Parallel VLSI Architecture Emulation and the Organization of APSA/MPP

by

John T. O'Donnell

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 203

Parallel VLSI Architecture Emulation and the Organization of APSA/MPP

by

John T. O'Donnell

October, 1986

To appear in the *Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation,* Sept. 1986, NASA Goddard Space Flight Center.

PARALLEL VLSI ARCHITECTURE EMULATION AND THE ORGANIZATION OF APSA/MPP

John T. O'Donnell
Computer Science Department
Indiana University
Bloomington, IN 47405

ABSTRACT

The Applicative Programming System Architecture combines an applicative language interpreter with a novel parallel computer architecture that is well suited for VLSI implementation. The Massively Parallel Processor can simulate VLSI circuits by allocating one processing element in its square array to an area on a square VLSI chip. As long as there are not too many long data paths, the MPP can simulate a VLSI clock cycle very rapidly. The APSA circuit contains a binary tree with a few long paths and many short ones. A skewed H-tree layout allows every processing element to simulate a leaf cell and up to four tree nodes, with no loss in parallelism. Emulation of a key APSA algorithm on the MPP resulted in performance 16,000 times faster than a Vax. This speed will make it possible for the APSA language interpreter to run fast enough to support research in parallel list processing algorithms.

Keywords: parallel simulation, VLSI, tree architecture, applicative language, functional language.

INTRODUCTION

The Applicative Programming System Architecture research project (Refs. 4, 5, 6, 7) (APSA) combines VLSI hardware design, computer architecture, and programming language research in one unified design for a high level language implementation. The hardware design of APSA uses parallelism extensively, making it impossible to run realistic simulations on sequential processors. This has been a major impediment in the research on APSA. Fortunately, the Massively Parallel Processor (MPP) provides exactly the kind of parallelism needed by APSA's hardware, as well as other parallel VLSI architectures. This paper discusses the goals and structure of the APSA research, and then it describes the methods used for programming the MPP to emulate APSA's parallelism. Finally, it discusses the suitability of the MPP's architecture for this work.

The tasks of designing a programming language implementation, a computer architecture for executing it, and a low level fabrication of the hardware usually proceed independently of each other. As a result, there is a small set of standard basic ways to relate these levels of abstraction to each other: instruction sets provide an interface between language and architecture, while registers, data paths and addressable memories relate the architecture to the hardware. These standards greatly limit the types of architecture and language that are available.

The idea of "high-level language architectures" attempts to improve a computer's performance by breaking away from the standard kind of instruction set, replacing it with a set of instructions especially well suited for a particular programming language. Several manufacturers have built such systems, supporting languages such as Algol, Cobol and Lisp. But these architectures are still limited because they are built with conventional hardware techniques and components.

The Applicative Programming System Architecture research is moving toward improved performance by designing the low level hardware, the high level architecture and the language translator together. Applicative programming languages (also called functional languages) have many advantages over conventional imperative languages, but they are notoriously slow in conventional implementations. The problem is that standard instruction sets give poor support for the basic data structure operations in applicative languages. Furthermore, the standard methods for designing high-level language architectures don't help much, because conventional hardware techniques prevent the architecture from supporting some of the most useful operations.

Research in applicative programming languages is very active, and many examples exist. The current work on APSA is focusing on two particular languages: Scheme (a dialect of Lisp) and SASL. These languages and their relatives play a central role in artificial intelligence, and their use in other fields is expanding. Thus it would be inherently useful to learn how to implement such languages on the MPP.

The APSA architecture contains instructions for manipulating lists, vectors, environments, and continuations — the key data structures for applicative languages. These instructions perform many operations in a constant amount of time which would require iteration in linear time with conventional instructions. For example, consider a program that needs to find the last element of a list. This normally requires a loop, where every iteration follows a pointer from one list element to the next. APSA has an instruction that can find the last element of a list in one cycle, and a similar instruction allows indexing into a list to find the *n*th element in 1 cycle. Of course, Fortran can do the same thing with an array on a conventional computer — but Fortran cannot then insert a new element into the middle of the array in constant time. The point is not that APSA supports a fast data structure operation, but rather that it supports many fast operations that can be applied to the same data structure (Ref. 6).

The power of APSA's instruction set results from the ability of its memory to perform parallel logic operations on data, in addition to just storing the data. Each basic storage unit in the memory, called a cell, can hold one word of data. An APSA word corresponds roughly to a Lisp cons box or a Fortran array element. The cells are organized into a linear address space, just as in conventional computer memories. However, each cell also contains low-level processing capability. In addition, there are two basic kinds of internal memory operation that require parallel hardware; these operations lead to the differences between APSA memory and conventional memory.

