A loosely-coupled applicative multi-processing system*

by ROBERT M. KELLER, GARY LINDSTROM and SUHAS PATIL
University of Utah
Salt Lake City, Utah

INTRODUCTION

The architecture of highly-parallel machines has received increased attention from researchers over the past decade. At first, because of the machines' novelty, workers were content with proposing elaborate machine architectures without giving due consideration to how such machines would ultimately be programmed to exploit their available computational power. Experience with Illiac IV, Star-100, etc. has shown this to be a mistake. Indications are that programming languages deserve consideration at the earliest stages of architectural conception. Included in such considerations are issues such as storage and task management.

This paper describes a proposed machine, AMPS (Applicative Multi-Processing System). It features a *loosely-coupled architecture*, incorporating a large number (say 1000) of processors functioning independently to a large extent, but effectively interacting when necessary. Furthermore, the programs supported are not tied to the structure of the machine, thereby facilitating expandability. Such expandability is further enhanced by the particular physical organization to be described. The architecture of AMPS attempts to bring costs of communication among processing units to a manageable level by taking advantage of *locality of reference*.

Our architecture is currently in the development stage. We present in this paper some of our major philosophical considerations, along with an execution model for a subset of the machine language.

LANGUAGE ISSUES

Heretofore, research on highly-parallel machines has predominantly emphasized statically-structured, usually numerical, computation. We are orienting our design toward dynamically-structured, often symbolic, computations. Although we do not exclude numerical applications from AMPS, we are designing it to provide direct support for languages such as Lisp which have been invented for symbolic computation. In fact, the machine language for AMPS itself resembles a dialect of Lisp. By retaining a close relationship between a higher-level language and its supporting machine language, debugging is facilitated. Furthermore, the *applicative* (i.e. based on *function application*) nature of our machine language obviates many pre-processing transformations of the type used to extract parallelism. Such transformations are really just means of extracting functional dependencies which are easily determined from an applicative program.

To further clarify, consider the task of counting the leaves of a binary tree. Figure 1a presents an applicative program for this task, whereas Figure 1b presents a corresponding non-applicative program employing a stack. Clearly, Figure 1a is easier to understand and its inherent parallelism is easier to detect than that of Figure 1b.

Lisp, with some minor extensions, such as *lenient cons* discussed in the seventh section (*cf.* References 9 and 12) seems to include all opportunities for exploitation of concurrency that proposed data flow languages do, and more. It provides concurrent operations on tree or graph data structures during their creation, and natural ways for dealing with conceptually infinite structures. By supporting such a language at the machine level, we also provide a natural means of communication between processes and their environment, e.g. file systems and I/O devices. Space limitations preclude discussing these issues here, but related ideas may be found in References 10 and 22. It is also worth mentioning that our machine language can directly support other languages which deal with infinite structures such as those in References 5, 2 and 13.

BASIC ARCHITECTURE

The physical arrangement of components in AMPS is a tree structure with two types of nodes. The scaled-down version in Figure 2 is merely meant to be suggestive, as we envision trees with 1000 or more nodes as being feasible in the next decade. Combined processor/memory units are attached as leaf nodes. The internal nodes of the tree structure are combined communication and load-balancing units. Other more specialized units might also be present, attached

* This work was supported in part by grants DCR-74-21822, MCS-77-09269 and MCS-78-03832 from the National Science Foundation.

```
a.  Applicative Program

    leafcount(t) = if atom(t)
                       then 1
                       else leafcount(left(t)) + leafcount(right(t))

b.  Non-Applicative Program

    leafcount ← 0;
    stack ← {t};
    while non-empty(stack) do begin
        pop T from stack;
        if atom(T)
            then leafcount ← leafcount + 1
            else begin
                push left(t) on stack;
                push right(t) on stack
                end
        end
```

Figure 1—Applicative and non-applicative versions of leafcount.

closer to the root node for enhanced accessibility and utilization, but we do not further discuss this possibility here.

This paper makes the assumption of a binary tree for simplicity, although technology considerations suggest that a 4-ary or 8-ary tree might be more appropriate. For expedited communication, we may eventually include additional links laterally connecting the tree, possibly as suggested in Reference 8.

A processing unit in AMPS is roughly the size of a conventional micro-computer, but its architecture is substantially different. It is able to carry out local computation, particularly with respect to assembly and dissemination of information, and to initiate actions for fetching information from other nodes of the tree. It is able to execute program tasks sequentially or in an overlapped mode, and to allocate storage in response to the execution of *invoke* instructions. *Invoke* instructions create tasks which are then executed either in the local processing unit or in another processing unit, as system loading dictates (see the eighth and ninth sections).

The primary memory of the system is distributed among the processing units. Each processing unit has direct access to that segment (e.g. 64K words) of memory located within it. It also has access, through the *communication network*, to the segments of memory located at other processing units. Even though the memory is distributed among the processing units, there is only one *unified address space*. Given the address of a datum, any node in AMPS is able to logically access it without address translation. Such addresses do not appear at the assembly language level, as they are generated by the system dynamically. The internal nodes of the communication network are responsible for any required routing of data. Access to auxiliary memory and other forms of external communication takes place through special-purpose leaf processors, which will not be discussed here.

COMMUNICATION NETWORK

The communication network in AMPS is designed to take advantage of locality of information flow, thereby reducing communication costs. Information first travels up the tree towards the root node until it encounters a node which spans the destination leaf, whence it proceeds down the tree until it finally reaches the desired destination. Thus, for sending or receiving information from neighboring leaves, it is not necessary for the information to travel the entire depth of the tree. Relatively local data flow therefore takes less time and the overall communication cost of the computation is thereby improved.

