THE CONVERSION VIA SOFTWARE OF A SIMD PROCESSOR INTO A MIMD PROCESSOR.

PS-2000, AN ARRAY PROCESSOR, BECOMES AHR, A GENERAL PURPOSE LISP MACHINE

Adolfo Guzmán ~Ψ Miguel Gerzso ~ Kemer B. Norkin ρ S. Y. Vilenkin ρ

Abstract

In this paper a method is described which takes a (pure) Lisp program and automatically decomposes it (*automatic parallelization*) into several parts, one for each processor of a SIMD architecture. Each of these parts is a different execution flow—a different program—. The execution of these different programs by a SIMD architecture is the main theme of the paper.

The method has been developed in some detail for the PS-2000, a SIMD Soviet multiprocessor, making it behave like AHR, a Mexican MIMD multimicroprocessor. Both the PS-2000 and AHR execute a pure Lisp program in parallel; the user or programmer is not responsible for its decomposition into n pieces, their synchronization, scheduling, etc. All these chores are performed by the system (hardware and software) instead.

In order to achieve simultaneous execution of different programs in a SIMD processor, the method uses a scheme of node scheduling (a node is a primitive Lisp operation) and node exportation.

Summary

The general goal: automatic parallelization of *one* program.
Let us define automatic parallelization as the automatic splitting (by the system, not by the programmer) of a program into n parts, one for each processor, such that this program executes efficiently in a multiprocessor with n processors. Automatic parallelization takes care not only of (1) the subdivision into n parts, but also of (2) their synchronization, (3) scheduling, etc. Clearly, responsibities for chores (1), (2), (3), ..., can be placed upon the programmer but this will reduce by much the efficiency of the programmer. Automatic parallelization is a good goal to achieve.

Achieving the goal using a MIMD architecture. Using pure Lisp (an applicative language), the AHR machine [3,4] shows how to achieve automatic decomposition (parallelization) for a MIMD architecture. Version 1 of AHR, built at the National Univ. of Mexico, uses up to 64 Z-80's to jointly execute a single Lisp program, each micro simultaneously executing some part of it, without the programmer worrying of parallelism —in fact, the programmer or user needs not be aware that his program is running in a parallel machine—.

Achieving the goal using a SIMD architecture. A SIMD architecture can achieve automatic parallelization in the cases normally designed for it —namely, the same program is executed by all processors, each of them operating on different data—.

Can a SIMD architecture achieve automatic parallelization for cases where each processor executes a different task? That is, can we simultaneously run different programs in the different processors of a SIMD machine? NO, if we want to maintain full speed (full use) of all processors. YES, with some degradation in the degree of parallelism.

This paper describes a method which performs automatic parallelization upon a (pure) Lisp program, breaking it into several parts, one for each processor of a SIMD architecture. Each of these parts is a different program —a different execution flow—. These different programs are, nevertheless, executed in parallel in the SIMD machine. In order to achieve this, the method uses (a) node scheduling; (b) node exportation; (c) results exportation. The most important of these is node scheduling, where first all similar nodes of the name or function type are collected, and later they are executed in the normal SIMD mode.

Software conversion of a SIMD into a MIMD architecture. Since we are able to make a SIMD machine behave like a MIMD AHR machine when executing arbitrary Lisp programs, it is clear that we can use a SIMD architecture for parallel execution of

Ψ Spending his sabbatical year (1983-84) at: Electrical Engineering Dept., CIEA-IPN; National Polytechnic Institute. Apdo. 14-740. 07000 México, D.F.
~ Permanent address: Computing Systs. Dept., IIMAS UNAM; Nat'l. University of Mexico. Apdo. 20-726 01000 México D. F.
ρ Institute for Control Sciences. Academy of Sciences of the USSR. 65 Profsoyuznaya St. 117342 Moscow, USSR.

CH1929-9/83/0000/0037$01.00

a single Lisp program. By this we do *not* mean that such Lisp program is replicated in the n processors of the SIMD machine and put to work simultaneously upon different data. *We mean that such a Lisp program is automatically partitioned into different independent but interacting portions (n of them), and the nodes of each portion are executed in such a way that (generally) all the n processors are executing [necessarily the same node, although upon different data] simultaneously.*

Possible defficiencies of the approach are

(a) the amount of overhead (book-keeping, system and administrative chores, and operating system overhead) versus the amount of effective computations.

(b) the amount of time that some of the n processors remain idle, while the remaining processors are executing some node. This may be the case if some of the n processors lack node "CONS" [for instance] to execute; thus, they will remain idle while the rest proceed to CONS execution.

The different sections of this paper. While the first section explains what is automatic parallelization, the second tells us how to achieve it using a MIMD architecture, and gives some description of the AHR computer built at the National University of Mexico in 1981 under these principles. The third section outlines the solution for applying the same approach to a SIMD machine. The last section gives account of the conversion (using only software) of "PS-2000", a SIMD processor, into a device capable of automatic parallelization, which also mimics the behavior of the AHR machine, in its capability to execute in parallel different parts of a single program.

What is Automatic Parallelization?