- *Shift.* Each cell reads the contents of its left or right neighbor, performs a logic operation on that value and its current state, and stores a new value — which may be its old value, the contents of its neighbor, or another value (resulting from a sweep operation). If a number of adjacent cells all store the contents of their left neighbor, the net effect is that a sequence of words shifts right. Since each cell's logic hardware controls the value that it stores, some cells may do a shift while others remain unchanged. This allows the memory to insert or delete a word in the middle of a data structure, in one clock cycle.
- *Sweep.* The memory performs a global logic operation on the contents of all the cells and an input from the memory controller. Part of the logic hardware computes a value that it returns to the controller; this is how the controller is able to fetch data from the memory. Other parts of the logic compute independent values to be sent to each cell in the memory. If we ask for arbitrary logic operations, the complexity of the hardware would get out of hand. Therefore, APSA supports only logic operations that can be implemented with a binary tree of combinational logic components. This restriction leads to manageable but powerful hardware.

The shift operation requires data paths connecting each pair of adjacent cells, and the sweep operation requires a binary tree of combinational logic whose root is a port to the memory controller and whose leaves are the cells. Figure 1 illustrates this organization.

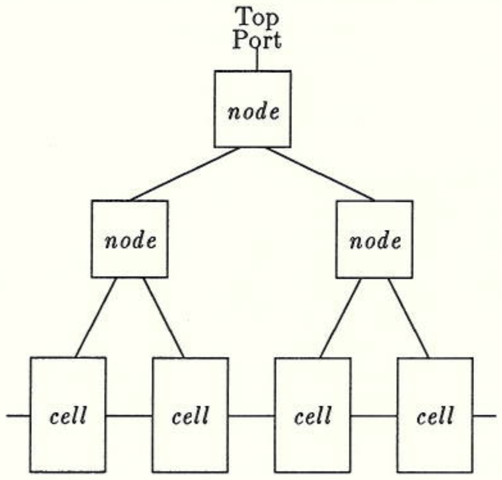

Figure 1. APSA Memory Organization

The reason that APSA algorithms cannot be used in implementing applicative languages on ordinary computers is simply that simulating the shift and sweep operations takes too much time. At the very least, a computer needs several thousand words of memory to be able to run interesting programs. But that would require a sequential simulator to operate separately on several thousand cells and several thousand logic tree nodes — just to compute the effect of APSA's memory during one clock cycle. APSA's algorithms are fast, but they are not fast enough to overcome this speed penalty of 3 to 6 orders of magnitude.

VLSI technology (very large scale integrated circuits) is good at implementing highly parallel systems with large numbers of small components, as long as the data paths connecting the components do not waste too much area on the chip. There is a standard method for laying out a binary tree on a square VLSI layout, where the subtrees appear on opposite sides of the node above them. For nodes at an even-numbered level, the subtrees are placed to the east and west; nodes at an odd-numbered level have their subtrees placed to the north and south. This results in the "H-tree layout" (Ref. 3). It also turns out that adjacent cells in an H-tree can be connected efficiently.

The original idea in the APSA project was to exploit the properties of VLSI in order to build a parallel data structure memory. This VLSI design is currently in progress, but there is another problem: details of the design of the cells and the logic tree have a profound impact on the large scale performance of the applicative programming language. For example, a low level decision on the representation of data in a cell may affect the frequency of garbage collections. Furthermore, the system must run for thousands of cycles before such effects show up. This has lead to a very serious difficulty: simulation of the VLSI layout is far too slow to allow enough experimentation with the system's overall behavior to be able to make correct decisions about details of the VLSI layout!

The MPP provides exactly the same kind of parallelism that VLSI does. Each processing element (PE) combines storage with logic, and is connected to nearby PEs. Thus it is possible to map a square VLSI layout onto the square MPP array, with each PE performing the function of the corresponding area of the chip. The MPP also has a limitation: long distance communications are relatively slow because messages must be sent along a path consisting of adjacent PEs. Long data paths also lead to poor performance in VLSI, so good chip designs tend to have a small number of long data paths and a very large number of short ones. Such designs are well suited for implementation on the MPP.

Ref. 8 describes the basic methods of digital circuit simulation and outlines how the MPP can simulate VLSI circuits. The next section discusses APSA data structure operations and parallelism, and the remainder of this paper is concerned specifically with the implementation of APSA on the MPP. The APSA/MPP program directly transfers all of APSA's parallelism into MPP parallelism, so it seems more appropriate to call it an emulator rather than a simulator.

DATA STRUCTURES IN APSA

A complete description of the data structures and algorithms for APSA is beyond the scope of this paper, but a brief example will clarify the general ideas. The most interesting algorithms are for maintaining aggregate data structures, environments, continuations and for performing garbage collections. These algorithms are too complex to discuss here, so this section describes operations on a simple combined list/vector data structure.