A second function of the communication network is to provide a reasonably balanced distribution of the computing load. Such a function is useful since the machine allocates tasks dynamically. Each node of our communication network periodically obtains *load monitoring signals* from its subordinates, which indicate their current degrees of utilization. When such signals indicate a sufficiently unbalanced state, the node can cause the transfer of uninitiated tasks from one subtree to the other (see the section on Load Balancing).

LOCALITY

One of the most important concepts of our architecture is an attempt to improve performance by exploiting locality of information flow. Locality is an established phenomenon in program execution, which should therefore be exploitable within applicative programs. Locality will be enhanced by the fact that functions are naturally apt to confine their references to certain portions of data structures. Secondly, repeated global references to the same data will become localized by a *caching effect* incorporated in the referencing scheme of AMPS. Note that the read-only nature of data in applicative systems greatly simplifies the implementation of such caches.

If computations which interact heavily with one another are allocated space in leaves that are a short average distance apart in the communication network, the overall time spent in information flow will be reduced. It is important to note that even if it is not possible to allocate space for a new computation in the storage space at the invoking leaf, the correctness of the overall computation will be maintained, even though the speed of the computation may be decreased. This is aided by the uniformly acceptable address space and the deferred binding of program blocks to physical addresses. Moreover, such locality should tend to cause traffic flow to decrease with node level, thereby balancing bandwidth with demand.

In designing a highly-parallel machine, care must be taken that costs associated with creating new tasks and communicating with them do not outweigh the speed advantage

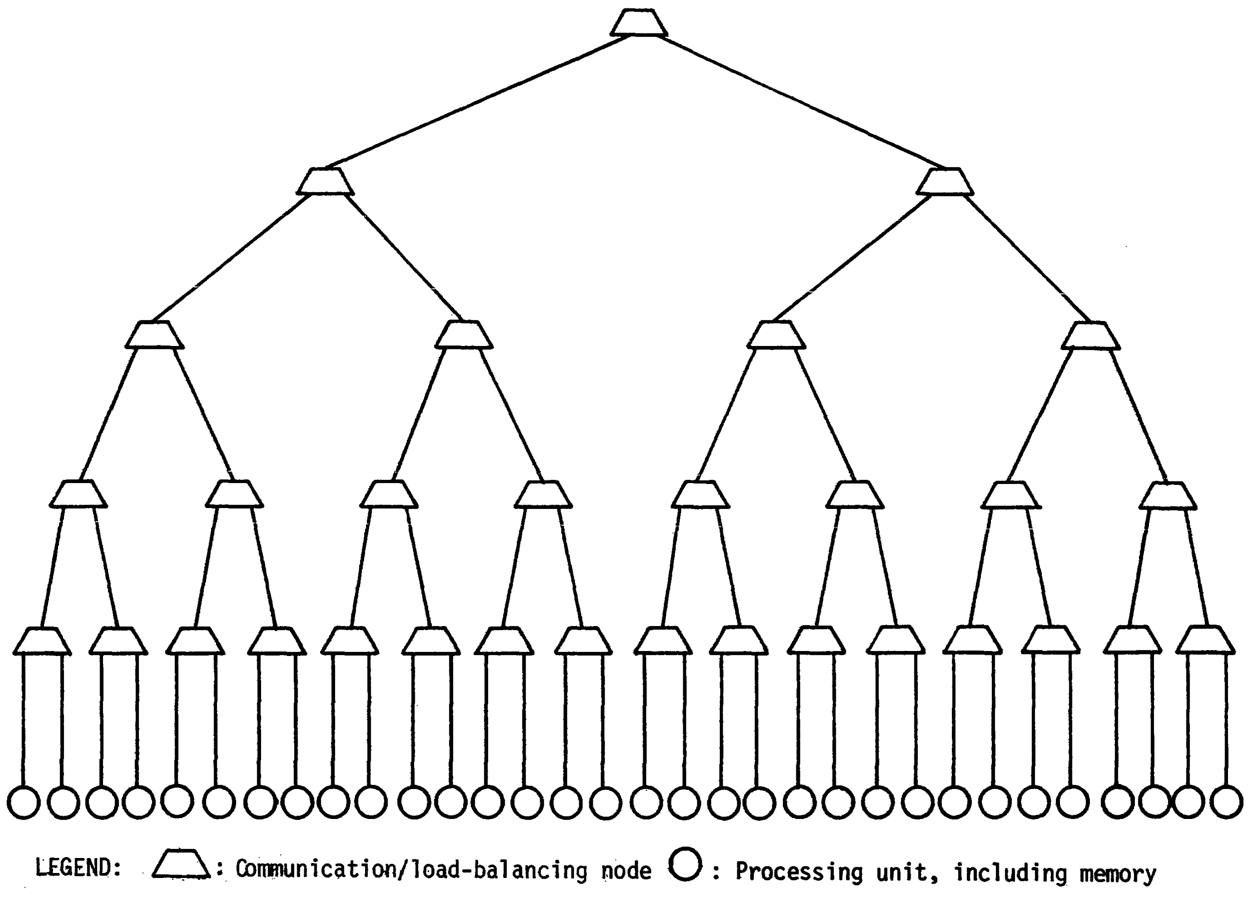

LEGEND:
: Communication/load-balancing node
: Processing unit, including memory

Figure 2—Physical form of AMPS.

gained from overlapped execution of these tasks. Consequently, our design prescribes that computation is divided into blocks in such a way that all computation local to a block (i.e. exclusive of communication with other blocks) will be done within one processing unit. Since such blocks are identical with the code blocks of the program (see the eighth section), locality is further enhanced by the clustering of connected operators within blocks. The global structure does not seek gains from parallelism on the level of, say, a simple arithmetic expression. Instead, this effect is achieved within the processing unit itself.