With the advent of cheap computing power, it is reasonable to produce architectures where several processors are running simultaneously, collaborating in the common execution of a program. On the other hand, software development is still expensive. For a multiprocessor having n processing elements (called *processors*), to have to write n different programs, plus n * (n-1)/2 synchronizations, plus scheduling, etc., is uneconomical from the point of view of programmer productivity. Thus, practical use of multiprocessors achieves one of the following forms:

(1) the programmer writes one program, and all the n processors execute this same program, although upon different data. At any given time, all the processors are executing exactly the same instruction (albeit some of them may skip the instruction, becoming idle during its execution). This solution has been popular for application to numeric matrices and vectors, and has caused the development of SIMD (single instruction, multiple data) architectures. To be efficient, all the n processors must be active most of the time. This limits the algorithms for SIMD architectures to be data-independent; otherwise [as later explained] some or much parallelism —hence, efficiency— is lost.

(2) the programmer writes one program, which is automatically decomposed (by the system) in small parts and given to a pipeline to execute [8]. This approach is fruitful, but useful mainly when the same algorithm has to be applied to a large collection, or vector, of similar data.

(3) several small programs are given to a MIMD machine, and each processor executes one of them. This is possible when these programs interact nothing or little with each other, since the interactions must be explicitly considered by the programmers. This approach is useful specially when each program is independent (does not need to interact), but needs access to some common data or resource (special processor). As example, we have the Tandem multiprocessor [10] system.

(4) the user writes one program, which is automatically decomposed (by the system) into n different parts, one for each processor; the system (and not the programmer) also takes care of synchronization, scheduling, etc., associated with these parts. The programmer may be unaware of the parallel environment. The parts are run in parallel by the multiprocessor.

To the tasks performed by the system in (4), we call automatic parallelization. It can be achieved using a MIMD architecture, because the n parts which result from the automatic decomposition will be different from each other, thus necessarily requiring [we thought at first] a MIMD architecture, where a plurality of instruction flows may be achieved.

As it turns out, it is also possible to execute these different parts using a SIMD architecture! How this is possible, will be explained later.

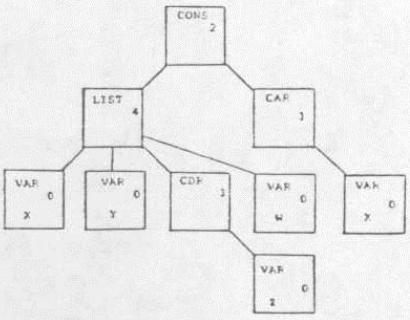
Use of applicative languages. Pure Lisp. If we remove from Lisp all the iterative parts (prog, goto, labels) and assignments (set, setq) we end up with *pure Lisp*, strictly applicative. Recursion is still there; iteration has dissappeared.

Applicative languages are specially useful for the task (4) above and for automatic parallelization, because evaluation (the replacement of an expression in Lisp by another having the same value; for instance (plus 3 5) gets replaced by 8) can then be performed in parallel. Data flow machines and applicative machines are then examples of (4) in automatic parallelization. The AHR machine [5] can be viewed as a kind of data flow machine.

Automatic Parallelization Using the AHR Machine

Outline of our approach. Account is given of our approach using the AHR machine, of MIMD type.

(1) Somehow, the program to be evaluated is converted into node form and stored into the active memory (or *grill*) of the AHR machine (Figure 'The AHR Machine'). For instance, (CONS (LIST X Y (CDR Z) W) (CAR X)) is to be stored in the grill as



Each square box represents a node. Each node has, among, others, fields for function name, space for arguments, field for "pointer to my father", and field "number of arguments not yet evaluated", or NANE. Those nodes with nane = 0 are ready for evaluation.

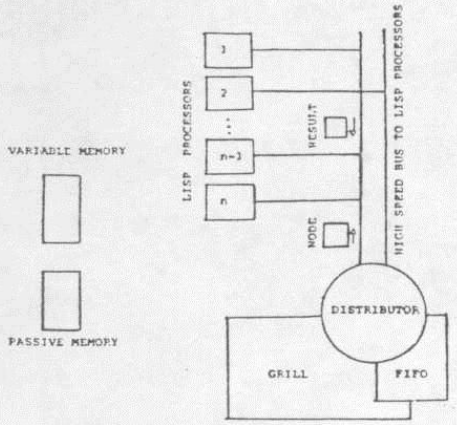


Figure "The AHR Machine"

Lisp processor 2 is ready to accept more work. The distributor fetches a node (to be evaluated) from the fifo and sends it to processor 2, while accepting the results of the previous evaluation performed by such processor. That result is stored in the grill, in a place indicated in the destination address of the result.

Such exchange of new work—previous result is performed at each cycle of the distributor.

The Lisp processors also have access (connections not shown) to the variable and passive memories.

The AHR machine communicates with the host computer by linking the passive memory of AHR to the main memory of the host.

(2) The AHR machine is a MIMD architecture formed by n (up to 64) processors. Each of them is called a Lisp processor, since it possesses in its local memory a Lisp interpreter written in Z-80 assembly language [3]. Incidentally, note that all the processors have the same Z-80 program, namely the Lisp interpreter. But, in general, each will be executing (evaluating) a different Lisp node —a different part of the user program—.