APSA can represent the list `(a b d e)` in a compact form by storing the list elements in consecutive neighboring cells. Each cell contains a type field; we are only concerned with the cells that contain a *value*. In addition, each cell contains several flags. The *attached* flag (denoted by ▷) indicates that a cell represents a value in a list that continues on into the next cell. Therefore all the list elements except the last will be in cells with ▷ set. This leads to the following representation:

	▷	▷	▷		
	val	*val*	*val*	*val*	
	a	b	d	e	

Now suppose that we want to insert a new element "c" into the list just after the occurrence of "b". This takes three instructions. First, the program must locate the point where the insertion should be made, using an associative search. The *match* instruction does this, and sets the *select* flag (denoted by •) in that cell:

	▷	• ▷	▷		
	val	*val*	*val*	*val*	
	a	b	d	e	

As the program inserts c into the list, it must move all the elements before it to the left, leaving room for the insertion. This will destroy the contents of the leftmost cell, which is part of the available space list. However, only the cells to the left of the point of insertion should store the contents of their right neighbor; cells to the right of the insertion must remain unchanged. Therefore the program must compute another flag called *mark* (denoted by ∘) which will control the shift. The *mark to select* instruction sets ∘ in all the cells that lie to the left of the selected cell:

∘	∘ ▷	• ▷	▷		
	val	*val*	*val*	*val*	
	a	b	d	e	

Now the program can issue the *insert (c)* instruction, which causes each cell to perform an operation that depends on its flag settings. Cells with ∘ set will store the contents of their right neighbor, the cell with • set will store the instruction operand c, and all other cells remain unchanged. The net effect is to insert c into the list, while destroying one element of the available space pool:

∘ ▷	∘ ▷	• ▷	▷		
val	*val*	*val*	*val*	*val*	
a	b	c	d	e	

Note that such an insertion always takes three cycles, regardless of where the insertion takes place. List processing systems based on linked representation can also quickly insert a new element. However, suppose that the program now needs to find the fourth element in the list. The linked list representation requires iteration to do this; the time to find the nth element is proportional to n. APSA can index directly to the nth list element, just as if the list is an ordinary vector. First, the program uses the *match* instruction again to select the beginning of the list:

● ▷	▷	▷	▷		
val	*val*	*val*	*val*	*val*	
a	b	c	d	e	

Next, the *index (n)* instruction directly locates the nth element in one cycle. Thus *index (3)* produces:

▷	▷	▷	● ▷		
val	*val*	*val*	*val*	*val*	
a	b	c	d	e	

The point of this example is that *one data structure supports both vector and list operations.* This leads to a richer set of data structures and algorithms than conventional computers provide. Notice that APSA used parallelism inside each instruction, allowing it to perform a task using fewer instructions than a conventional system.

APSA uses more complex hardware to execute fewer instructions. This leads to a crucial question: *does APSA's speedup in number of cycles overcome its overhead in cycle time?* Consider first the speedup. Indexing to the nth element of a list normally takes time $O(n)$, while APSA does this in time $O(1)$. Many other APSA algorithms show the same $O(n)$ reduction in the number of instructions executed. Calculating the overhead in cycle time is a more subtle problem. At first sight, it appears that APSA's cycle time is slower than a conventional machine's by a factor of $O(\log n)$ because of the tree. We normally think of a computer's RAM memory as having a constant access time: as the problem size grows the memory does not slow down. However, that is only true as long as the memory is large enough to hold the problem. And it turns out that as the size of a RAM memory increases, its access time slows down. There are two distinct reasons for this. First, a RAM uses a decoder to select the addressed word — *and a decoder is a tree of combinational logic.* Second, as a RAM grows in size its wires become longer, and the electrical delay across the wires becomes significant. Of course, both of these factors affect APSA: it contains log time delay in its tree, and as its memory grows its data paths become longer, requiring more communication time. The final result is that

- the APSA memory has the same asymptotic cycle time as a RAM memory, although it is slower by a constant factor K, and
- fewer cycles are needed to run an algorithm on APSA than on a conventional machine.

Table 1 compares the cycle time for RAM vs. APSA using three different measures: counting one time unit per cycle, considering the delay through the combinational logic trees, and considering also the electrical wire delay.

measure	*RAM*	*APSA*
constant	1	1
logic delay	$\log n$	$\log n$
wire delay	$\sqrt{n}$	$\sqrt{n}$

Table 1. Cycle time complexity

An APSA cycle is slower than a RAM's by a constant factor K, which determines the actual attainable speed. Fortunately, the MPP sheds light on the value of K. The MPP contains a tree of logic that computes the logical *or* of the P register in all 16,384 PEs. This corresponds roughly to a 1-bit upsweep, and it takes about half a microsecond. The APSA instructions require several bits going both up and down, but it should still be feasible to attain a hardware cycle time on the order of 10 microseconds, which would lead to excellent performance. However, the emulator for APSA that runs on the MPP takes several hundred microseconds per cycle. This is good enough for extensive experimentation, but will probably not be competitive with Lisp implementations on conventional computers.

The preceding discussion concerned parallelism within data structure operations. There is also another level of parallelism. The APSA memory can perform a parallel data structure operation on several different data structures — in parallel. For example, the Lisp **mapcar** functional applies a function to every element in a list. In some cases, APSA can execute all these applications simultaneously. For example, the *index* operation can index into many lists at the same time. However, storage allocation can lead to problems. If a program tries to perform many **cons** operations at the same time, the system may require a number of cycles to obtain all the new storage words that it needs. Further work is needed to assess the potentials and limitations of this form of parallelism.

