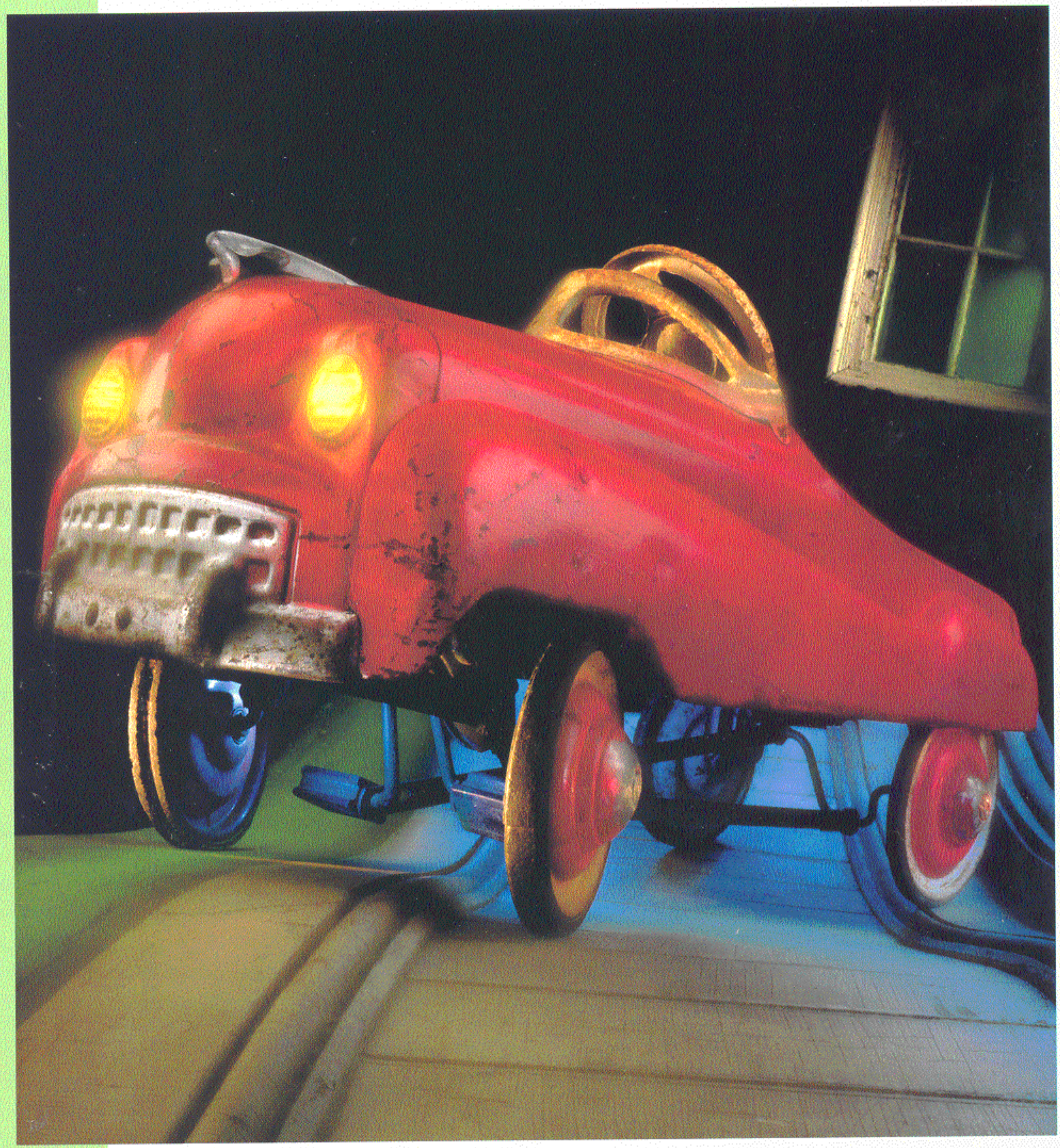
NOVEMBER 1994

COMPUTER

Innovative technology for computer professionals



Associative Processing

Remembrance of things past

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

ASC: An Associative-Computing Paradigm

Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal,
Chokchai Leangsuksun, and Chandra Asthagiri

Kent State University

Today's increased computing speeds allow conventional sequential machines to effectively emulate associative computing techniques. Here is a parallel programming paradigm designed for a wide range of computing engines.

