SEQUENTIAL and PROCEDURE. For instance, the graph procedure specified in Figure 2 results in a CDS containing 7 LINK microprocesses and 3 NODE microprocesses. This CDS for sequencing of a graph procedure has been designed so that, once generated, its structure need not be modified. Thus, the structure-building overhead is only incurred once and consequently is not a function of the number of node executions. In addition, the generation of the CDS for all NODE microprocesses can be done in parallel.

The LINK microprocess is responsible for retrieving and storing data from a link's queue space and updating the queue pointers. The LINK microprocess acts as a semaphore process for controlling access to the link's queue space. A semaphore process is required for controlling access to a link queue because at the same time, one node may desire to place data on the queue while another node may desire to remove data from the queue at the same time. The LINK microprocess is also used to avoid "busy waiting" when a node desires a data item and the link queue is empty. In this case, instead of the node repeatedly querying the LINK microprocess whether input link data is available, the LINK microprocess accepts a request for data from the node and then, when the data is available, transfers the data to the node which is in a waiting execution-state. Thus, as will be seen in more detail later, the LINK microprocess allows the trigger function of a node (i.e., deciding when a node is ready to execute) to be monitored in a non-busy way. A similar handshaking mechanism is used to avoid a node "busy waiting" until there is room on the link to store output link data. In addition, the LINK microprocess allows the updating of queue pointers to go on in parallel with a node's further processing.

The NODE microprocess implements the following overhead operations required to sequence a node: 1) fetching the input data items from the appropriate input links, 2) deciding when the node operation is ready to be executed, 3) transferring the input data items to the

appropriate microprocess that will perform the node operation, and 4) transferring the output of the node operation to appropriate output links. The CDS associated with each NODE microprocess is designed so that as many of these overhead operations can be done either in parallel or overlapped between consecutive executions of a node.

The overall CDS for the AGML emulator is pictured in Figure 3c. This section has presented the AGML emulator in terms of a set of microprocesses, each of which performs a small independent task. The next section will discuss how these microprocesses dynamically interact to perform the emulation of a graph program. These interaction patterns will be detailed through a discussion of the PARALLEL-NODE microprocess.

III.2 The Microcoding of the Parallel Node

The PARALLEL-NODE, whose CDS is pictured in Figure 4a, is the most complex of the three types of NODE microprocesses because of the control structures required to generate and keep track of the multiple concurrent initiations of the primitive operations of the node. In order to generate multiple initiations, the fetching of input link data for an operation, which is done by the INPUT-PNODE microprocess, is separated from the storing of output link data for an operation, which is done by the OUTPUT-PNODE microprocess. This separation of the input and output phases of a PARALLEL-NODE permits the fetching of input data for one operation to be performed concurrently with the storing of output data for a previously initiated operation. In order to insure the output-determinancy of the graph procedure, multiple initiations of an operation must terminate in the same order as they were initiated. The PARALLEL-NODE maintains the correct ordering of multiple initiations through a mechanism which holds up the storing of the output of an operation until the output of all previously initiated operations have been stored.

The PARALLEL-NODE interacts directly with the following microprocesses: GRAPH-PROCEDURE, SCHEDULER, INPUT-PNODE, OUTPUT-PNODE, and PROCESSOR(1) ... PROCESSOR(n). The PARALLEL-NODE control environment is a two level hierarchy of global process environments. The top level allows for access to SCHEDULER microprocesses while the lower level allows for access to the LINK microprocesses. The PARALLEL-NODE microprocess and its son microprocesses communicate with each other in two ways: through a port attached to each microprocess and through a global data base which these microprocesses all share. The semantics of interaction patterns of the PARALLEL-NODE with these microprocesses is indicated in Figure 4b.

The GRAPH-PROCEDURE microprocess initiates the PARALLEL-NODE microprocess and then, when the graph procedure termination condition has been met, signals the PARALLEL-NODE to terminate. The PARALLEL-NODE, after it has received the terminate signal, waits until all outstanding node operations are completed and then signals back to the GRAPH-PROCEDURE its termination.