A promising method for implementing applicative programming languages is to translate them into combinators, which can then be reduced on an appropriate architecture. APSA has the ability to reduce many combinators simultaneously, although a number of problems with storage allocation remain. This looks like a good approach for designing parallel combinator reduction machines.

THE SKEWED TREE LAYOUT

The basic idea in mapping the VLSI memory design onto the MPP array unit is to simulate an area of the chip with each processing element. If that is done in a straightforward manner, only 1/4 of the PEs will

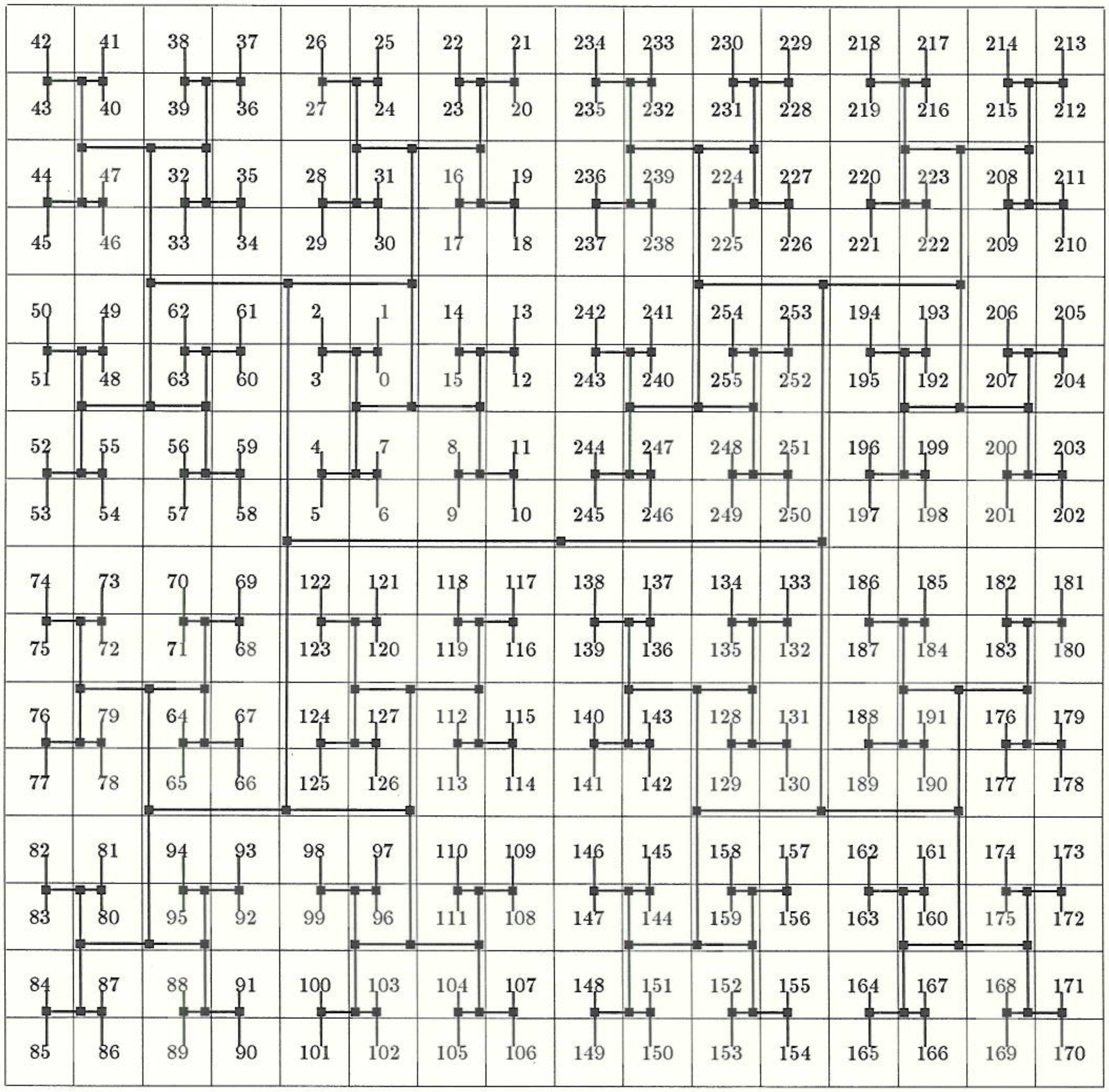

Figure 2. Skewed H-tree layout

simulate a memory cell; the others will be dedicated to the tree nodes and the data paths connecting the nodes, while some of the PEs would be wasted entirely. The original VLSI layout for APSA had these properties (Ref. 8). There is nothing inherently wrong with this approach, but it is possible to go from a 4,096 cell layout to a 16,384 cell layout without losing any potential parallelism.

By placing one memory cell in each PE, the emulator can perform all the cell logic operations in parallel. The nodes never execute in parallel with the cells anyway, because of data dependencies in the sweep algorithm. This means that we lose no parallelism if each PE simulates a node in addition to a cell. Furthermore, the nodes execute simultaneously only at one level in the tree at a time. For example, on an upsweep all the cells execute, then all the bottom level nodes, then all the nodes at the next higher level, etc. Therefore a PE can simulate a cell and several nodes, as long as all of its nodes appear at different levels in the tree.