(3) At the start of the execution (as well as in any other instant in time), the Lisp processor looks into the grill for nodes ready for evaluation [those with nane =0] [11]. Each Lisp processor is either busy evaluating some previous node or looking for work (a new node with nane = 0) to do [12]. We can think that a Lisp processor "attaches" itself to a node with nane = 0, and begins to process it. First, the node is marked "under process", to prevent other processors from wanting to execute it. Using the field "function name" of the node [actually, a number], a dispatch is done to the appropriate code that handles the Lisp primitive which the node represents. Nodes with nane = 0, being ready for evaluation, have all their arguments already evaluated, and inside the node.

While evaluation is in progress, another Lisp processors are simultaneously evaluating another nodes, no one being aware of what the others are doing. No message interchange takes place. No explicit synchronization is necessary, no semaphores, scheduling, etc., are placed upon the shoulders of the programmer.

After a processor completes evaluation of its node, it places its results into the corresponding slot of the node which is the father of the node just evaluated [13]. It also decrements the nane of the father (since the father has *one less* argument without evaluation). If such nane becomes 0, it also registers the father in the fifo (See figure 'The AHR Machine'), meaning that the father is now ready for evaluation.

Then, the processor requests additional work (a new node), thus initiating a new step (3).

(4) The machine gradually evaluates the tree from the leaves towards the root, or the Lisp expression from the inside to the outside, until the tree —the program— has become a single result. Execution has finished. At this point, all processors are waiting, requesting 'more work to do', but there is none. The fifo (list of nodes ready for evaluation) is empty.

(5) Recursion is handled in a similar manner, substituting the name of the function by the lambda expression corresponding to it. This makes the tree grow. [3,4] give details.

(6) Input/output is handled through a *window* that maps part of the address space of the

AHR machine into (part of) the address space of the *host machine* [4]. Thus, the AHR machine can be thought of as a memory-to-memory processor, as a back-end processor, or as an "intelligent peripheral device", into which Lisp programs are written and from which results or evaluations of such programs are read by the host machine.

(7) The (serial) conversion of a source Lisp program (with Ascii characters and lots of parentheses) into the tree of step (1) is performed by the host machine of step (6), through a loader from disk (in the host) into the memory of the host, and *via* the window, into the memory of AHR. Printing of the results, that is, conversion of a list (stored in AHR memory as list-cells) into a sequence of Ascii characters, is also performed (serially) by the host machine, which accesses AHR memory via the window.

(8) Everything that is placed in the grill is in the form of a tree of nodes, which will eventually disappear, because it will be transformed into a result. Thus, results can not be kept in the grill. They are kept in *passive memory*, another memory of the Lisp machine, which also contains programs (written in list notation, using list cells). These programs can be later placed in the grill, to be evaluated. Such copying is done by EVAL, which transforms programs in cell notation (in passive memory) into programs in node notation (in the grill). You can think of the programs residing in passive memory as "master copies", which are necessary since everything placed upon the grill is destroyed, converted into a result, evaluated, or "cooked"; hence the same "grill".

Who places the master copies in passive memory? The host machine, during input, as explained in step 6, converting from Ascii in to cell (list) structure.

(9) And, who performs EVAL in step 8? The very Lisp processors, since EVAL is just another Lisp primitive, with the main duty of transporting a program (more likely, a piece of it) from list notation in passive memory into node notation in grill memory; leaving the program in grill assures evaluation (by the Lisp processors). Thus, EVAL can be performed in parallel: several processors can be executing EVAL at the same time, most probably on different data.

The parts of the AHR machine. Having explained in general the functioning of AHR, we now give a more detailed description of its parts. Refer to figure 'The AHR Machine'.

The *memories* of the AHR machine are the grill or active memory, where the programs to be executed reside in node notation; the passive memory where data (lists, atoms, numbers) and programs (master copies) reside in list (cell) notation; and the variables memory, holding different stacks (a tree of stacks, a cactus of stacks, a spaghetti stack) relating variable names to their values. Also, each Lisp processor has its local or private memory, holding some workspace as well as the Lisp interpreter, a collection of Lisp primitives written in Z-80 assembly language.

Also, we have mentioned the fifo or blackboard, a first-in first-out small memory associated to the grill, holding pointers to nodes in the grill with name = 0.

The *active elements* of AHR are the Lisp processors. Each is an 8-bit microcomputer, with its own local memory. They perform the conversion from nodes in the grill into results in passive memory, evaluating nodes given to them by the distributor, another active element. There may be up to 64 (this number can be easily expanded) Lisp processors. The distributor is a piece of hardware (although in the first version of AHR, built in 1981, it was a micro with associated software) that takes nodes ready for evaluation from the fifo and handles them to the Lisp processors when they request additional work. It also takes results already computed by the Lisp processors, and stores them in the corresponding place in the node of the father. Footnotes [12] and [13] should now be clear.

The *interconnection parts* of AHR are the high speed bus, linking the distributor with the Lisp processors, and carrying nodes (new work) and results (old results); the passive bus and variables bus (not shown in the figure), linking the Lisp processors to passive and variables memory; the window, connecting the passive memory to the memory of the host machine. Also, used for debugging and statistics gathering, AHR has the slow speed bus (not shown in the figure), linking the Lisp processors directly to the host machine.