The PARALLEL-NODE, once initiated, activates the INPUT-PNODE microprocess to fetch the input data from the

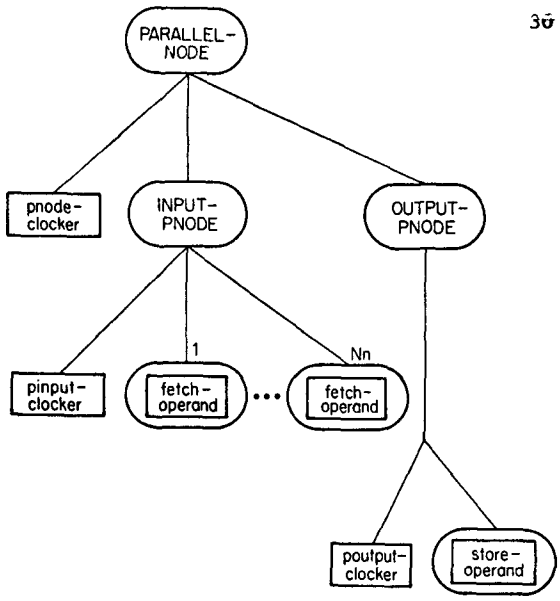


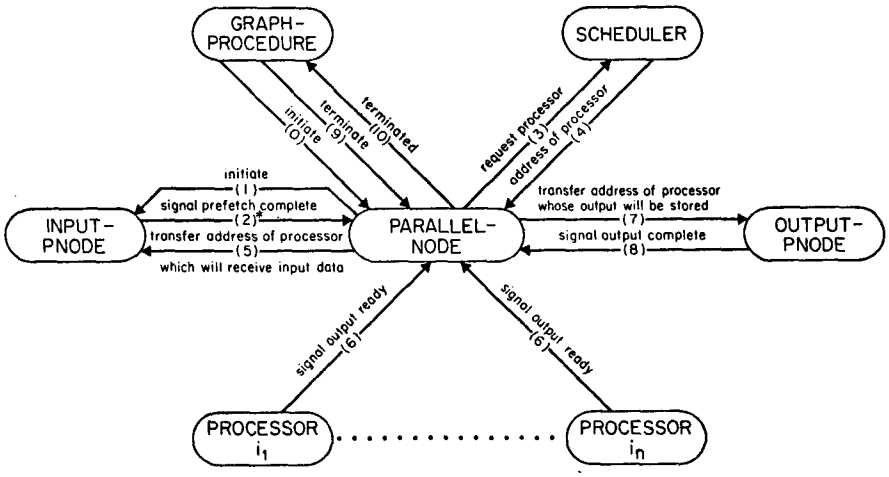
Figure 4a.. Control Data Structure for PARALLEL-NODE

appropriate input links. After receiving the prefetch complete signal from the INPUT-PNODE, the PARALLEL-NODE then activates the appropriate scheduler microprocess to assign a PROCESSOR to perform the operation.* In this way, a PROCESSOR is not assigned to perform a node operation until the data necessary for the operation has been fetched.

The PARALLEL-NODE, after receiving the address of the assigned PROCESSOR from the SCHEDULER microprocess, queues the address and activates the INPUT-PNODE with this address. The INPUT-PNODE then transfers the prefetched input data to the assigned PROCESSOR. After the input data has been transferred, the INPUT-PNODE attempts to prefetch the input data for the next operation.

The PROCESSOR(i) microprocess, after completing the desired operation, signals back to PARALLEL-NODE that the output data is ready. The PARALLEL-NODE then checks whether PROCESSOR(i) is at the top of the initiation queue. If PROCESSOR(i) is at the top of the queue, then the address of PROCESSOR(i) is transferred to the OUTPUT-PNODE microprocess. Otherwise, an indicator is set in the initiation queue that PROCESSOR(i) is ready to store its output data. Thus, through the initiation queue mechanism, the outputs of the PARALLEL-NODE are FIFO ordered so as to make the PARALLEL-NODE determinate.