These considerations lead to a "skewed H-tree" layout that allows all 16,384 PEs to contain a cell. Each data path between two subtrees is skewed to one side or the other, so that its node and path elements actually lie on top of one of the subtrees (forcing some PEs to hold several nodes). We place the node and path on whichever side yields the shorter data paths (and hence the greatest speed).

Figure 2 shows the skewed layout for a 16×16 array, with 256 cells. Each cell is represented by its address (the leftmost cell is 0 and the rightmost is 255). Small black squares indicate nodes, while the data paths connecting nodes appear as thick lines. Thin lines show the PE array, making it easy to see what cells, nodes and paths lie in any PE.

This layout consists of a recursive sequence of squares. A *square* is a structure with

- a *central node* near the center (i.e., in one of the four PEs adjoining the true geometric center),
- data paths from the central node going east and west to the *eastern node* and the *western node* respectively, and
- data paths leading from the eastern and western nodes to the *northern subtrees* and the *southern subtrees*.

For example, consider the 2×2 square at the northwest corner of Figure 2, consisting of cells 40–43. The central node is at level 2 in the tree (there are 2^2 cells below it) and it happens to be in the PE to the southeast of the geometric center of the 2×2 grid: thus this node has *SE parity*. The eastern and western nodes, both of level 1, appear in cells 40 and 43 respectively. Since the northern and southern subtrees are at level 0, they just consist of cells. Notice that 2×2 squares appear with all four possible parities: NE, NW, SE and SW.

The largest square (level 8, in cell 245) illustrates several points that affect the communication algorithms. The eastern node has W parity, in order to minimize its data path. For the same reason the western node has E parity. Both of these level 7 nodes inherit their N parity from the central node. One of the data paths from a node to its subtrees will always be shorter than the other one — and the communication algorithms must be able to handle this.

The skewed H-tree has a useful property: no PE represents more than four nodes, regardless of the size of the layout. That is significant because every PE must allocate memory for one cell and all its nodes. Each PE contains only 1024 bits, so an unbounded number of nodes per PE would take too much space.

The sweep algorithm needs to know how long the tree data paths are in order to control its shifting loops. A simple recurrence equation yields the length d_i of a long path from a node at level i down to its subtrees (the short path has length $d_i - 1$):

$$d_1 = 1 \qquad d_2 = 1 \qquad d_3 = 1 \qquad d_4 = 1$$

$$d_i = d_{i-1} \quad \text{for } i \text{ even, } i > 4$$

$$d_i = 2d_{i-1} \quad \text{for } i \text{ odd, } i > 4.$$

The sweep algorithm computes these recurrences as it moves up or down the tree. Thus it must periodically multiply the current d by 2 on an upsweep and divide d by 2 on a downsweep. The sweeps run entirely in the PE Control Unit, without any help from the Main Control Unit, and the PECU has no scalar arithmetic facilities — so the sweep algorithm does its multiplications and divisions using Peano arithmetic, completely in parallel with other operations!

It is possible to place data paths connecting neighboring cells in the existing tree structure. Using that method, a shift operation would send data from a cell part way up the tree, and then back down the other branch of the tree to the neighbor. This is probably the best way to implement the cell neighbor data paths in a VLSI layout, because it reduces the number of wires that must cross each other. However, it is faster on the MPP to route data in a straight line from each cell to its neighbor, and the APSA emulator takes this approach.

The shift algorithm operates by sending the contents of each cell to its neighbor within one layout square. It repeats the process for each square, working from the smallest square size (2×2) up to the largest (normally 128×128). At each square size, all squares

d k			*sequence of operations*		
0 1			*rw*		
0 2			*rw rw*		
0 3			*rw rw rw*		
0 4			*rw rw rw rw*		
1 1	*r*	→	*s*	→	*w*
1 2	*r sr*		→		*w sw*
1 3	*r sr*	→	*wsr*	→	*w sw*
1 4	*r sr*	→	*wsr wsr*	→	*w sw*
1 5	*r sr*	→	*wsr wsr wsr*	→	*w sw*
1 6	*r sr*	→	*wsr wsr wsr wsr*	→	*w sw*
2 1	*r*	→	*s s*	→	*w*
2 2	*r sr*	→	*s*	→	*w sw*
2 3	*r sr sr*		→		*w sw sw*
2 4	*r sr sr*	→	*wsr*	→	*w sw sw*
2 5	*r sr sr*	→	*wsr wsr*	→	*w sw sw*
2 6	*r sr sr*	→	*wsr wsr wsr*	→	*w sw sw*
2 7	*r sr sr*	→	*wsr wsr wsr wsr*	→	*w sw sw*
3 1	*r*	→	*s s s*	→	*w*
3 2	*r sr*	→	*s s*	→	*w sw*
3 3	*r sr sr*	→	*s*	→	*w sw sw*
3 4	*r sr sr sr*		→		*w sw sw sw*
3 5	*r sr sr sr*	→	*wsr*	→	*w sw sw sw*
3 6	*r sr sr sr*	→	*wsr wsr*	→	*w sw sw sw*
3 7	*r sr sr sr*	→	*wsr wsr wsr*	→	*w sw sw sw*
3 8	*r sr sr sr*	→	*wsr wsr wsr wsr*	→	*w sw sw sw*
4 1	*r*	→	*s s s s*	→	*w*
4 2	*r sr*	→	*s s s*	→	*w sw*
4 3	*r sr sr*	→	*s s*	→	*w sw sw*
4 4	*r sr sr sr*	→	*s*	→	*w sw sw sw*
4 5	*r sr sr sr sr*		→		*w sw sw sw sw*
4 6	*r sr sr sr sr*	→	*wsr*	→	*w sw sw sw sw*
4 7	*r sr sr sr sr*	→	*wsr wsr*	→	*w sw sw sw sw*
4 8	*r sr sr sr sr*	→	*wsr wsr wsr*	→	*w sw sw sw sw*
4 9	*r sr sr sr sr*	→	*wsr wsr wsr wsr*	→	*w sw sw sw sw*