Advantages of the AHR architecture. Among the advantages of AHR, we have:

* AHR achieves automatic parallelization for a MIMD architecture.
* User unaware of parallel environment/execution.
* User does not have to split his programs into parts.
* Synchronization and subtasking automatically done by the system --in fact, by the hardware--.
* No operating system is required for AHR.
* Incrementally expandible.
* If a Lisp processor stops, AHR continues running showing only slight degradation.

Current status of AHR machine. Version 1, having five Lisp processors, was finished and operational by the end of 1981 [5]. It fulfilled all the premisses/expectations of the design [3]. The machine was taken apart early in 1983, to allow for additional design and construction of Version 2. However, Version 2 still has not started to be built ($ shortage).

A sister of Version 1, built upon a PS-2000 SIMD machine [6], to be described in this paper, was designed [7] and is expected to be operational soon.

Automatic Parallelization Using a SIMD Approach

It is now desired to perform automatic parallelization in a SIMD architecture. By such architecture is meant a collection of n processors, called also processing elements (p.e.'s), which execute the same instruction upon different data. Each processor has its own private memory. A control unit (c.u.) outside the n processors has the following duties:

* To hold the program to be executed by all the processors.
* it fetches from c.u. memory the current instruction, decodes it and broadcasts it to all p.e.'s, for simultaneous execution.
* it also executes c.u.'s instructions (mainly scalar operations, as opposed to vector operations performed by the p.e.'s), which can be done in parallel with p.e.'s instructions.

Input/output is complicated, but parallel paths there exist to all p.e.'s. Generally, a modified disk (head-per-track) is used. Usually, a SIMD architecture is slave to a host computer.

Connectivity (what processor is to the right of, or above which other) among processors can be modified by execution of special c.u. instructions. Once in a particular connection or configuration, the p.e.'s can simultaneously execute instructions such as "move data from your memory address x to your neighbor at your left", and all of they give some information to their left neighbor. The best example of a SIMD architecture is Illiac IV[1].

Algorithms best suited for SIMD machines. From the above description, it is easily seen that SIMD machines will attain full speed when executing programs

(a) that apply the same algorithm to different columns (vectors, matrices) of data. For instance, if a SIMD has 64 processors, then the same algorithm should be applied to 64 different collections of numbers; and

(b) that do not depend on the data being processed. The algorithms (although, of course, not the results) should be data-independent. For instance, the average of n numbers can be expressed as an algorithm which does not depend on the values of the numbers being averaged. On the contrary, the square root of a number may be computed by an algorithm "a" that uses routine "b" to produce real numbers, when the input is possitive or zero but uses routine "c" to produce complex numbers when the input is negative. Thus, algorithm "a" is data-dependent.

Difficulties in a straightforward approach to parallelization. Data dependent algorithms can be executed by a SIMD architecture, but with substantial loss of speed. For instance, suppose we apply algorithm "a" above to an input of 64 real numbers, one in each p.e. Many of them (half, in the average) will be positive or zero, so that branch "b" of the program has to be executed by the corresponding p.e.'s, *while the others* (those having negative inputs) *wait*. After "b" is completed, branch "c" of the program has to be executed *by the other p.e.'s, while the former p.e.'s wait*. Thus, in the average, parallelism is only n/2 instead of n. This gets worse if we have nested IF's.

There is another way to execute in parallel data-dependent (which essentially means, different) algorithms, which may possibly be more efficient. A general idea of it is now given.

General Idea of the Solution

Static Part of Design

The data structure. We are using in the PS-2000 all the standard data structures already in use in Version 1 of AHR [5]. Lists, arrays, fifo, stacks, trees of stacks, etc.

Continuity of address space. On the other hand, we had to make a careful design for pointers that go outside the memory space of a processor, into the memory space of another processor, of the control unit, or even of the host machine. This is because the data stored in one processor differs from the data stored in another processor. Not only the data differs but, unlike the normal SIMD case, the structure of the data is also different. That is, in the typical SIMD case, every processor has the same array stored in the same place, beginning in the same local address, etc. The arrays possess different numerical values, when you visit the same cell in different processor memories. That is not the case of our design; in a processor memory, in locations x through x+y may be residing an array; in another processor memory, in the same locations x through x+y some lists may be sitting.

The solution was to have pointers that span the whole set of memories; part of the pointer is interpreted as a number that indicates "processor number", control unit, host processor, etc. In addition, use is made of the fact that certain structures do not point towards the host processor.

Dynamic Part of Design

The basic idea about how to place the AHR architecture inside the PS-2000 architecture, was to have the lists stored "globally" through all memories. That is, there was not going to be repetition of data. A given data resides in just one place of the PS-2000. This requires, as mentioned, global pointers.

Then, each processor "analyzes" its local memory, looking for nodes with nane = 0 and proceeds to their evaluation, substracts 1 from the nane of the father, etc. [3,4,5].