Associative computing evolved in an era when associative memories were both relatively new and, because they required a comparator at each bit of memory, relatively expensive. In the early 1970s, Goodyear Aerospace improved upon early associative processing techniques with its Staran SIMD (single instruction, multiple data) computer.[1] Goodyear realized that the massively parallel search capability of bit-serial SIMDs could simulate associative searching, with the cost advantage of sharing the comparison logic (that is, the processing elements) over all the bits in an entire row of memory. This approach provided two additional benefits: The word widths could be very large (from 256 bits to 64 kilobits), and the data could be processed in situ using the same PEs.

However, today's lower hardware costs and increased computing speeds allow parallel techniques to be effectively emulated on conventional sequential machines. Accessing data by associative searching rather than addresses and processing data in memory require a new programming style. One goal of our research is to develop a parallel programming paradigm that is suitable for many diverse applications, is efficient to write and execute, and can be used on a wide range of computing engines, from PCs and workstations to massively parallel supercomputers.

Our associative-computing (ASC) paradigm is an extension of the general associative processing techniques developed by Goodyear. We use two-dimensional tables as the basic data structure. Our paradigm has an efficient associative-based, dynamic memory-allocation mechanism that does not use pointers. It incorporates data parallelism at the base level, so that programmers do not have to specify low-level sequential tasks such as sorting, looping, and parallelization.

Our paradigm supports all of the standard data-parallel and massively parallel computing algorithms. It combines numerical computation (such as convolution, matrix multiplication, and graphics) with nonnumerical computing (such as compilation, graph algorithms, rule-based systems, and language interpreters).[2] This article focuses on the nonnumerical aspects of ASC.

The ASC model

The ASC model is the basis of a high-level associative-programming paradigm and language. As described in the sidebar, "Properties of the ASC model," the extended model provides a basis for algorithm development and analysis similar to the

0018-9162/94/$4.00 © 1994 IEEE

PRAM (parallel random-access memory) models, with the additional provisions that hardware can be built to support this model and that its primitive operations are sufficiently rich to allow efficient use of massive parallelism.[3] These features let us develop parallel algorithms for large problems that can be abstractly analyzed and executed. Furthermore, algorithms based on a common model will have greater applicability and retain their importance longer than ones based on a specific computer that may be out of production within a few years. Briefly, the model calls for data-parallel execution of instructions, constant-time associative searching, constant-time maximum and minimum operations, and synchronization of instruction streams using control parallelism. The simplest ASC model assumes only one instruction stream (IS). This model can be supported on existing SIMD computers and is assumed throughout unless we state otherwise.

The sidebar lists the specific properties that the hardware must have to support the model. Reflecting these specifications, the ASC language is characterized by built-in associative reduction notation, associative responder iteration, responder-based flow of control, responder reference and selection mechanisms, and a multiple instruction stream capability that provides dynamic control parallelism on top of data parallelism. ASC supports recursion and special command constructs with automatic backtracking for complex context-sensitive searching. Fundamental to the nonnumerical focus of ASC are the unique structure code features and dynamic memory allocation. The most important features of ASC are discussed below. (ASC language syntax is described in detail in Potter.[2])

Associative programming techniques

Generally, a few basic techniques determine the "feel" of a programming paradigm, such as pointers in C and tail recursion or list processing in Lisp and Prolog. In ASC, the associative search is the fundamental operation, and its influence is felt in constant-time operations, tabular representation of abstract data structures, responder processing, and control parallelism.

Constant time operation. Data parallelism is a basic model used in many languages. ASC uses data parallelism as the basis for associative searching, which takes time proportional to the number of bits in a field, not the number of data items being searched. Thus, assuming

Properties of the ASC model

We have applied the ASC paradigm to a wide range of applications, including image processing (for example, convolution[1]), graph algorithms (for example, the minimal spanning tree), rule-based inference engines (for example, OPS5[2]), Associative Prolog,[1] graphics (ray tracing[3]), database management,[4] compilation (first pass[5] and optimization[6]), and heterogeneous networks.[7, 8]

Our intention is that ASC be efficiently supported in hardware by a continuum of compute engines. The first step in this continuum has been to install the ASC language on conventional sequential computers such as PCs and workstations. Second, associative functions and operations can be sped up by using accelerator cards similar to the one currently being developed for ARPA[9] by Adaptive Solutions Inc. Conventional SIMD computers provide the third level of associative functionality. (ASC has been installed on Staran, Aspro, Wavetracer, and the CM-2.) The highest, most complex, and fastest level would be a multiple instruction stream SIMD computer built to meet the specifications of the following computation model:[10]

Cell properties

- Cells consist of a processing element (PE) and a local memory (see Figure A).
- The memory of an associative computer consists of an array of cells.
- There is no shared memory between cells. Each PE can only access the memory in its own cell. The cell's PE can be interconnected with a modest network (for example, a grid).
- Related data items are grouped together (associated) into records and typically stored one per cell. We assume that there are more cells than data.