Table 2. *The pattern of communications for send*

in the system participate simultaneously in the shift, which sends data between the four *corners* of the square. For example, at the 2×2 square size in Figure 2, a shift right operation (moving the contents of cell i to the neighbor input field of cell $i+1$) causes the contents of cell 40 to move north to 41, while cell 41 moves west to 42, 42 moves south to 43, and 43 moves east to 40. All other 2×2 squares perform this pattern of moves at the same time.

In general, the four data movements (north, west, south and east) will include three legitimate movements and an invalid one. Thus it was invalid to move the contents of cell 43 east to 40. However, the particular direction of movement that is invalid depends on the parity of the square's central node. Since central nodes appear with each of the four parities NE, NW, SE, SW, each of the four data movements will be invalid for 1/4 of the squares.

Fortunately, the invalid fields left by data movement at one square level are all overwritten when the shift algorithm does the next higher level. After doing the 2×2 squares, the corner cells of all the 4×4 cells send their data to their neigbors (within the square). For the 4×4 square in the northwest corner of Figure 2, the four corner source cells are 35, 39, 43 and 47 — so the west move writes the contents of cell 39 into cell 40, replacing its old invalid contents left by the 2×2 moves. Of course, 1/4 of the moves at this level are also invalid, but when the shift algorithm reaches the highest square level (128×128), the only cell that remains with an invalid neighbor input field is the leftmost cell in the entire memory: cell 0 holds the contents of cell 255. If the shift algorithm was called to rotate the memory, this is the desired effect. Otherwise, after handling the highest level square, the shift algorithm writes a value into cell 0 that it receives from the memory controller.

THE COMMUNICATION ALGORITHMS

The sweep and shift algorithms must move k bits (called the *data path width*) from a set of *source PEs* to the corresponding *destination PEs*. The preceding section outlined how shift and sweep locate the sources and destinations and calculate the path distances d for each move. It is also important to consider the details of the data movement algorithm, *send*, because this is the innermost loop of the APSA emulator, accounting for much of the total execution time.

One factor simplifies the implementation of send: all of the movements that it must perform at once go in a straight line for the same distance d. However, two other factors complicate it:

- Simultaneous movements may go in different directions. In particular, sweep always needs to send data east and west, or north and south. The shift algorithm always sends data all four directions.

- The two key parameters of a send — the distance d and the number of bits k — are variables computed by the calling algorithm. Therefore the send code must be flexible enough to handle automatically any combination of d and k.

Sending a bit from the source to the destination involves three basic operations:

r *read* the bit into the source PE;

- *s* *shift* each bit on the data path one PE closer to the destination; and
- *w* *write* the bit into the destination PE.

Since all the data paths have the same length and width, and none of them cross each other, we can concentrate on what happens to the data during a single movement.

Any send operation consists of a pattern of the *r*, *s* and *w* primitives. For example, *rsssw* will send a single bit (k=1) along a path 3 PEs long (d=3). The situation is more complex when $k > 1$. In order to minimize the communication time, it is essential to pipeline data along the path. This leads to a number of distinct patterns of communication, illustrated by Table 2. The send algorithm must use the values of *d* and *k* to determine the best communication pattern.

All the sources, destinations and data paths necessary for sweep and shift must be marked in advance by an initialization algorithm. The initializer's main function is to compute a set of *mask lists*, one for every node level in the tree. Every processing element has its own set of mask lists, and each one contains 14 individual masks that indicate whether the PE contains a node at each level in the tree, what the nodes' parities are, and what data paths pass through the PE. The mask lists allow the *r*, *s* and *w* primitives to operate in the relevant PEs without disturbing data in the others. For example, the *r* primitive places a "node present" mask in the G register, and then executes a `loadm P,SOURCE` instruction.