Another anticipated effect which will contribute to locality might be called the *seeding effect*. When a task A in execution creates a second task B, the latter may be allocated its storage in any of the processing units in which there is sufficient space. Since B may cause the creation of other tasks C1, C2, . . ., Cn, locality is enhanced if the storage for the latter is allocated in processing units near to that of B in the tree. Hence even if B is a long distance from A, thus incurring non-trivial communication cost between the two, this cost may be balanced out by the lower costs of communicating between B and C1, C2, . . ., Cn.

The seeding effect creates a tradeoff in resolving a choice of how far away a created task should be placed. It also demonstrates the possibility of a certain amount of *natural re-localization* in recovering from bad task-placement decisions by the system. For example, even if B were placed in a congested area, the storage from completing tasks near B could eventually be reclaimed to provide more space for C1, C2, . . ., Cn. Although such a scheme will not always work with optimal efficiency, it will work until all space is in use.

INFORMATION FLOW

The characterization of information flow within the machine is dependent on the conceptual level being considered. For example, at the *task level*, we are concerned with the flow of operands among tasks, which we implement in a *demand driven* fashion. In the demand-driven scheme a task may actively seek additional pieces of data after its initiation. In contrast, most proposed data flow machines are primarily data-driven, in that an instruction never asks for data to be sent to it. Instead, it waits to receive data itself, and when all necessary operands have been received, it initiates the computation, the result of which is then sent to all other designated instructions.

At the *communication network level*, we find the information flow separated into the flow of *tasks* (which at this level are always *invoke* instructions), *operands* (single data words), and *blocks* (multiple data words), and requests for the latter two. All such pieces of information are accompanied by a destination address. All information transmitted through the communication network is handled by *packet switching* (i.e. *store-and-forward*). Line-switching is not used because of the potential congestion caused by tying up long paths through the network.

A node of the communication network communicates to its parent through a form of handshaking. For block transfers, a *burst mode* of communication is used in which the handshaking occurs only before and after the entire block has been transferred, thus drastically reducing the associated overhead.

MACHINE LANGUAGE

As its language, AMPS executes a compiled dialect of Lisp called *FGL*, for *Flow-Graph Lisp*, or *Function Graph Language*. Although FGL programs are stored as blocks of compiled code, we prefer to present them using *function graphs*, as described in Reference 15. FGL allows us to display clearly the data flow between operators and thus the potential concurrency within programs.

A program in FGL consists of a main function graph, together with *productions* for programmer-defined functions. These productions specify how a node containing a function name (the *antecedent* of the production) is to be replaced by a function graph (the *consequent* of the production). FGL provides a repertoire of basic operators (e.g. the primitive functions of Lisp) that may be used in constructing function graphs.

For the sake of this presentation, let us suppose that data structures are *trees*, with integers and *nil* as atoms. Conditional evaluation is obtained through the use of the function *cond*, which causes the evaluation of its second or third argument, depending on whether its first argument is non-*nil* or *nil*, respectively.

Trees are built using the *cons* operator, which forms trees from atoms or other trees by connecting its arguments as subtrees of a common root. The selector functions *car* and *cdr* extract the left and right subtrees respectively of a tree built by *cons*. The *cons* of FGL is in fact *lenient cons*, which allows the machine to exploit concurrency which it could not with conventional *strict cons*.[9] More precisely, FGL semantics provide that the identities *car*(*cons*(x,y))=x and *cdr*(*cons*(x,y))=y hold independent of whether x and y are convergent.

For simplicity, we do not discuss input of trees. Rather, we assume them to be resident at the beginning of the computation, built by an appropriate graph of *cons* operators and atoms. Conceptually, trees flow on the arcs of an FGL graph. In actual implementation, however, the flow of a non-atomic tree is represented by the flow of a pair of pointers to its subtrees. For the current presentation, iteration is implemented by recursion, in the manner of Reference 19. This can be shown to give automatically the same concurrency-detection effect of "look-ahead" processors, which "unfold" iterations to achieve concurrency.[14]

To cause a result to be printed, a *demand* is generated at some *print* node in the function graph. This causes propagation of the demand to the operator feeding the *print*. When that operator ultimately produces a value, it will then be printed.

Evaluation consists of a combination of transmutations to the graph and the application of basic operators. In this sense, AMPS is a *reduction machine*,[4] executing a *reduction language*.[3] By exploiting the richer connectivity of graphs, we can avoid much of the *combinatorial explosion* which takes place in purely string-oriented reduction machines.

Figures 3 through 7 give examples of programs in FGL. Figure 3 presents a production for the function of Figure 1a, which counts the leaves of a tree. This example uses the *strict* operator (i.e. one which demands *both* of its arguments) *add* to cause the creation of instances of operators which can be evaluated concurrently. Figure 4 shows a possible snapshot of the program during its application to a specific tree. Several concurrent sub-computations are visible.

In Figure 5a, we present a *main program* which calls a recursively-defined function NATNUMS, the graph of which is presented in Figure 5b. Intuitively, NATNUMS(n) "computes" the infinite sequence {n, n+1, . . . } by constructing its representation in the form of a tree as shown. In the context of the main program, the value printed is the second element of the sequence where n=2, i.e. *car*(*cdr*(NATNUMS(2))). Adjoined to the graphs in Figure 5 are listings of their FGL assembly code, the meaning of which is explained in the next section. The parenthetic labels on the graph indicate the correspondence between the graph and the code.