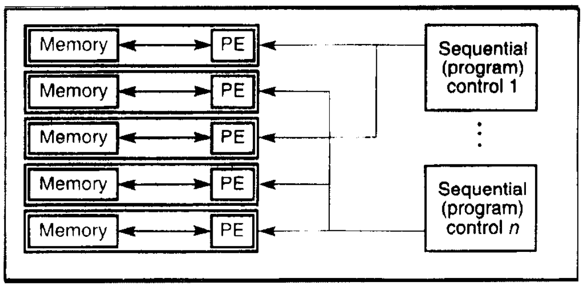

Memory
PE
Memory
PE
Memory
PE
Memory
PE
Memory
PE
Sequential (program) control 1
Sequential (program) control n

Figure A. Cellular memory.

Instruction stream (IS) properties

- Each IS is a processor with a bus to all cells. The IS processors are interconnected (for example, by a bus, network, or shared memory). Each IS has a copy of the program being executed and can broadcast an instruction to all cells in unit time. The parallel execution of a command is SIMD in nature.
- Each cell listens to only one IS. Initially, all cells listen to the same IS. The cells can switch to another IS in response to commands from the current IS.
- The number of cells is much larger than the number of IS's.
- An active cell executes the commands it receives from its IS, while an inactive cell listens to but does not execute the commands from its IS. Each IS has the ability to unconditionally activate all cells listening to it.

Associative properties

- An IS can instruct its active cells to perform an associative search. Successful cells are called responders, while unsuccessful cells are called nonresponders. The IS can activate either the set of responders or the set of nonrethat all the data fits in the computer, it executes in constant time,[2] just as comparison, addition, and other data-parallel arithmetic operations do. In addition to basic pattern searching, ASC makes extensive use of constant time functions[4] (maximum, minimum, greatest lower bound, and least upper bound). The constant time functions have corresponding constant-time associative index functions (maxdex, mindex, prvdex, and nxtdex), which are used for associative reduction. For example, the query "What is the salary of the oldest employee?" requires a maximum search on the age field, but the associated salary, not the age, is the desired item. The maxdex function in "salary[maxdex(age$)]" expresses the association between the maximum age and the associated salary. Computers with the properties specified in the sidebar can execute these functions in constant time. In addition, today's sequential computers are powerful enough to emulate these operations for many problems.

Tabular data structures and structure codes. Tabular data structures (that is, tables, charts, and arrays) have two advantages for ASC. First, they are ubiquitous; tables and arrays are a common and natural organization for databases and many scientific applications, and users need only a minimal introduction to manipulate them effectively. Second, the concept of processing an entire column of a table simultaneously is easy to comprehend.

There are a number of common abstract data structures, including stacks, queues, trees, and graphs, that are normally implemented using address manipulation via pointers and indexes. In an associative computer, in contrast, physical address relationships between data are not present. Instead, structure codes, which are numeric representations of the abstract structural information, are associated with the data. The codes are generated automatically, and appropriately named functions — for example, parent(), sibling(), and child() — are used to manipulate them. The programmer need be aware only of the data structure type being used (tree, graph, and so forth) and not the internal structure codes themselves.

One of the major advantages of structure codes is that they allow the data to be expressed in tabular form so that they can be processed in a data-parallel manner. This means that lists, trees, and graphs can be searched associatively in constant time instead of having to be sequentially searched element by element. Tabular organizations are stored one row per cell in an associative computer. Thus, any one field (a column of the table) can be searched in parallel by broadcasting the desired value to all cell PEs, which then compare it with their local values.

sponders. It can also restore the previous set of active cells. Each of these actions requires one unit of time.

- Each IS has the ability to select an arbitrary responder from the set of active cells in unit time.
- Each IS can instruct the selected cell to broadcast data on the bus. All other cells listening to this IS receive the value placed on the bus in unit time.

Constant time global operations

- An IS can compute the OR or AND of a binary value in all active PEs in unit time.
- An IS can identify the cells with the maximum or minimum value in each of its active PEs in constant time.