ORGANIZATION OF THE EMULATOR

The APSA system breaks naturally into three components, and the MPP implementation reflects this structure by running the components on separate computers:

- The memory instruction set, which provides the heart of APSA's parallelism and which requires simulation of a VLSI architecture layout, runs entirely in the Processing Element Control Unit (PECU). This software, written in Pearl, includes the initializer, shift and sweep algorithms, as well as the node and cell logic functions for each instruction. The speed of these programs is of paramount importance for the overall goals of the research.
- The memory controller and applicative language interpreter, written in MCL, reside in the Main Control Unit (MCU), and issue instructions to the APSA memory through the call queue.
- The I/O system, written in Fortran, runs on the MPP host.

PERFORMANCE RESULTS

This section discusses the performance of the emulator's initializer and sweep algorithm. The shift algorithm is similar to sweep, but it will run considerably slower because its data path width will usually be around 64 bits, while typical sweeps only require 3 or 4 bits.

The MPP execution times were measured using the performance monitor, which is extremely accurate and repeatable. In addition, the MPP simulator predicted the same performance times for the smaller APSA layouts.

All of the emulation algorithms (initialize, sweep and shift) take a parameter that specifies the size of the APSA layout — they don't depend on the 128×128 size of the MPP array unit. In general, the layout may be $n \times n$ for any n that is a power of 2. Consequently, the height of the binary tree may be any even number.

Table 3 shows the time required to initialize APSA for all the layouts that fit in the MPP array. While growing a layout the initializer must perform one complete shift and five complete downsweeps, in addition to performing several other functions. The *Layout* column shows the dimensions of each APSA layout. The *Level* column gives the height of the binary tree for each layout; thus a layout with *Level*=n will contain 2^n *Cells*. The *Square* column shows the depth of recursion in the H-tree layout, which is also the number of iterations in the outer loops of the sweep and shift algorithms.

Square	*Level*	*Layout*	*Cells*	*Time (μs)*
1	2	2×2	4	270
2	4	4×4	16	429
3	6	8×8	64	596
4	8	16×16	256	784
5	10	32×32	1,024	1,019
6	12	64×64	4,096	1,339
7	14	128×128	16,384	1,817

Table 3. APSA initialization times in μs (MPP)

Table 4 gives the MPP execution times of the upsweep algorithm for data paths that are 1, 2 and 3 bits wide. (The downsweep algorithm is almost identical.) There are three phenomena that account for the variations among these times. First, the number of iterations of the main sweep loop and the number of calls to the node logic function both depend on the layout size (the *Square* column in Table 3 gives these values). Second, the data paths become longer toward the top of the tree in larger layouts — that is why the execution time grows faster than linearly as a function of the *Square* size. Third, the width of the data path (i.e.,

Layout	*1 Bit*	*2 Bits*	*3 Bits*	*Increase*
2×2	12	17	23	
4×4	20	31	42	+19
8×8	29	45	62	+20
16×16	39	62	85	+23
32×32	51	80	110	+25
64×64	68	104	140	+30
128×128	95	137	179	+39

Table 4. Upsweep times in μs (MPP)

Layout	*MPP*	*Vax 780*	*Speedup*
2×2	23	610	27
4×4	42	2,500	60
8×8	62	10,500	169
16×16	85	53,000	624
32×32	110	175,000	1,591
64×64	140	706,000	5,043
128×128	179	2,872,000	16,045

Table 5. Comparison of upsweep times in μs

the number of bits being sent up the tree) determines the number of iterations of sweep's inner loop, and the node logic function also requires more time to operate on more bits.

At this point it is interesting to consider the implications of Table 4 on the future course of the APSA research project. Serious development of parallel Lisp and SASL will require a moderately large memory. That is why the emulator supports a skewed H-tree instead of the conventional H-tree: it can place a memory cell in every MPP processing element, providing the largest memory size that is possible without loss in parallelism. Therefore we are primarily interested in the last line of Table 4, which gives sweep times for a 16,384 cell memory. Assuming two sweeps and several parallel operations in all the cells, a typical APSA instruction should take about half a millisecond. This is certainly adequate for extensive experimentation and development. There is also the possibility that some applicative algorithms may by able to perform the equivalent of several hundred operations simultaneously. If that happens, the MPP/APSA system could partially achieve a longstanding goal of the programming language research community: automatically speeding up a list processing program on a parallel machine, *without requiring the programmer to specify any parallel operations* (Refs. 11, 12). Of course, this goal is still far in the future.

Table 5 illustrates what the future of APSA would be like without the MPP: it repeats the upsweep times for a 3 bit data path and compares them with an APSA upsweep algorithm running on a Vax 780 computer. The Vax program is written in C, and it exploits some of the standard techniques for efficiency in C (for example, it increments pointers into arrays in order to avoid most array index calculations). These timings were generated using the "time" command in Berkeley Unix 4.2BSD, and there is an error range of about ±10%. The columns headed *MPP* and *Vax 780* give the running time in microseconds of a 3 bit upsweep on each layout size. The *Speedup* column gives the ratio of *Vax 780* time divided by *MPP* time. It is important to realize that these figures do *not* show how much faster the MPP is than a Vax. They merely indicate how much the performance of upsweep can be improved by moving from a Vax to the MPP.