Basic difficulty. We soon hit the following difficulty: *because it is a SIMD machine, one processor can not be taking CAR of some data, while another is making CONS of two Lisp expressions.* Very strictly, the SIMD construction requires that each and every processor perform exactly the same instruction.

The solution found was, roughly described, as follows: each processor will take notice (in a local list with as many entry groups as there are primitive Lisp operations) of what operations are ready to be done (what nodes have name = 0). After this phase finishes, all the Lisp processors proceed to execute all the CAR's that need to be executed. Those processors having no CAR's or only a few of them, will soon go idle. Then, all the Lisp processors proceed to execute all the CONS'es that need to be executed. And so on.

This solution will be described in some detail below. Notice that this solution, together with a *scheduler* (a program that somehow decides what group of primitives to execute next, and how many of them: how many CAR's, how many CONS'es, etc.), *effectively converts a SIMD machine into a MIMD architecture.*

The different execution flows. Each p.e. (there are 64 of them in a PS-2000) runs a different user program. Thus, there are as many different Lisp programs as there are Lisp processors. Each program or instruction flow is composed of many atomic operations. Each atomic operation (a node) corresponds to a Lisp primitive. Each instruction flow is decomposed into its corresponding atomic operations, or nodes. Of these, some have name = 0, being ready for evaluation. Those nodes with name = 0 are inscribed into a list, during the first part of the scheduler: a list of CAR's ready for execution; a list of CDR's ready for execution, ...

Distribution of a program into n subprograms. As the program is coming from the host machine, it is converted by the c.u. into node notation, and spread over the different local memories of the p.e.'s. This is possible, since global pointers are employed. Thus, each p.e. will have, initially, a few nodes with name = 0 in its memory, where evaluation will begin.

The function that spreads a program among the memories of the p.e.'s has to have some careful design. For the function (F (G1 x) (G2 y) (G3 z)), it is better if its sons (G1 x), (G2 y), (G3 z) are placed in different processors, because then they can be evaluated in parallel. But, when the results are produced --let us call them r1, r2 and r3--, we have (F r1 r2 r3), but the results (the sons of F) are in different processors than the function F. Thus, the results have to be *exported* to the processor possessing F. Hence, spreading the arguments across the p.e.'s increases the parallelism, but also increases the exportation of results.

In the current implementation, the spreading function places the first son in the same processor as the father; each of the other sons are placed in different processors. In the example, F, G1 and x are placed in the same processors, while (G2 y) goes in the second p.e., and (G3 z) in a third.

The scheduler. The first part of the scheduler simply takes note of how many CAR's, how many CDR's, etc., each Lisp processor has ready to execute, and where they are located in local memory. This information is collected in local lists held in local memory of the p.e.'s. This collection of information is done by the p.e.'s, in parallel. Thus, each p.e. maintains a CAR-fifo, a CDR-fifo, a CONS-fifo, etc. One fifo for each primitive function.

Then, the scheduler proceeds to ascertain the best order of evaluation among the Lisp primitives. Should it be first the CONS, then the CDR's then the CAR's to be executed? Or should the order be CAR-CONS-CDR, or what? This is not easy to determine, we think. The execution first of CAR's for instance, could give rise to many more CONses (ready for evaluation) to appear. Then, the order should be CAR-CONS. On the other hand, to be looking for the optimal ordering may consume more machine time than the time saved. In the current implementation [7], the most popular primitives are executed first. This is to exploit the idea that the most popular nodes will "free" additional nodes for evaluation, which then will be evaluated *gratis*, keeping all or most of the p.e.'s busy. [14]

Then, the scheduler determines how many nodes of each type to evaluate. How many CAR's, how many CDR's, etc. For instance, suppose that after phase 1, the distribution of the number of CAR nodes ready for evaluation is 0, 0, 20, 9, 11, 10, 10, 20, assuming only eight processing elements. To order 20 executions of CAR will keep only two processors busy; two will never have work to do (because they have no CARs) and four of them will become idle at the middle in time. Perhaps would have been better to order only 10 or 11 executions. In this manner, the processors having 20 CARs would have to wait for the next "CAR execution cycle", leaving some CAR nodes unexecuted in this cycle.

The scheduler makes the above determination based in the count of phase 1, which is a static count. For instance, knowing that a processor has 11 CAR's ready for execution really means that it has *at least 11*, since during execution of another primitives --and even of the CAR's themselves--, more CAR's ready for evaluation are likely to appear.

Then, the scheduler proceeds to the execution (done by the p.e.'s) of the determined number of primitive 1, then the determined number of primitive 2, etc. That is, "execute 11 CAR's, then 23 CDR's, then 15 CONS'es, ..." In this phase, the scheduler asks all the Lisp processors to execute the same primitive function (or to remain idle), although, of course, over different data.

After finishing the above, the scheduler looks for more nodes with name = 0, starting the first part of another cycle. In general, the execution of nodes with name = 0 tends to make zero the names of their parents; in this way more nodes with name = 0 are produced.

The execution ends when the scheduler finds (in its phase 1) no nodes with name = 0.