Figure 6 presents a program which employs the function NATNUMS to generate the prime numbers in increasing order. The reader may wish to refer to the similar examples in References 2 and 13.

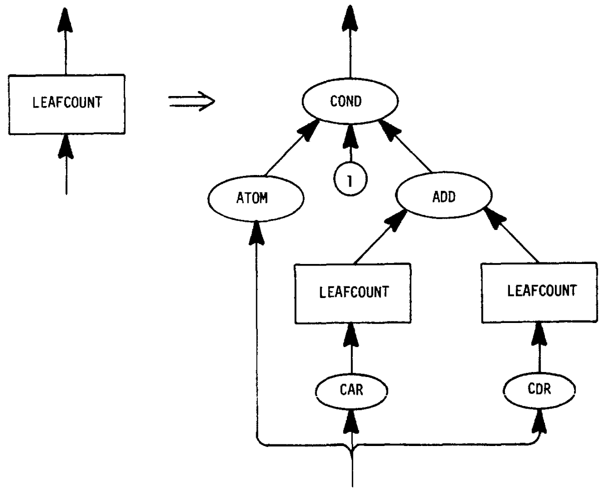


Figure 3—FGL production for the leafcount function.

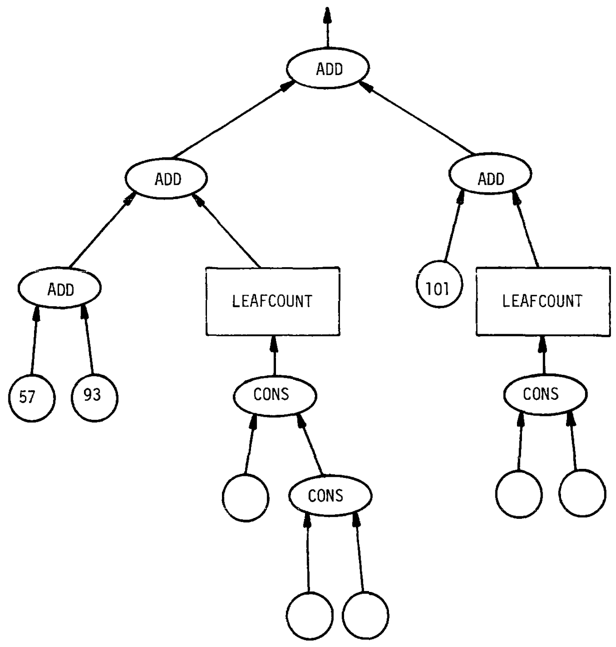
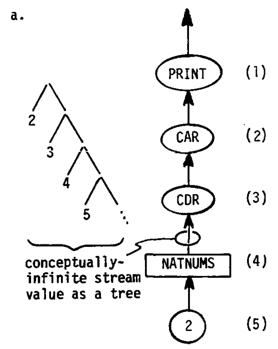


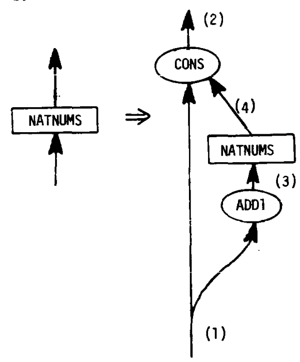
Figure 4—One possible snapshot of the program of Figure 3 during its computation on a tree. Unlabeled leaves are those of the original tree. The ultimate result will be the number 256.

a.



DEF	MAIN	RESULT: 1
1	PRINT	2
2	CAR	3
3	CDR	4
4	INVOKE	NATNUMS 5
5	NUMB	2

b.



DEF	NATNUMS	RESULT: 2
1	PARAMETER	
2	CONS	1 4
3	ADD1	1
4	INVOKE	NATNUMS 3

Figure 5—Sample FGL main-program (a) and production for defined-function NATNUMS (b) with assembly-language listings.