Control parallelism

- Cells without further work to do are called idle cells and are assigned to a specified IS, which (among other tasks) manages the idle cells. An idle cell can be dynamically allocated to an IS in unit time. Any subset of cells can be deallocated and reassigned as idle cells in constant time.
- If an IS is executing a task that requires two or more subtasks involving data in disjoint subsets of the active cells, control (MIMD) parallelism can be invoked by assigning a subtask to an idle IS. When all subtasks generated by the original IS are completed, the cells are returned to the originating IS.

A new programming paradigm called Heterogeneous Associative Computing[7] (HASC) is presently under development at Kent State University. From the ASC model, this paradigm takes the concept of cells and instruction broadcasting. It uses tabular data and massively parallel searches to match commands and data to machines. The result is that, in an extension of polymorphism, commands are executed on the machines best suited for them.

References

1. J.L. Potter, *Associative Computing — A Programming Paradigm for Massively Parallel Computers*, Plenum Publishing, N.Y., 1992.
2. J.W. Baker and A. Miller, "A Parallel Production System Extending OPS5," *Proc. Frontiers of Massively Parallel Computation*, CS Press, Los Alamitos, Calif., Order No. 2772-02, 1990, pp. 110-118.
3. T. Krochta, *Parallel Ray Tracing*, master's thesis, Kent State Univ., Kent, Ohio, 1986.
4. K. Mamoozadeh, *Relational Databases on Associative Processors*, master's thesis, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, 1986.
5. C. Asthagiri, *Context-Sensitive Parsing Using an Associative Processor*, master's thesis, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, 1986.
6. R. Miles, *Optimizing Associative Intermediate Code*, master's thesis, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, 1993.
7. S.L. Scott and J.L. Potter, "Heterogeneous Associative Computing — HASC," *2nd Associative Processing and Applications Workshop*, Syracuse Univ., Syracuse, N.Y., July 1993; Tech. Report CS-9305-05, Dept. of Mathematics and Computer Science, Kent State University, Kent, Ohio, May 1993.
8. C. Leangsuksun, S.L. Scott, and J.L. Potter, "Implicit Task Mapping in a Heterogeneous Environment," Tech Report CS-9409-08, Dept. of Mathematics and Computer Science, Kent State University, Kent, Ohio, May 1993.
9. *Electronic Eng. Times*, Feb. 7, 1994, p. 41.
10. J.W. Baker and J.L. Potter, "A Model of Computation for Associative Computing," Tech. Report CS-9409-07, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, Sept. 1994.

Additional information is available via WWW at http://nimitz.mcs. kent.edu/ {~potter, ~chokchai, ~sscott, ~arvind, ~jbaker}.

row	value
1	95
2	17
3	36
4	47

(a)

row	column	value
1	1	92
1	2	89
2	1	63
2	2	52

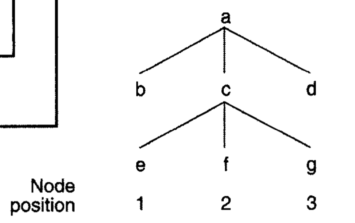
(b)

Figure 1. Example structure codes for vector (a) and for a two-dimensional array or matrix (b).

time	value
6	35
2	15
4	20
3	10
1	100

```
FIFO VALUE = VALUE [MINDEX (TIME$)]
LIFO VALUE = [MAXDEX (TIME$)];
```

Figure 2. Associative LIFO and FIFO queues using maximum and minimum function to retrieve associated values.



Code	Node
1000	a
1100	b
1200	c
1300	d
1210	e
1220	f
1230	g

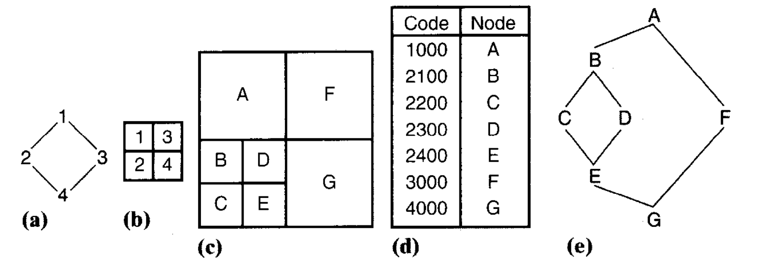
Figure 3. Canonical tree structure and structure codes as represented in associative memory.