Node exportation. It often occurs that a processor needs to perform a primitive operation over data which resides in another processor. Certain primitive operations are capable of being performed even if data resides elsewhere (for instance, if the needed information is *inside* the global pointer). Nevertheless, most primitive operations need to be done "locally" that is, the processor that owns the data should execute it. In order to accomplish this, a node is exported to the processor owning the data, if it ever happens that such node is tried for execution, only to find that its data is elsewhere. This is easier than bringing the data to reside together with the node that wants to manipulate it.

Exportation takes place in two phases:

* In phase e1, a node to be exported is placed in a "list of nodes which desire to be exported". Phase e1 occurs all the time. When it is impossible to perform an operation signaled by a node, because its data is not locally available, then "Phase e1" is called, which annotates this node into the list, and the node is considered "formally exported". (The node is still waiting for execution).

* In phase e2, exportation actually takes place. All the p.e.'s use the inter-processor communication facilities, and all proceed to exchange nodes.

  After phase e2, nodes imported are considered to "belong" to the importer processor, and we are sure that it can handle them.

  The main idea is: when you have a node, try to do as much work as it is possible to perform locally; when you can't do any further work, export it. But do not export it to any processor; export it precisely to the processor that owns the data that is "causing the problem". In this way, we try to guarantee that there are no nodes "perpetually circulating around processors" and getting no attention from any of them. More details in [7].

What happens if there is a function, (PLUS r1 r2 ... rn) that wants to add all its arguments (already evaluated; shown as results r_i), but every argument resides in different p.e.'s, so that, no matter to whom you export the node PLUS, it can not do anything about it? We believe that this should not happen, if the following steps or precautions are taken:

* provide space in the node for fixed and real numbers. This will solve the problem with PLUS above.

* carefully design the primitives of the language, so that no one of them depends on more than one "non-present" arguments (an argument is not present if it needs more information than that carried in the pointer).

* if necessary, decompose the primitives into another primitives having dependency in at most one "non-present" argument. This decomposition can be done by a macro expansor at loading time (performed by c.u. or by the host computer). For instance, suppose that the language has a primitive (PLUSCAR x1 x2) that adds the CAR of list x1 to the CAR of list x2. That is (PLUSCAR A B), when A is (3 4) and B is (5 6), gives a result of 8. Then A and B are "non-present" arguments because, even if they are already evaluated --their values being (3 4) and (5 6) respectively--, the pointer to (3 4) does not contain information about "3". Then, PLUSCAR should be deleted as a primitive, expanding it at loading time into something like (PLUS (CAR A) (CAR B)), or (PLUSS (CAR A)B). In this last case, PLUSS has B as the only "non-present" argument, and thus is still safe.

Exportation of results. Once a node is evaluated, its result is sent to the node higher up (to its father) in the tree. If the node of the function which is to receive the result is not in the same processor as the node [that is, if the father of a given result is in another processor], then the result is exported (as a "results node", something similar to QUOTE) to the processor which has the function node (the father).

HOW TO REPROGRAM A SIMD MACHINE TO MAKE IT BEHAVE LIKE A MIMD

Reasons to change the philosophy of operation of PS-2000. Although the way of working of a SIMD machine is simple and well understood, we want to change it to a MIMD machine "mode of operation" due to the following reasons:

* In the MIMD case, we can have algorithms that depend on the data and even in this case full parallelism is sustained.

* It is desired to do other operations, such as symbol manipulation, instead of simple numerical operations.

In addition, we have reasons for choosing the PS-2000:

* it is the multicomputer designed, built and available at the Institute of Control Sciences, where this work was done.

* this computer is widely available, commercially, in the Soviet Union.

* It is a powerful computer. Each of its 64 processors is a minicomputer both in speed, in word size (24 b) and in memory size (64 K words).

In addition, we have reasons for choosing the AHR computer as the target architecture:

* the AHR machine works, and it has a proven and sound design.

* the AHR machine is easy to program, differing in this from more conventional MIMD machines.

* the AHR machine was built and existed at IIMAS-UNAM (Mexico) and there was a great deal of familiarity and experience with its design, its architecture and its functioning.

* while doing design and construction of AHR, some improvements came to mind, that were postponed to a later version. There was some desire to bring these improvements and variations into existence.

Only software was used to accomplish the change. During the design and construction of AHR it was soon learned that, if we have both the ability to specify the hardware and the software (that is, if the design engineers are allowed to change both hardware and software), the resulting structure is more easily tuned to requirements than if we can specify or change only hardware (or only software). Thus, it was our original idea to modify both software and hardware of the PS-2000, in our efforts to convert it into an AHR-like machine. Nevertheless, this was not done. In fact, we did not do any hardware change. We accomplished the conversion using only programs --software, that is--. The reasons for doing this rather contrived design were:

* time. We had only two months to learn about the PS-2000, to design the changes and to begin implementing them. To have gone into the detailed circuits of the machine, and through the extensive documentation it possesses, would have meant more additional efforts. Although this, in return, would have produced a more efficient result (a new PS-2000 running in AHR mode more efficiently, faster than the current machine).