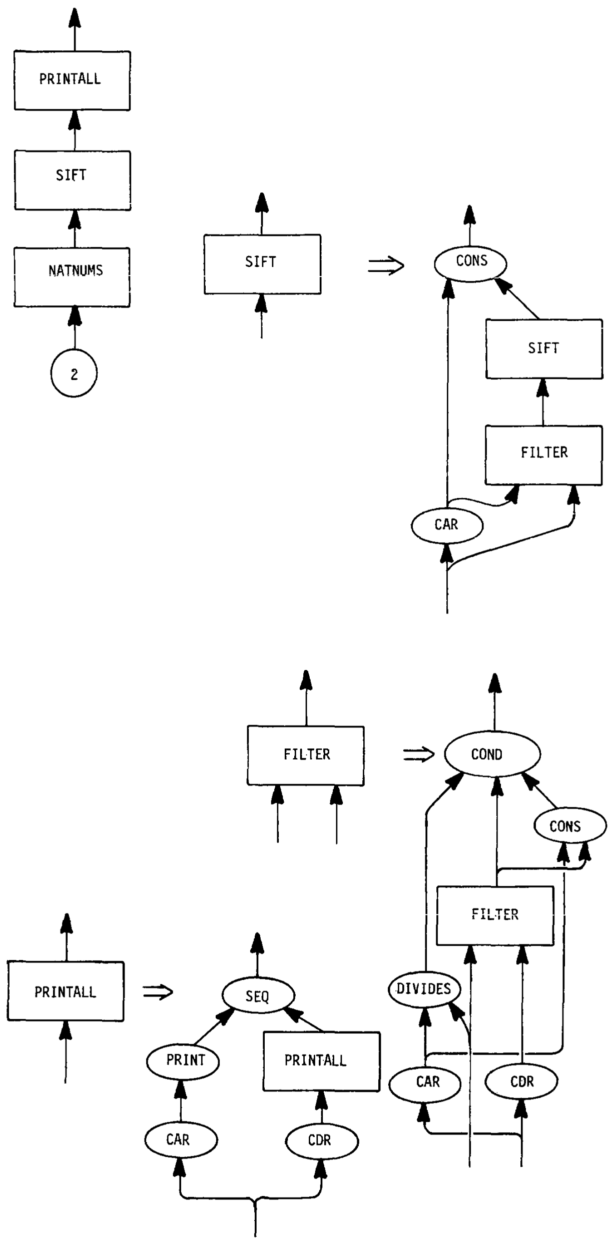


Figure 6—FGL program for printing prime numbers in increasing order. The production for NATNUMS is given in Figure 5. The primitive function *divides* yields *nil* unless its second argument evenly divides its first. The primitive *seq* causes its arguments to be evaluated in sequence.

It can be proved that every well formed interconnection of FGL operators computes a unique function (*cf.* Reference 16), even an interconnection involving cycles. Cycles provide one means of efficiently implementing bi-directional communication between two functions. Such an example is shown in Figure 7, which illustrates a program for the breadth-first production of all atoms in a tree. Detailed description of the recursively-defined functions PASSATOM,

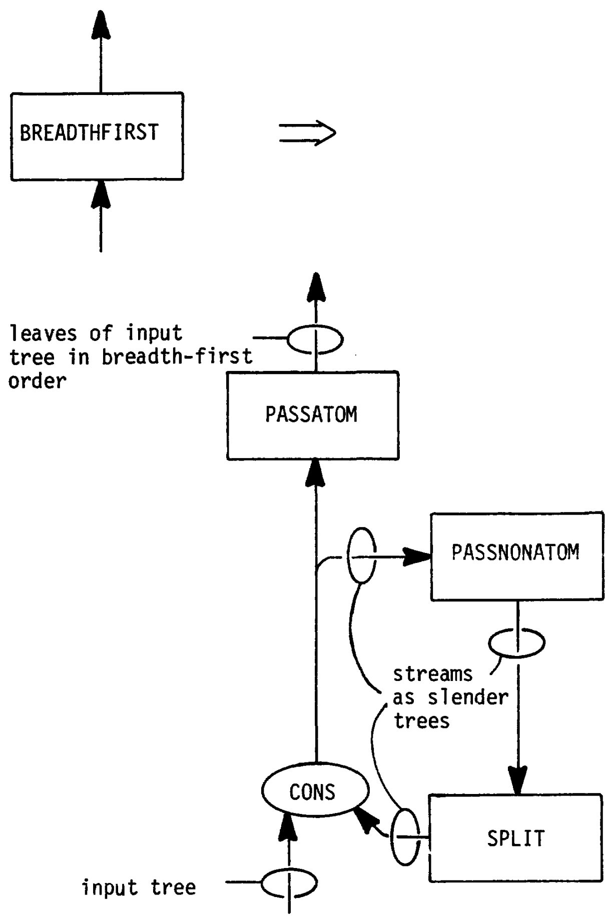


Figure 7—Production defining a function which produces the leaves of a tree in breadth-first order. Productions defining PASSATOM, PASSNONATOM, and SPLIT are not shown.

PASSNONATOM and SPLIT are not presented here. It should be noted that FGL employs fan-out of arcs to effectively avoid recomputation of the same value, an effect which must be obtained by recognition of common subexpressions in some applicative languages.

In the next sections we describe, in more detail, program storage, task execution, typical operators, and production application via the special operator *invoke*. We do not discuss storage reclamation here, for lack of space.

PROGRAM STORAGE AND EXECUTION

All storage is allocated in *blocks*. The use of blocks makes storage management more efficient, and is consistent with trying to keep the *locality* of a computation contained within one processing unit. A block is either a *data block* or a *code block*. The contents of a code block form a linear representation of an FGL graph, which is copied as the source of initial code to be stored in a newly allocated data block. This copying may be viewed as the application of an FGL production, i.e. replacing the antecedent node with its consequent graph. Each word in a data block is initially a code word or a literal value. A code word may get changed to a literal value during execution. The *ready bit* distinguishes whether the word is a value or a code word.

A code word in a data block corresponds to a node in an FGL graph. A value corresponds to what eventually appears on the output arc of that node. The code word contains the local addresses of words corresponding to nodes at the opposite end of its input arcs, i.e. the sources of its operands. Local addresses are relative to the start of a block. We assume here for simplicity that each operator has only one output arc, although such arcs may fan out as necessary.

In addition to specifying the input arcs of its operands, an instruction word may include *notifiers*, which are local addresses of operators which have this operator's output arc as one of their input arcs. These are usually set dynamically as demands occur, although it is possible to have them preset in the initial code.

In addition to an *operation code*, the following fields may or may not be present, depending on the nature of the particular operation:

1. Local addresses of *arguments* of the operator.
2. *Notifiers*, i.e. local addresses of *notifiee* code words, which are instructions which have demanded this instruction's value.
3. A single *global address*, used to provide linkage across blocks, and for specifying the code block in the case of an invoke instruction.

The *invoke* instruction, when demanded, causes the allocation of storage for a data block and the copying of a code block into that storage. The demand driven approach thus provides a natural means of deciding whether and when to trigger the invocation of a defined function, which requires the allocation of a storage block. An *invoke* instruction also initializes various linkage instructions which provide an interface between the nodes of the graph containing the antecedent of the production and those of the consequent, since local addresses cannot be used to provide this linkage. Details on these linkage instructions may be found in Reference 18. Linkage instructions are not shown in the code blocks in Figure 5, as they are supplied automatically by the assembler.