The simplest structure codes are those for arrays. For nonnumerical applications, a vector can be represented by a row field and a value field as shown in Figure 1a. Likewise, we can represent a two-dimensional array or matrix by a row field, a column field, and a value field. In Figure 1b, the matrix value at position (1, 2) can be found in constant time by searching for row 1 and column 2 and retrieving the associated value — 89.

Frequently, we can represent directly useful information in the structure code. For example, the time of arrival is used to implement FIFO and LIFO queues. For example, in Figure 2 the FIFO value in the queue is retrieved using the mindex function to select the first (smallest or oldest) time entry and its associated value.

Trees and graphs require more sophisticated structure codes. If trees are put into a canonical form, and the position of the nodes on every level are numbered from left to right, we can generate a code for every node in the tree by starting at the root of the tree and listing the node numbers along the path to the node in question. If the code is left justified with zero fill, it will support parallel searching, concatenation, insertion, and deletion. Figure 3 gives an example of a tree and its structure codes as represented in an associative memory. The left and right siblings of node *f* can be found in constant time by using the sibdex function — sibdex(code[node$=='f']).

This expression can be read from the inside out. First, the node field is searched for the value *f*; the response is used to select the associated structure-code value (1220), which is passed to the sibdex function. Sibdex combines the greatest-lower-bound and least-upper-bound search functions to identify codes 1210 and 1230 as being adjacent to 1220, and their associated nodes — *e* and *g* — as siblings of *f*. All operations are constant time. This kind of operation is very useful for expression parsing.[5]



Code	Node
1000	A
2100	B
2200	C
2300	D
2400	E
3000	F
4000	G

Figure 4. Quadsected square encoding for (a) binary graph, (b) quadsected square, (c) recursively quadsected square, (d) structure codes, and (e) binary graph.

Quadsected square codes are structure codes for graphs that can be applied to the generation of node domination, node influencing, and similar information useful in control flow graph analysis. A quadsected square is a square divided into four subsquares. The quadrants of a quadsected square can be recursively subdivided to any level. The quadsected square code calculation and manipulation functions are performed in data-parallel mode for all nodes of a graph. For example, given the code for a node, the dominance relationship between the node and all other nodes in the graph can be computed in constant time independent of the size of the graph.

Figures 4a and 4b illustrate the dual relationship between binary graphs and quadsected squares. The graph's binary fan-out (node 1 branches to 2 and 3) and binary fan-in (node 2 and 3 converge on 4) shown in Figure 4a are mapped onto the location code map shown in Figure 4b. A more complex example is given in Figures 4c, 4d, and 4e, where the control flow starts in quadrant A and flows into the upper left-most subdivision of the two adjacent quadrants (B of BCDE and F). Each subdivision continues this recursive process until the final two quadrants within a subdivision are joined at their right-most subdivision (C and D are joined at E, and E and F are joined at G).

We obtain the structure code for a recursively quadsected square (Figure 4d) by specifying the position of the top-level subdivision first (as the left-most digit), then the position of the next recursive subdivision, and so on, with zero fill used on the right.

Responder processing. The responders of an associative search are those cells that successfully matched the associative search query. Data-parallel operations applied to the responders essentially act as substitutes for the index-based loops used in Fortran and C. However, it is sometimes desirable to process each responder individually. In responder iteration, a responder is arbitrarily selected and processed using both sequential and parallel operations. When processing is complete, the responder is idled and another responder is selected for processing. Responder iteration is an effective way of using parallel searching to avoid sorting unordered data.

We use responder selection to achieve constant-time memory allocation. Idle cells are assigned to a single instruction stream. When an IS needs one or more new cells, they are arbitrarily selected from the idle pool and allocated to the requesting IS. When that IS no longer needs those cells, they are identified by associative search, released in parallel, and returned to the idle IS. Figure 5 illustrates the difference between associative-memory allocation and C-based data-parallel memory allocation, where additional fields, not cells, are allocated to the active processors. (The "loop while" statement in Figure 6 is an example of responder iteration.)