* The PROCEDURE-NODE is precisely the same as the PARALLEL-NODE except that the PROCEDURE-NODE, instead of calling the SCHEDULER microprocess, generates a state vector for the GRAPH-PROCEDURE microprocess. The address of this newly defined GRAPH-PROCEDURE microprocess is then treated in the same way as the address of the assigned PROCESSOR microprocess.



*Steps 2 – 8 represent the sequence of interactions required for a single node computation.

Figure 4b. Interaction Patterns of PARALLEL-NODE

The OUTPUT-PNODE microprocess, upon receiving the address of PROCESSOR(i), transfers PROCESSOR(i)'s output data to the appropriate output links. After the completion of this transfer, the PARALLEL-NODE is notified. The PARALLEL-NODE then examines the initiation queue to determine whether the PROCESSOR(j) at the top of the queue has already signaled that its output is ready. If so, then the OUTPUT-PNODE is reactivated with the address of PROCESSOR(j).

These interaction patterns allow the fetching of input link data, storing of output link data, the execution of an arbitrary number of primitive node operations, and the processing of requests to store the output of an operation all to proceed in parallel. The PNODE-CLOCKER microprogram, which is the collection of these microcode routines for handling communications to the PARALLEL-NODE, is less than 70 (64-bit) microinstructions long. The microprogram memory required for the entire AGML emulator is less than 600 microinstruction words. Of this microstorage, approximately 220 microinstructions are used for building up the CDS, 300 microinstructions for dynamic control, and the remainder for holding data constants. An entire listing of the AGML emulator is contained in Appendix C of [LES72].

III.3 Dynamic Execution Behavior

In order to verify that this emulator design actually exploits the implicit parallelism* of an AGML program and performs the overhead operations required to sequence a graph procedure in a parallel manner , the emulator was run on a simulator of the microcomputer architecture so as to measure its dynamic execution behavior. The performance statistics to be presented in the remainder of this chapter are based on the emulation of two graph programs. The first graph program, Sum-Squared (Figure 5) calculates the sum of the squares of the elements of a vector of numbers. The vector is initially placed on the external input link with its last element being zero. The node "2 Copies" copies the data on its input data link to its two output links. The node "Branch and Route" routes the data on its first input link (connected to the "+" node) to the external output link if its second input link (connected to the "=0" node) contains a true value; otherwise, the output data from the "+" node is routed back to the "+" node for continued summing. The computational structure of this graph program can be thought of as a three level pipeline that flows into an iterative summation network.