Regarding efficiency, the use of local addresses achieves economy in code word storage, and avoids relocation in copying. The copying of code blocks may be contrasted with approaches such as that in Reference 21, which interpret a pure code block without copying. The approach taken here is more effective in keeping references local to a processing unit. It also eliminates separate fetching of code and data words for each task. Due to the use of a burst mode, blocks are copied more efficiently than the same number of words individually.

TASK EVALUATION

Although code words may be conceptualized as representing functions, we must describe how these words are interpreted by the machine to cause values to be produced. The loosely-coupled aspect of task evaluation is achieved in AMPS through a *task list* organization, which allows many processors to partake in the evaluation of *tasks*, i.e. particular instances of operators with their associated data. The task list is decomposed into two separate lists which may be served independently. These are:

demand list—Contains addresses of operators for which evaluation is to be attempted.
result list—Contains addresses of operators, along with their corresponding values after evaluation.

The *invoke* tasks on the demand list are distributed to individual processing units by the communication network, which takes into account the current processor load profile. Only such tasks are considered for distribution, since it is only these which might profitably be executed in another processing unit, due to the communication costs incurred in their transmittal. Hence, each processing unit has its own *invoke list*, a sublist of the demand list containing only *invoke* instructions.

Figure 8 shows the organization of the task evaluation mechanism. The diagram is to be interpreted in an informal sense, and is less akin to conventional flowcharts than indicative of the flow of tasks in the system. Further details may be found in Reference 18.

Initially, the address of the word which will produce the "main result" is put on the *demand list*. The word itself is then fetched. The instruction specifies certain arguments, which are also fetched. If the arguments are all ready, the function is evaluated. If not, then demand is propagated to each unready argument not previously demanded by placing its address on the demand list and setting a notifier in it. A word may contain several notifiers indicating which instructions have demanded it. The presence of at least one notifier is used to signify a previous demand. Figure 9 sketches symbolically the flow of demand in the example of Figure 5, along with the corresponding production applications and evaluations which take place.

Once evaluated, a result value replaces the code word as *ready* data. Via the result list, any instructions which were specified by notifiers as awaiting this result as an argument are then notified by putting them on the demand list to be retried. Observe that all demanded operators remain accessible until they are replaced by ready data, either through being put on the demand list or through being referenced by a notifier of an accessible operator.

Forms of evaluation other than pure demand evaluation can be supported by judicious pre-setting of notifiers and demand bits and advanced placement on the demand list. Special operators are also available to the programmer which have the purpose of generating advanced demand to enhance parallelism, and for postponing demands to avoid premature allocation of data blocks.

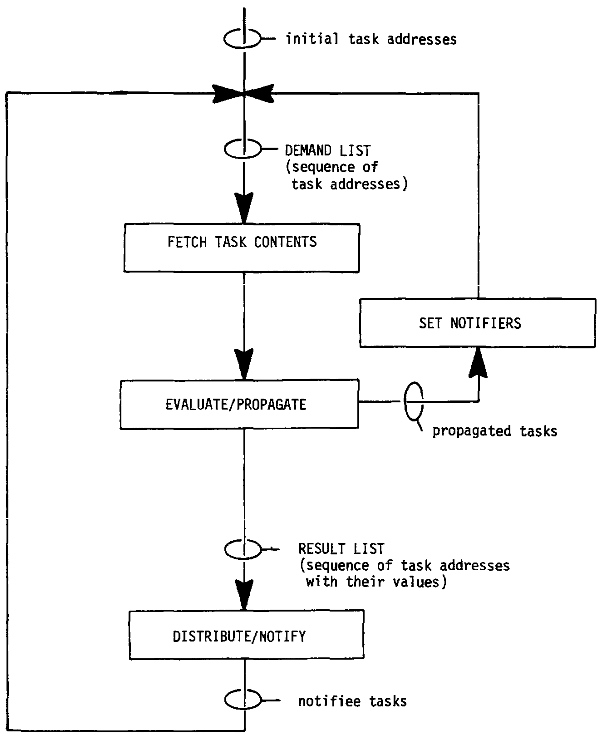


Figure 8—Simplified task processing flow.

PROCESSING UNIT ARCHITECTURE

We do not go into great detail here on the organization of individual processing units. As described in the previous section, each unit selects tasks from its demand list. While on this list, a task is represented by its address in memory. The content of this address is fetched and, if a code word, an attempt is made to evaluate it. This normally entails reference to one or more additional words in the memory.

Since a referenced word might reside in the physical memory of any processing unit, fetching may involve transmission of words through the communication network. In order that the processor need not be idle while such a fetch is taking place, we provide a *staging area* for buffering a set of such tasks while their operands are being assembled. The staging area is conceptually similar to a conventional *pipeline*, except that order of task execution is unimportant, all essential ordering being explicit in the program graph.

The size of the staging area is chosen to maintain reasonably good utilization of the function units within the processing unit, which carry out each operation once its operands are assembled. Of course, each function unit could itself be pipelined, depending on economic advantages which would accrue under particular application loads. Design of such a staging area is fairly routine and therefore will not be discussed further here.