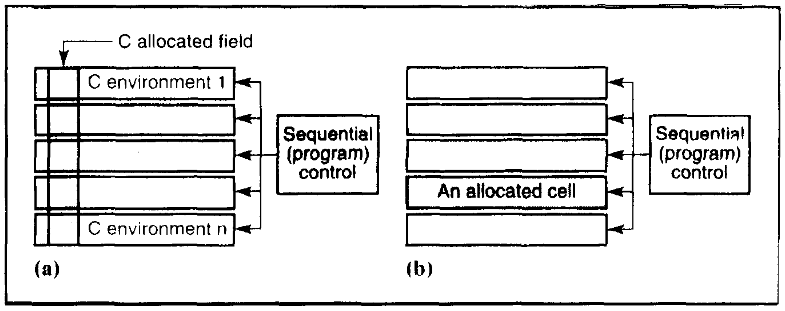
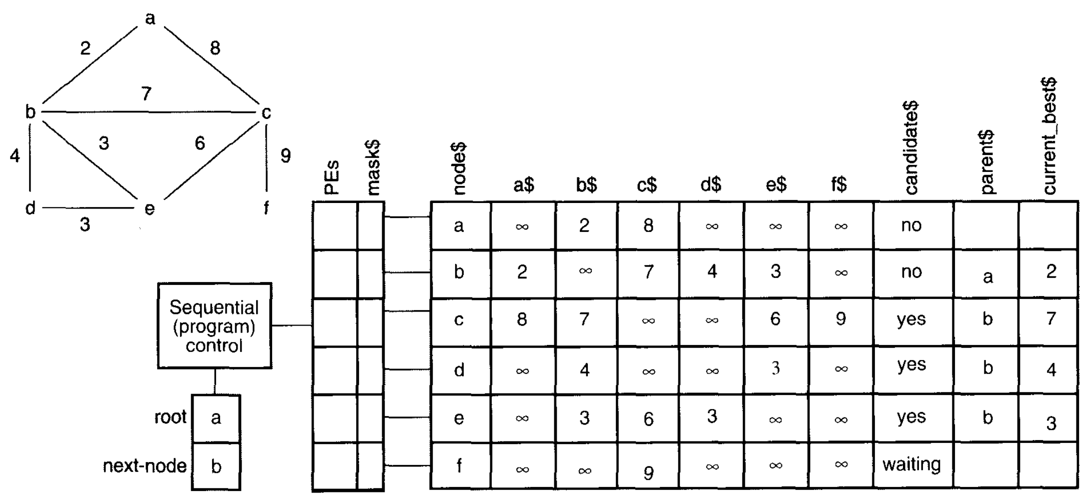


Figure 5. Dynamic memory allocation for (a) C-based environment and (b) the associative computing model.

Control parallelism. To this point our discussion has centered on data paral-



node$	a$	b$	c$	d$	e$	f$	candidate$	parent$	current_best$
a	∞	2	8	∞	∞	∞	no		
b	2	∞	7	4	3	∞	no	a	2
c	8	7	∞	∞	6	9	yes	b	7
d	∞	4	∞	∞	3	∞	yes	b	4
e	∞	3	6	3	∞	∞	yes	b	3
f	∞	∞	9	∞	∞	∞	waiting		

```
ASC-MST-PRIM (root)
initialize candidates to "waiting"
if there are any finite values in root's field, then
    set candidate$ to "yes"
    set parent$ to root
    set current_best$ to the values in root's field
    set root's candidate field to "no"
loop while some candidate$ contain "yes"
    for them
        restrict mask$ to mindex(current_best$)
        set next_node to a node identified in the preceding step
        set its candidate to "no"
        if the values in next_node's field are less than
        current_best$, then
            set current_best$ to value in next_node's field
            set parent$ to next_node
    if candidate$ is "waiting" and the value in next_node's
    field is finite
        set candidate$ to "yes"
        set parent$ to next_node
        set current_best$ to next_node
        set current_best$ to the values in next_node's field
```

Figure 6. An associative minimal spanning tree algorithm.

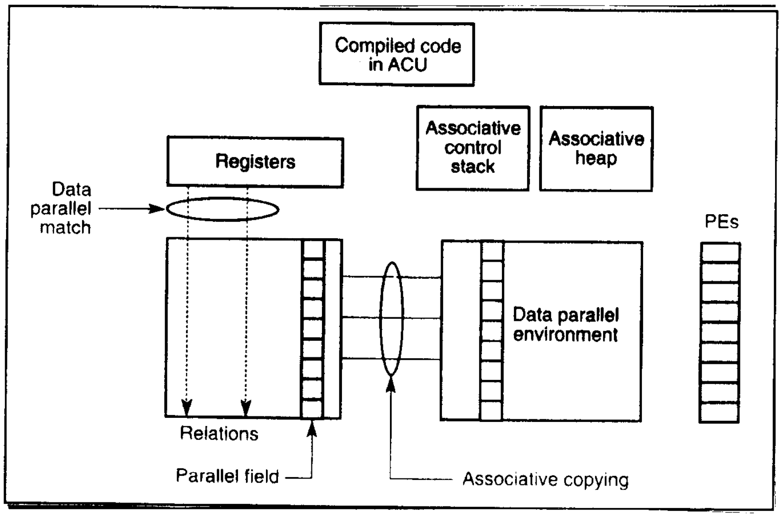


Figure 7. Associative logic programming.