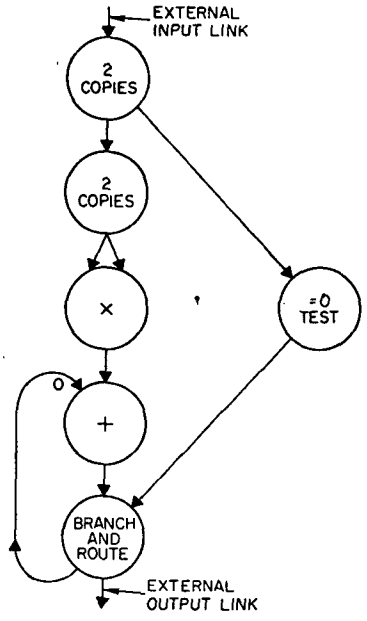


Figure 5. Sum-Squared Graph Program

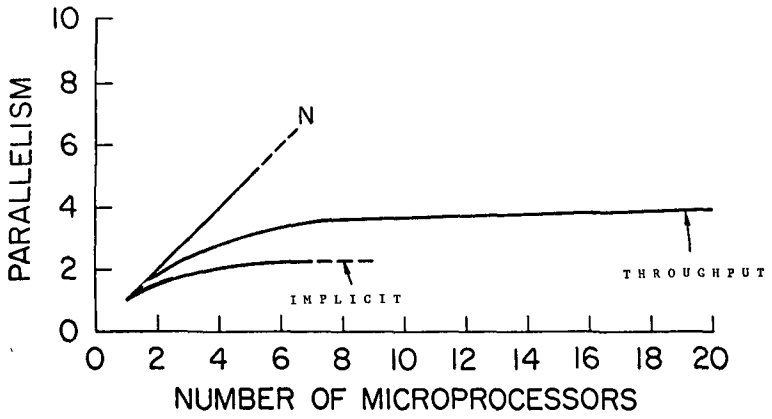


Figure 6. Performance Characteristics of AGML Emulator on Sum-Squared Graph Program(S)

The second graph program, Sum-Eighth-Power, calculates the sum of the eighth power of the elements of a vector of numbers. The computational structure of this second graph program is similar to that of the first graph program except that the pipeline part of the computation has seven levels.

The dynamic performance characteristics of the AGML emulator were evaluated in two ways. The first evaluation technique compared the implicit parallelism* of the Sum-Squared graph program with that of the throughput parallelism of the emulator when emulating this graph program (Figure 6), as a function of the number of processors. The implicit parallelism of the AGML program, as a function of the number of processors, is measured through the use of a simulation technique developed by Nelson [NEL70], whereas the throughput parallelism is calculated by dividing the total time to execute the emulator in a hardware configuration containing n microprocessors by that for a single microprocessor. The comparison of these two curves indicates the AGML emulator takes advantage of the implicit parallelism of the graph program (i.e., throughput curve dominates implicit parallelism curve) and the overhead operations are made parallel (i.e., the postive difference between the two curves). These conclusions indicated by this comparison can also be observed by examining the Microprocessor Utilization Curves of the Sum-Eighth-Power graph program in Figures 7a-c. The dynamic activity of the AGML emulator can be partitioned in terms of six sections, as labeled in Figure 7a. The activity of the first section, which is mostly sequential, represents the dynamic construction of the CDS for the particular graph program being emulated. The activity of the second section represents the initiation of all nodes in the graph, and their subsequent activity involved with determining whether they can execute. The activity of the third section mirrors the gradual initiation of the pipeline part of the graph computation. The activity of the fourth section mirrors the execution of a fully loaded pipeline. The activity of the fifth section mirrors the unloading of the pipeline part of the computation followed by the iterative summation part of the computation. Finally, the activity of the sixth section represents the termination of all the nodes of the graph after the final output appears on the external output link. This sequence of microprocessor utilization curves indicates that an AGML emulator can use available microprocessors, where sufficient parallelism exists, to reduce in a proportional way the time it takes to complete each of the sections of the curve.

* The implicit parallelism of an AGML program is based solely upon the sequencing rules of the AGML, i.e., the rules which define when a node may execute. This measure of parallelism does not take into consideration any of the bookkeeping operations, both software and hardware, required to implement the parallel activity of the graph program, e.g., the fetching and storing of data on links or the monitoring for when a node can fire.

IV. A Post-Mortem of the Emulator Design

The major problem in designing an emulator as a set of closely-coupled microprocesses is how to structure the decomposition so that parallelism at the virtual level (microprocess activity) is directly translatable into an actual speedup in the emulation of the IML. This mapping problem in turn centers on three design issues, each of which relates to how microprocesses interact:

1. the design of the interlock structure for a shared data base,
2. the choice of the smallest computational grain at which the system exhibits parallel activity, and
3. the techniques for scheduling a large number of closely-coupled microprocesses.