For the case of most interest — a 3-bit upsweep on a full 128×128 layout — the MPP requires 179 μs, while the Vax takes 2.87 seconds. At this rate, one MPP-minute of upsweeps would correspond to eleven Vax-days of CPU time. Thus the MPP makes it easy to run emulations that would be inconceivable using conventional computers.

DISCUSSION

The speedups in Table 5 may appear surprising, because the MPP is nominally only on the order of 512 times faster than a Vax (it has 16,384 times more processors and its word size is 1 bit, compared with 32 on the Vax). But the actual performance speedup depends in detail on the interaction between the algorithm and each computer's architecture. In particular, the upsweep algorithm is inherently bit serial, so the large Vax word length does not help at all. The shift algorithm uses long words, so the MPP's speedup will be much smaller than for sweep.

It would be extremely valuable to investigate the performance of APSA on a Connection Machine (Ref. 2) and compare it with the results given above. Since the Connection Machine contains a network with long data paths, it will be faster than the MPP for sufficiently large layouts. On the other hand, the MPP's faster cycle time should make it faster for smaller layouts. Table 4 shows that much of the time for an upsweep for a 128×128 layout goes into bit-serial operations and the node logic functions, where the MPP is faster. The longest data paths have length 32; the MPP is probably still faster than the Connection Machine at this size because of its synchronous communications and fast clock.

There are several other approaches to parallel implementation of Lisp and related languages. Multilisp (Ref. 1) gives the programmer a parallel construct called a *future*, which allows parallel evaluation of independent expressions. Multilisp is being used to program an MIMD parallel computer. Connection Machine LISP (Ref. 10) takes an approach similar to APSA: it makes parallel data structures available to the user programmer. However, it does not integrate parallel data structures as deeply into the interpreter's environment and continuations as APSA does. Another method for implementing car and cdr on the MPP (Ref. Potter) allows parallel searching of many lists, but its cons function requires time and space proportional to the length of the second argument.

CONCLUSION

The Applicative Programming System Architecture research is concerned with designing an applicative language, a computer architecture and a VLSI hardware implementation together, so that they cooperate effectively. This research is leading toward better language implementations and new architecture designs. However, simulating the low level hardware on a conventional computer is too slow to allow experimentation with APSA's parallel algorithms.

The Massively Parallel Processor is ideal for simulating VLSI circuits that have regular designs and short data paths, such as systolic arrays. The VLSI layout for APSA's memory contains parallel logic in every memory cell and a binary tree for communications and additional logic. The tree layout contains only a few long paths, so it is well suited for implementation on the MPP. The key tree communication algorithm shows a huge speedup — by a factor of 16,000 — compared with simulation on a conventional computer.

The basic APSA operations are fast enough on the MPP to allow experimentation with a realistic parallel implementation of an applicative programming language, and a parallel implementation of Lisp is in progress. Thus the MPP is making it possible to study new ideas in parallel VLSI architectures.

ACKNOWLEDGEMENT

I would like to thank Judy Devaney for helping me with the I/O macros in MCL, and especially for maintaining the simulator, which is extremely useful for developing large Pearl programs.

REFERENCES

1. Robert H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation", *Trans. on Prog. Lang. and Systems*, pp. 501–538, Vol. 7, No. 4, Oct. 1985.
2. W. Daniel Hillis, *The Connection Machine*, Cambridge: The MIT Press, 1985.
3. Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.
4. John T. O'Donnell, *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, University of Iowa, Iowa City, 1981.
5. John T. O'Donnell, "An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages", *1985 IFIP Symposium on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985, *Lecture Notes in Computer Science 201*, pp. 164–189, New York: Springer-Verlag.
6. John T. O'Donnell, "An efficient architecture for implementing sparse array variables," *Proceedings of the Twenty-third Allerton Conference on Communication, Control and Computing*, pp. 986–995, Coordinated Science Laboratory, University of Illinois, October, 1985.
7. John T. O'Donnell, "Finely Grained Parallelism in an Applicative Architecture", *Proceedings of the Workshop on Future Directions in Computer Architecture and Software*, pp. 372–374, Army Research Office, May, 1986.
8. John T. O'Donnell, "Simulating VLSI Systems Using the Massively Parallel Processor", *Proceedings of the 1986 Summer Computer Simulation Conference*, pp. 153–158, The Society for Computer Simulation, July, 1986.
9. Jerry L. Potter, "List Based Processing on the MPP", in *The Massively Parallel Processor*, (ed. J. Potter), Cambridge: The MIT Press, 1985.
10. Guy L. Steele, Jr., W. Daniel Hillis, "Connection Machine LISP: Fine-Grained Parallel Symbolic Processing", pp. 279–297, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming.*
11. Philip C. Treleaven, "Computer Architecture for Functional Programming", pp. 281–306 in *Functional Programming and its Applications* (ed. J Darlington), Cambridge University Press, 1982.
12. Steven R. Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984.