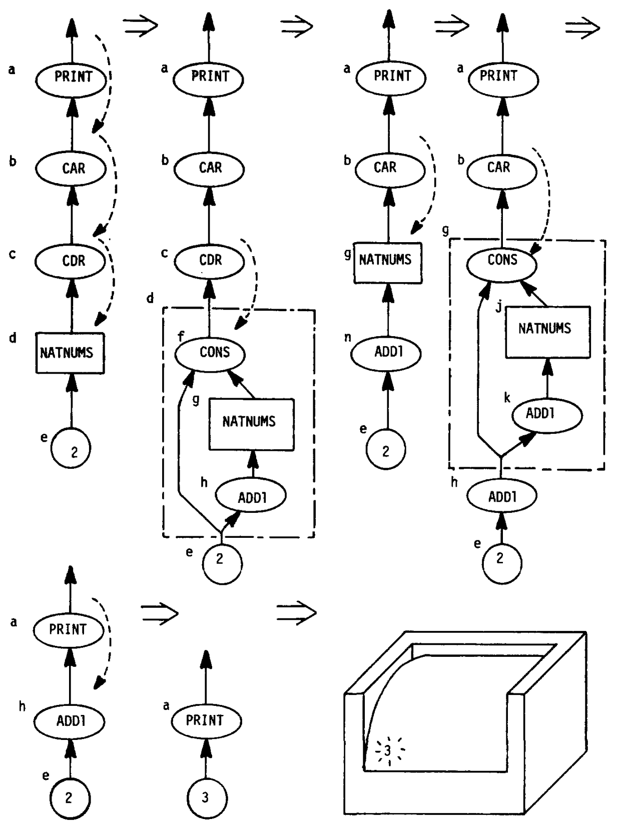


Figure 9—Snapshots of the evaluation of an FGL program. Perforated arrows illustrate demand propagation.

LOAD BALANCING

Load balancing occurs through the redistribution of tasks from the invoke list of one processing unit to that of another. This is a function of the communication network which is separate from, but topologically compatible with, the routing of operand data.

By the *load* at a processing unit, we mean the storage requirement of tasks on the invoke list at that unit. In a similar manner, we can define the *load* at any node of the communication network to be the sum of the loads at its spanned leaves, divided by the number of such leaves as a normalizing factor.

Again, to simplify the explanation, we are assuming that the communication network is a binary tree. Each node of the communication network maintains desired lower and upper limits, L and U, on the loads of its immediate descendants. These are functions of the amount of storage currently in use by the leaves spanned by those descendants. If the load of one descendant is above U and that of the other below L, the network attempts to shift tasks from the invoke list of the overloaded descendant to that of the underloaded one. If loads of both its descendants are above U, this will be communicated to its parent (if any), so that the latter may try to shift some of the load to one of *its* descendants having load less than L. In this way, the balancing function is distributed throughout the communication network, with each node thereof applying the same balancing strategy.

The effectiveness of the balancing scheme relies on the loosely-coupled nature of the system. That is, no task is bound to a particular processor until storge is allocated for it.

COMPARISONS WITH RELATED MACHINES

The data flow machines of References 7 and 1 originally influenced the structure of AMPS. The communication network in AMPS plays the role of the arbitration and distribution network of the Dennis data flow machine. However, the processing units which assemble instructions and initiate information flow are of a higher level, as are the processors in Reference 1.

Even though the architecture of AMPS has a tree like structure, it is not a "recursive architecture" in the sense of Reference 6. The hierarchical structure and method of storage allocation in the Davis machine seem to impose certain constraints on the creation of new computations and on the flow of information in the machine. For example, when a processing element creates a task, the latter must be placed either in the space of the processor carrying out the application or in the space of a subordinate processor, even if all subordinates are crowded for space and the machine has other processors which have plenty of space. This problem does not occur in AMPS, due to the construction of the communication network, the uniformity of the address space, and load balancing.

A common feature of all of the previous architectures is that they are data-driven rather than demand-driven, as is AMPS. One might be led to think that the latter presents some additional overhead. However, closer examination reveals that FGL programs are often simpler in that they do not require explicit instructions for the gating of data.

In data-driven machines, a form of ready-acknowledge signalling is often used for transmission of data via storage words. This is, in fact, a special case of demand-driven computation, in which the demand for an operand is equated with readiness of its recipient. The demand-driven approach seems to provide more flexibility in the relationship between elaboration of a programmer-defined function and the evaluation of its arguments. It is also clear that the demand-driven feature is a necessity in supporting lenient *cons*. On the other hand, demand-driven computation could possibly be *engineered* on other architectures by treating demands as data, but this seems cumbersome.

The tree-structured reduction language machine of Reference 20 is fundamentally different in its operation from AMPS. In the former machine, an expression to be evaluated is mapped symbol by symbol onto the leaves of the tree. In AMPS, an expression would first be converted to function graphs which then reside in the memory space of one or more processing units of the machine.

Our system has in common with those just cited the desire to integrate architectural, communication and language considerations. This is one of the ways it differs from superficially similar systems, such as Cm*.[23] In Cm*, parallel processing is based on the concept of interacting sequential processes that run on conventional processors (DEC PDP-11's), while AMPS is capable of directly evaluating function graphs, integrates considerations of evaluation and local balancing, and directly supports communication among tasks.

CONCLUSIONS AND FUTURE RESEARCH

We have stated that machine architectures should be developed with greater attention paid to ultimate programmability. With this motivation, we have presented a loosely-coupled parallel processing system AMPS which executes an applicative language, FGL. We have sketched the internal representation of programs in AMPS and the execution of programs on it.

Our implementation seems to be the first detailed one presented for Lisp programs on a parallel machine. An implementation has been described qualitatively in Reference 11. However, their description relates mainly to the issue associated with *colonel* versus *sergeant* tasks, sergeants being distinguished from colonels as tasks whose evaluation may never actually be required, but which provide a potentially useful way of employing otherwise idle processors. In contrast, all tasks in the machine described here are of the colonel variety, whose existence may be traced to certain *strict* operators, such as *add* in the leaf count example of Figure 3. On the other hand, subtle details, such as occur in the implementation of a *global notifier* scheme, have been discovered in the course of designing our evaluator.[18] How such subtleties interact with an implementation which does offer sergeant tasks remains a topic for future investigation.