The first design issue is important because in a closely-coupled process structere many processes may attempt to access a shared data base at the same time. In a uniprocessor system, the sequentialization of access to this shared data base does not significantly affect performance because there is only one process running at a time. Whereas in a multiprocessor system, if the interlock

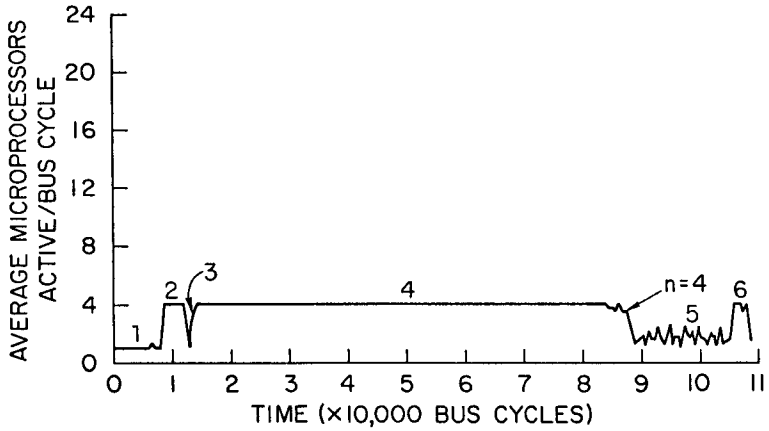


Figure 7a. Microprocessor(4) Utilization Curve for Sum-Eighth-Power Graph Program(10)

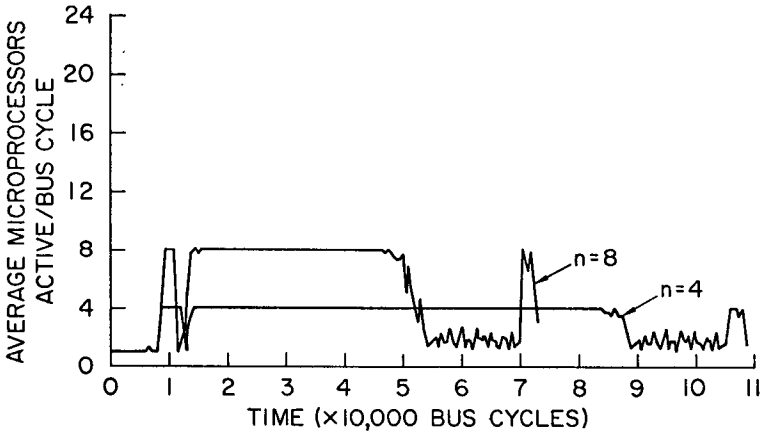


Figure 7b. Microprocessor (4 vs 8) Utilization Curve Sum-Eighth-Power Graph Program(10)

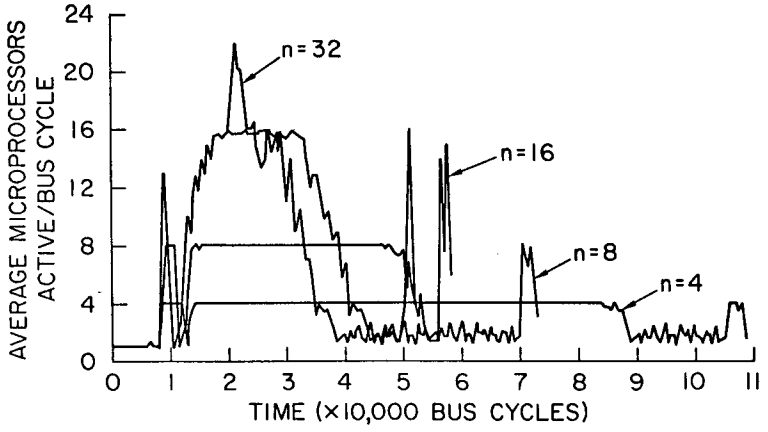


Figure 7c. Microprocessor (4,8,16,32) Utilization Curve Sum-Eighth-Power Graph Program(10)

structure for a shared data base is not properly designed so as to permit as many non-interfering accesses as possible, then access to the shared data base becomes a significant bottleneck in the system's performance [MCC72]. This design issue was manifested in the initial design for the AGML emulator. This initial design contained only one SCHEDULER microprocess, which was used to schedule all node operations (i.e., the scheduler queue was the shared data base which accesses to were sequentialzed). In simulation runs, the SCHEDULER microprocess was a significant bottleneck in the emulator's performance when there were bursts of parallelisms of in the AGML program being emulated. This bottleneck was substantially reduced by a redesign of the emulator which contained multiple SCHEDULER microprocesses; each class of node operations being scheduled by its own SCHEDULER microprocess.