lelism. However, the ASC model accommodates both data and control parallelism so that the computer can efficiently use all its cells. The control-parallel component depends on the dynamic manipulation of instruction streams in response to associative searches. The mechanism relies on partitioning the responders into mutually exclusive subsets. For example, the evaluation of an IF's conditional expression divides the active cells into two mutually exclusive partitions: one containing the cells that respond TRUE and one containing the cells that respond FALSE. These partitions can be processed using control parallelism by forking the process: One IS is assigned to execute the THEN portion of the IF statement with the TRUE responders, and another IS is assigned to execute the ELSE portion with the FALSE responders. The IS's execute in parallel, each in a data-parallel mode. The programmer needs no control-parallel statements, such as FORK or JOIN, since the control parallelism is inherent in the statements. Case statements are another example of control parallelism, except that there are n partitions — one for each of the n cases — instead of two partitions as in the IF-THEN-ELSE.

A significant speedup of up to k in the runtime of certain algorithms is possible using an associative computer with a constant number k of instruction streams. Moreover, if the number of instruction streams is not restricted to being constant, then new algorithms with lower complexity times may be possible.

Example applications

An ASC version of Prim's minimal spanning tree (MST) algorithm[3] using associative-computing techniques with only one IS is given in Figure 6. The values given there indicate the state of the algorithm after the first iteration through the "loop while." All the statements in the algorithm execute in constant time. The data for each node is stored in a record, and the records are stored with at most one record per cell. The cell variables are identified with a "$" symbol following the variable name. The cost of an edge from node x to node y is stored in the x\$ field of node y and the y\$ field of node x. Each record also has the additional variables, candidate$, parent$, and current_best$. Root and next-node are scalar variables. If root = n, then the terminology "root's field" refers to the field n\$. Since one tree edge is selected by each pass through the loop and a spanning tree has $n - 1$ tree edges, the runtime of this algorithm is $O(n)$. This is a cost-optimal parallel implementation of Prim's original MST algorithm, which has a sequential running time of $O(n^2)$. Moreover, no additional overhead is incurred as the size of the graph increases, because the algorithm only requires additional cells and is thus easily scalable to larger data sets. Finally, since no networks are used and no task-forking or join operations are needed, we have minimized the communications and synchronization overhead costs.

ASC has been combined with logic programming to achieve high-performance intelligent reasoning, data-parallel scientific computing, and efficient information retrieval from large knowledge bases.[6] The strategy in the design of the associative-logic programming system is to maximize the use of bit-vector and data-parallel operations and to minimize the movement of scalar data. Facts, relations, and the left-hand sides of rules are represented as records (associations) of parallel fields with one record per cell. The right-hand sides of the rules are compiled into an abstract instruction set. A simplified schematic of the model is given in Figure 7.

Some advantages of combining associative and logic computing are

(1) the speed of knowledge retrieval is independent of the number of ground facts,
(2) knowledge retrieval is possible even if the information is incomplete, making knowledge discovery possible,
(3) relations with a large number of arguments are handled efficiently with little overhead,
(4) associative lookup is fast, allowing the tight integration of high-performance knowledge retrieval and data-parallel computation without any overhead due to data movement or data transformation, and
(5) the model is efficient for both scalar and data-parallel computations on various abstract data types such as sequences, matrices, bags, and sets.

These advantages suggest that this paradigm can be successfully applied to data-intensive problems such as geographical information systems, image-understanding systems, statistical knowledge bases, and genome sequencing. For example, in geographical information systems, spatial data structures such as quadtrees and octtrees are represented associatively with structure codes. As a result, different regions having the same values can be identified using associative searches in constant time.

The integration of data-parallel scientific computing, knowledge base retrieval, and rule-based reasoning provides necessary tools for image-understanding systems. Statistical queries can directly benefit from associative searches, associative representation of structures, data-parallel arithmetic computations, and data-parallel aggregate functions. Genome sequencing requires integration of knowledge retrieval, efficient insertion and deletion of data elements, and efficient manipulation of matrices for the heuristic matching of sequences.