* portability. If we do *any* hardware changes to our PS-2000, it ceases to be a PS-2000, in the sense that it no longer behaves as its PS-2000 sisters installed elsewhere. To these sisters, the same hardware changes would have to be done in order to run in AHR-mode. Also, our PS-2000 may even stop running programs that were previously running in an unmodified PS-2000. Thus, it was decided not to touch the hardware, so as to

  * allow our installed changes to run in any PS-2000 machine;

  * allow the PS-2000 to continue running old PS-2000 programs.

General characteristics of the design. The general organization of the PS-2000 Lisp is such that the input and the output routines reside in the host machine, and the evaluation routines reside in the PS-2000. The consequences of this organization are several. To begin with, the host machine always retains the oblist (object list); that is, the literal atoms. The PS-2000 has an indirect reference [not to be confused with indirect addressing] to the elements of the oblist by the fact that the first cells of the property list in the PS-2000 reside at the same addresses as the literal atoms in the host machine. Next, the Lisp evaluation system, which includes EVAL, resides completely in the PS-2000, and in particular in the memories of the c.u. Parallel execution of th Lisp primitives is achieved by the p.e.'s. Other modules also reside in the memories of the c.u., such as the interprocessors communication routines among the p.e.'s, the memory management of the entire system, etc.

How a Lisp expression is evaluated. The process for evaluating a Lisp expression is begun by a user typing a Lisp expression. The input routine that reads this expression (the Reader) in the host, converts the character string representation of the expression into an internal representation consisting of nodes and pointers. Some syntatic verification of the expression is performed by the reader, such as balanced parentheses and the like.

Then, the expression (in internal form) is sent to the PS-2000 for evaluation. Upon arrival in the PS-2000, the expression is spread over several p.e.'s. The number of processors that are required for evaluating the expression depends upon the size of the expression and the total number of processors that the PS-2000 configuration has. The expression is then evaluated in the PS-2000. Finally, the result of the evaluation is sent back to the host machine. In it, the result (a Lisp expression) is converted back into a character string by the output system (Printer).

Both the reader and printer are standard Lisp input/output routines as found in sequential machines. However, the evaluation process in the PS-2000 is not standard. It works as follows:

The system continuously maintains a table of the nodes to be processed. The table is ordered according to the so-called popularity of the node function type. This means that the table is a representation of the demand for nodes to be processed; the nodes that have the most entries in the table are considered first, the nodes with next-to-the-most entries in the table are considered second, and so on. The table is updated at certain times during the processing, but not after each node is processed. The table resides in c.u.'s memory.

The popularity table is used by the system to decide what node to process. Once this decision has been made, the system transfers control to the routine that evaluates such node.

The consequence of this evaluation may be more nodes, which may or may not be registered in their respective fifos. Independently of whether new nodes are created or not, the evaluation of the original node is finished, possibly temporarily, and control is returned to the system. Then the process begins over again by selecting a new node to evaluate according to the popularity table.

It should be emphasized that much of the process of getting the node out of the Grill and evaluating it is done in parallel. However, if a node is selected but a given p.e. does not have such a node, then that p.e. waits until another node function type, which the p.e. may have, is to be processed.

It has been mentioned before that an expression is loaded by spreading it over several processors. The consequence of this action is that at the time a node is evaluated, the node may require one or more of its parameters to be accessed because they reside outside the processor of the node being evaluated. Therefore, the system suspends the evaluation on the node temporarily and registers the node to be exported to the processor where the parameter resides. At some time later, the node is exported, the parameter is accessed and its value is used to help to evaluate the node. If there are other parameters in other processors, then the node is re-exported to the processors of the parameters, and subsequently evaluated. In the end, the node is evaluated, and its result is sent to the father node (the node above the evaluated node in the tree). And as in the case of the parameters, if the father resides in another processor, the result is exported to the location of the father node.

As in the case of standard Lisps, the PS-2000 Lisp maintains its various memories by garbage collection in parallel. This is done when the cell lists are exhausted. Processing of Lisp expressions is suspended until the free lists are reconstructed again.

Other related work. Strong [9] analyzes the problem of how to sequence (schedule) different programs (flow graphs) residing in the processors of a SIMD machine, so as to be optimally executed by it. His solution has theoretical insights, while ours is a "practical" bridge between two existing machines.

CONCLUSIONS

* It is possible to attain automatic parallelization in a MIMD machine, as the AHR machine shows.
* It is possible to attain automatic parallelization in a SIMD machine, as the emulation of AHR by the PS-2000 shows. More over, this emulation requires no hardware modification.
* The paper describes a procedure which enables a SIMD architecture to execute (in parallel) different programs, each one residing (as nodes) in each processing element. Thus, it is possible to mimic the behavior of a MIMD machine using a SIMD architecture.
* The above execution seems to be rather efficient, because we can know and control (with the scheduler) how many processors are going to be idle, during the execution of a given type of node.

Recomendations for further work.

* Finish the ongoing implementation of the scheduler, [7], and the parallel garbage collector.
* Measure the efficiency of some critical parts:
  * the % of time that some p.e.'s wait because they lack the type of node being currently executed;
  * the % of time that the scheduler takes. That is overhead due to the scheduler and to handling the queues of nodes --one queue for each Lisp primitive function--.
* Diminish, if needed, the overhead due to the scheduler, by
  * improving it through software changes and theoretical considerations;
  * transferring some time-consuming part of it to hardware, inventing suitable machine instructions for p.e.'s and c.u.