The second issue relates to how closely-coupled processes can interact. If the grain of decomposition is such that the overhead involved in process communication is significant in relation to the amount of computation done by the process, then the added virtual parallelism achieved by a finer decomposition can decrease, rather than increase, the performance of the system. This design issue was manifested in the decomposition of the NODE microprocesses which should not have been decomposed such that link data was fetched in parallel. This added level of parallelism did not justify introducing another level of process structure and communication (i.e., between the INPUT microprocess and the FETCH-OPERAND microprocesses). This added level of process structure introduced a great deal more hardware system overhead (discussed in the next section) and microprocess communication without any actual speed up in the emulation process.

The third issue relates to a phenomenon called the "control working set" which was discovered through simulation runs of AGML emulator [LES72]. This phenomenon predicts that the execution of a closely-coupled process structure on a multiprocessor may result in a significant amount of supervisory overhead caused by a large number of process context switches. The reason for this high number of process context switches is analogous to the reason for "thrashing" within a data working set [DEN68]. For example, in a uniprocessor system if two parallel processes closely interact with each other, then each time one process is waiting for a communication from the other it would have to be context switched so as to allow the other process to execute. If these two processes communicate often then there would be a large number of context switches. However, if there were two processors, each containing one of the processes, then there would be no process switching. The implications of this phenomenon are that the process scheduling strategy should schedule a group of processes rather than a single process at a time, and the grain of decomposition of the system should relate to the number of available processors.

V. Summary

The design of the AGML emulator has demonstrated on both the representational and execution levels that a complex emulator can be efficiently structured as a large number of microprocesses, each performing a small independent task, that interact in a closely-coupled manner. In particular, on the representational level, it has been shown that an emulator can be compactly and simply coded, i.e. the entire emulator requires less that 600 64-bit word of microprogram memory; and that a wide variety of different types of control structures (e.g., distributed control, semaphore processes, message queuing, broadcast control, etc.) can be naturally integrated together in a single framework. On the execution level, it has been shown that an emulator can be structured so as to fully map parallelism expressed at the virtual machine language level into parallelism at the hardware level. In particular, it has been shown that the emulator can be structured so as to utilize a large pool of identical microprocessors (i.e., greater than sixteen). In addition, the design of the emulator has incoporated the idea of tailoring the emulator's control structure not only to the emulated machine language but also dynamically to the specific program currently being emulated. In summary, the design of the AGML emulator has indicated that this new structuring paradigm for an emulation is a feasible technique for the design of a complex emulator operating in a parallel hardware environment.

BIBLIOGRAPHY

ADA68 D.A. Adams, "A Computational Model with Data Flow Sequencing", (Ph.D Thesis), Report No. CS117, Computer Science Department, Stanford University, December 1968.

AND67 D.W. Anderson et al, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling", IBM Systems Journal, Vol. 11, No. 3, January 1967, pp. 8-23.

BEL70 C.G. Bell and A. Newell, "The PMS and ISP Descriptive System for Computer Structure", in 1970 Spring Joint Comput. Conf.,. AFIPS Conf. Proc., Vol. 22. Montvale, N.J.: AFIPS Press, 1970, pp. 657-676.

DEN68 P.J. Denning, "The Working Set Model for Program Behavior", Commun. Ass. Comput. Mach., Vol. 11, May, 1968, pp. 323-333.

ERS71 A.P. Ershov, "Parallel Programming", Artificial Intelligence Laboratory, Report No. AIM-146, Computer Science Department, Stanford University, July 1971.

KAR66 R.M. Karp and R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", SIAM J. Appl. Meth. 14, November 1966.