The ideas presented here were derived after considering many possible alternatives. We may, of course, elect to return to one or more of these alternatives after more experience in programming the machine has been gained. A simulator for the evaluation model has been written in Pascal to assist in such a venture. Thus far, the simulator has been used to verify that the evaluation mechanism works and to experiment with additions to the language FGL. Construction of a Simula-67 simulator for the tree architecture is now in progress. We have no immediate plans for construction of a physical realization of the machine.

Issues remaining to be investigated include the necessary support for FGL in terms of storage reclamation and scheduling. We are currently contemplating how best to organize the distributed heap for efficient medium-term data storage.

We have preliminary results on how to deal with other features of Lisp, such as *funargs, setq,* and *prog*. A description of the handling of funargs appears in Reference 18. Efficient access of array-like structures is handled by extending *cons* from pairs to tuples and providing an indexed selector function.

A related issue is whether *indeterminate* computations should be supported, as there are some indications that they permit efficiency gains not otherwise achievable.[17] Under investigation also is the use of machine operators for the support of resource allocation. These have been programmed into the experimental simulator and seem to fit very naturally with our method of evaluation. The usefulness of applicative programs in allowing graceful backup when a processing unit fails also remains to be exploited. Thus many issues, at levels from detailed processor construction to more fundamental language problems, await us.

ACKNOWLEDGMENTS

Comments by Al Davis, Milos Ercegovac, Mark Franklin, Kim Gostelow, Jed Marti, Elliott Organick and the anonymous referees are appreciated. A discussion with Gilles Kahn provided incentive to use demand-driven computation.

REFERENCES

1. Arvind and K. P. Gostelow, "A computer capable of exchanging processors for time," *Proc. IFIP '77*, June 1977, pp. 849-853.
2. Ashcroft, E. A., and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *CACM*, Vol. 20, No. 7, July 1977, pp. 519-526.
3. Backus, J., "Programming language semantics and closed applicative languages," *Proc. ACM Symp. on Principles of Programming Languages*, 1973, pp. 71-86.
4. Berkling, K. J., "Reduction languages for reduction machines," *Second Annual Meeting of Computer Architecture*, 1975, pp. 133-138.
5. Burge, W. H., *Recursive programming techniques*, Addison-Wesley, 1975.
6. Davis, A. L., "The architecture and system method of DDM-1: A recursively-structured data driven machine," *Proc. Fifth Annual Symposium on Computer Architecture*, 1978.
7. Dennis, J. B., and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *Proc. 2nd Annual Symposium on Computer Architecture*, December 1974, pp. 126-132.
8. Despain, A. M., and D. Patterson, "X-tree: A tree structured multiprocessor computer architecture," *Proceedings of the Fifth Annual Symposium on Computer Architecture*, 1978.
9. Friedman, D. P., and D. S. Wise, "CONS should not evaluate its arguments," in Michaelson and Milner (eds.), *Automata, Languages, and Programming*, Edinburgh University Press, 1976, pp. 257-284.
10. Friedman, D. P., and D. S. Wise, "Aspects of applicative programming for file systems," *Sigplan Notices*, Vol. 12, No. 3, pp. 41-55, March 1977.
11. Friedman, D. P., and D. S. Wise, "Aspects of applicative programming for parallel processing," *IEEE Trans. on Computers*, Vol. C-27, No. 4, April 1978, pp. 289-296.
12. Henderson, P., and J. H. Morris, Jr., "A lazy evaluator," *Proc. Third ACM Conference on Principles of Programming Languages*, 1976, pp. 95-103.
13. Kahn, G., and D. MacQueen, "Coroutines and networks of parallel processes," *Proc. IFIP '77*, 1977, pp. 993-998.
14. Keller, R. M., "Look-ahead processors," *Computing Surveys*, Vol. 7, No. 4, December 1975, pp. 177-195.
15. Keller, R. M., "Semantics of parallel program graphs," University of Utah, Dept. of Computer Science, Tech. Rept. UUCS-77-110, July 1977.
16. Keller, R. M., "Denotational models for parallel programs with indeterminate operators," E. J. Neuhold (ed.), *Formal description of programming concepts*, North-Holland, 1978, pp. 337-366.
17. Keller, R. M., "An approach to determinacy proofs," University of

Utah, Dept. of Computer Science, Tech. Rept. UUCS-78-102, March 1978.
18. Keller, R. M., G. Lindstrom and S. Patil, "An architecture for a loosely-coupled parallel processor," University of Utah, Dept. of Computer Science, Tech. Rept. UUCS-78-105, October 1978.
19. McCarthy, J., "Towards a mathematical science of computation," *IFIP '62 Proc.,* 1963, pp. 21-28.
20. Mago, G. A., "A network of microprocessors to execute reduction languages," Department of Computer Science, University of North Carolina, March 1978.
21. Patil, S., "An abstract parallel-processing system," M. S. Thesis, MIT Dept. of Electrical Engineering, June 1967.
22. Ritchie, D. M., and K. Thompson, "The Unix time-sharing system," *CACM,* Vol. 17, No. 7, July 1975, pp. 365-381.
23. Swan, R. J., S. H. Fuller and D. P. Siewiorek, "Cm*—A modular, multi-microprocessor," *AFIPS Proc.* Vol. 46, June, 1977, pp. 637-644.