ACKNOWLEDGMENTS. We want to thank Professors Herbert Freeman (USA) and Goffredo Pieroni (Italy) for the opportunity to present this material at the NATO Advanced Study Institute.

This paper is based on the work done [6,7] under the Joint Research Agreement between the USSR Academy of Sciences and CONACYT, the National Council for Science and Technology (Mexico).

Work herein described has partially supported by CONACYT (Grants PVT EE NAL 81 1211 and 14112H22-044)

We acknowledge our institutions, the Institute for Control Sciences and IIMAS-UNAM; specially the members of the AHR project [5].

Finally, A. Guzmán acknowledges the fruitful research environment provided at the Electrical Engineering Department of CIEA-IPN by Profs. Juan Garduño (Dept. Head), Héctor Nava Jaimes (Director of CIEA-IPN) and Dr. Manuel Ortega (Undersecretary of Public Education for Technological Research, Federal Govt. of Mexico).

REFERENCES

1 Bouknight, W.J., et al. The Illiac IV System, Proc. IEEE 60 4 April 72 369-388.

2 Glushkov, V.M., et al. Recursive machines and computing technology. Proc. IFIP 1974, North Holland, 65-70.

3 Guzmán A. A parallel heterarchical machine for high level language processing. In Languages and Architectures for Image Processing, M.J.B. Duff and S. Levialdi (eds). 1981 Academic Press, 230-244. Also in: Proc. 1981 Int'l Conf. on Parallel Processing, 64-71.

4 Guzmán, A. A heterarchical multi-processor Lisp

machine. Proc. 1981 IEEE Workshop on Computer Architecture for Pattern Analysis and Image Data base Management. IEEE Publication 81CH-1697-2, pages 309-317.

5 Guzmán A., and Norkin, K. The design and construction of a parallel heterarchical machine: final report of phase 1 of the ARH Project. Technical Report AHR-82-21, AHR Lab, IIMAS, Nat'l Univ. of Mexico 1982.

6 Guzmán, A., Gerzso, M., Norkin, K., and Kuprianov B. The PS-2000 SIMD computer: technical description and instruction set. Tech. Report AHR-82-23, AHR Laboratory IIMAS. Nat'l Univ. of Mexico 1982.

7 Guzmán, A., Gerzso, M., Norkin, K., and Vilenkin, S.Y. Functional design of Lisp interpreter for the PS-2000 SIMD computer. Technical Report AHR-82-24, IIMAS, Nat'l University of Mexico, 1983.

8 Russell, R.M. The Cray-1 computer system C ACM 21 1 Jan 78, 63-72.

9 Strong, H.R. Vector execution of flow graphs J ACM 31 1 Jan 83 186-196.

10 Tandem NonStop II system description manual, Vols 1 and 2. P/N 82077 Tandem Computers Inc. Cupertino Ca, USA. April 1981.

11 Glushkow [2] postulated this search. To avoid it, AHR uses a fifo holding nodes ready for evaluation; they are handed out by the *distributor*.

12 The Lisp processor does not actually look for more work to do; instead, it just "signals" to the distributor that it wants more work; the distributor accesses the fifo and provides a new node to the processor.

13 Actually, the Lisp processor just requests *that thing* to the distributor, which actually does the placement of the result into the father, as well as the decrementing of the nane of the father and its optional inscription in the fifo.

14 To give an example, let us suppose that the scheduler has just run its first part and it counted 12, 14, 7, 9, 10,... CAR's and 5, 8, 2, 4, 6,... CONS'es, in processors 1, 2, 3, 4, 5, ... Using this information, it decides to go through 10 (parallel) executions of CAR's and 6 (parallel) evaluation of CONS'es, in that order. Let us suppose that the evaluation of the CAR's has generated 2, 1, 3, 2, 0,... additional CONS'es ready for evaluation. That is, there are new 7, 9, 5, 6, 6,... CONS'es ready. When coming to the evaluation of the CONS'es [which was already decided to be 6], each processor evaluates 6 CONS'es (or less, if it had fewer ready). Efficiency was lost only in processor 3 (who evaluated 5 CONS'es and wasted one CONS evaluation cycle), as opposed to the case when no additional CONSES were made ready. In this last hypothetical case, since the CONS count remained at 5, 8, 2, 4, 6, ..., processors 1, 3, 4, ... would be below 100% efficiency. The example shows that, without spending additional computing time, the additional CONS'es made ready by the CAR evaluations, improved the efficiency of every processor who had fewer CONS'es that the number (six, in our example) of executions chosen by the scheduler.

Those processors with 6 or more CONS'es did not improve their efficiency; it deteriorated neither: all of them remained busy during the 6 executions of CONS'es, and efficiency was 100% whether more CONS'es appeard or not for those processors.

Between a scheduler intervention and the next, what is said for CAR's with respect to CONS'es is also true of CAR's for any other Lisp primitives: the execution of primitive i will improve the efficiency of execution of primitive j, if j is executed at any time (between two consecutive scheduler interventions) after i. Thus, the popular primitives should be executed first.