LAM68 B.W. Lampson, "A Scheduling Philosophy of Multiprocessing Systems", Commun. Ass. Comput. Mach., Vol. 11, May 1968, pp. 347-359.

LES72 V.R. Lesser, "Dynamic Control Structures and their use in Emulation"(Ph.d Thesis), Report SLAC-305, Stanford Linear Accelerator Center, Stanford University, September 1972.

LES71 V.R. Lesser, "An Introduction to the Direct Emulation of Control Structures by a Parallel Microcomputer", IEEE Tran. Comput., Vol. C-20, July 1971, pp. 751-763.

MCC72 J.W. McCredie, "Analytic Models of Time-Shared Computing Systems: New Results, Validations and Uses", (Ph.D. Thesis), Comp. Sci. Dept., Carnegie-Mellon Univ., Chapter 5, 1972.

MCK67 W. McKeeman, "Language Directed Computer Design", in 1967 Fall Joint Comput. Conf., AFIPS Conf. Proc., Vol. 31. Washington, D.C.: Thompson, 1967, pp. 413-418.

NEL70 E. Nelson, "Graph Program Simulation", Report No. CS185, Department of Computer Science, Stanford University, October 1970.

NEL72 E. Nelson, "Free Running and Resource Limited Graph Programs", (Ph.D. Thesis), Computer Science Department, Stanford University, September 1972.

RIC71 R. Rice and W.R. Smith, "SYMBOL: A Major Departure from Classic Software Dominated von Neumann Computing Systems", in 1971 Spring Joint Comput. Conf., AFIPS Conf. Proc., Vol. 38, Montvale, N.J.: AFIPS Press, 1971, pp. 601-616.

ROD67 J.E. Rodriquez, "A Graph Model for Parallel Computations", (Ph.D. Thesis), Mass. Institute of Technology, September 1967.

ROS69 R.F. Rosin, "Contemporary Concepts of Micro-programming and Emulation", Comput. Surveys, Vol. 1, December 1969, pp. 197-212.

THO64 J.E. Thornton, "Parallel Operation in the Control Data 6600", in 1964 Fall Joint Comput. Conf., AFIPS Conf. Proc., Pt. II, Vol. 26. Washington, D.C.: Spartan Books, 1964, pp. 33-40.

TUC65 S.G. Tucker, "Emulation of Large Systems", Commun. Ass. Comput. Mach., Vol. 8, December 1965, pp. 753-761.

WIL72 W.T. Wilner, "Design of the B1700", Burroughs Corporations, Santa Barbara Plant, Goleta, Calif., May 1972.

[Following the talk, there was a lengthy question and answer session related primarily to the meaning of Lesser's Figure 5, the "Sum-Squared Graph Problem". Several in the audience, assumed that the example illustrates a purely parallel organization, and worried that an inopportune timing might cause exit through the bottom-most node before the computation was complete. Dr. Lesser pointed out that although the execution is essentially parallel, the values going into the input link of a node are serially processed and thus the values into the "=0" node test are serial. In point of fact, there is a FIFO queue for each node for preserving the order of the inputs. This discussion was followed by more general questions. Ed.]

Q: You mentioned that there were internal communication problems. Do you have any feeling for the amount of hardware you are going to need to build the communication?

Lesser: I did most of a hardware design and it didn't look technologically impossible to build such a machine but I cannot conjecture what part of the hardware would go into communication.

Q: Can you make more of a distinction between a functional unit and a microprocessor?

Lesser: In my system, the semantics are intertwined -- wherever you use a functional unit, you can use a microprocess to implement the functions performed by a functional unit. The calling sequence for invocation of a functional unit is a subset of the calling sequence for a microprocessor.

[The first section of the Rossmann/Jones talk, the general description of functional memories was given by Professor Rossmann. The second section, presented by Professor Jones, concerned the uses functional memories are put to and why the topic is appropriate to the Microprogramming-Programming Language Interface. Professor Rossmann then returned to describe the application to SNOBOL. Ed.]