The associative techniques of the 1970s augmented with new techniques — such as structure codes, dynamic memory allocation, responder iteration, multiple instruction streams, associative selection, and reduction notation and pronouns — form the basis of a programming paradigm that makes use of today's inexpensive computing power to facilitate parallel programming. The ASC paradigm uses a tabular-data organization, massive parallel searching, and simple syntax, so that the paradigm is easily comprehensible to computer specialists and nonspecialists alike. Furthermore, the ASC paradigm is suitable for all levels of computing, from PCs and workstations to multiple instruction stream SIMDs and heterogeneous networks. ■

Acknowledgments

The authors thank Selim Akl for his helpful comments. Stephen L. Scott's work on heterogeneous associative computing is supported by a NASA GSRP-HPCC fellowship and by NRaD.

References

1. K. Batcher, "Staran Parallel Processor System Hardware," *Proc. National Computer Conf.*, AFIPS, 1974, pp. 405-410.

2. J.L. Potter, *Associative Computing — A Programming Paradigm for Massively Parallel Computers*, Plenum Publishing, N.Y., 1992.

3. J.W. Baker and J.L. Potter, "A Model of Computation for Associative Computing," Tech. Report CS-9409-07, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, Sept. 1994.

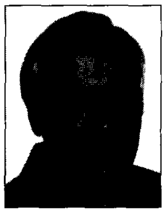
4. A. Falkoff, "Algorithms for Parallel Search Memories," *J. Associative Computing*, Mar. 1962, pp. 488-511.

5. C. Asthagiri and J.L. Potter, "Associative Parallel Common Subexpression Elimination," Tech. Report CS-9405-06, Dept. of Mathematics and Computer Science, Kent State Univ., Kent, Ohio, May 1994.

6. A. Bansal, J.L. Potter, and L. Prasad, "Data-Parallel Compilation and Extending Query Power of Large Knowledge Bases," *Proc. Int'l Conf. Tools With Artificial Intelligence*, CS Press, Los Alamitos, Calif., Order No. 2905-02, 1992, pp. 276-283.

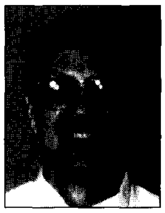
Jerry Potter is a professor of computer science at Kent State University. His research interests include the continuing development of the associative-computing paradigm, the integration of associative SIMD computers with other architectures in a heterogeneous supercomputer environment, and natural-language and artificial-intelligence processing on SIMD computers. While at Goodyear Aerospace, he was involved in the software development for the Staran, Aspro, and MPP SIMD computers.

He received his bachelor's degree from the University of Iowa, his master's from Stevens Institute, and his PhD degree from the University of Wisconsin, Madison.



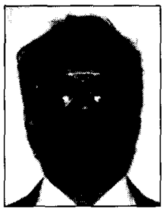
Johnnie W. Baker is an associate professor and coordinator for computer science in the Department of Mathematics and Computer Science at Kent State University. His research interests include parallel algorithms, parallel production systems, applications of parallel computers in artificial intelligence, and parallel computer modeling.

Baker received a BS degree in mathematics from Hardin Simmons University in 1958 and an MS and PhD degrees in mathematics in 1965 and 1968, respectively, from the University of Texas at Austin. He has also published in the areas of Banach spaces and general topology and has served as an editor for *Parallel Processing Letters* since 1991.



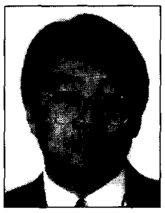
Stephen L. Scott is a PhD candidate in the Department of Mathematics and Computer Science at Kent State University. His research interests include heterogeneous, parallel, distributed, associative, and high-performance computing, operating systems, networking, object-oriented systems, task mapping and scheduling, and multimedia.

He received a BA degree in 1984 from Thiel College, Greenville, Pennsylvania and an MS degree from Kent State University in 1992 after a number of years in the software industry. He is a member of IEEE, ACM, and AFCEA.



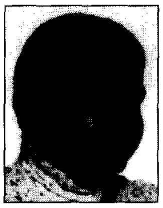
Arvind Bansal is an associate professor of computer science at Kent State University. His research interests are AI tools and languages on scalable massively parallel architectures, associative computing, integration of knowledge retrieval and data-parallel computing, program transformation tools for parallelization, and computational tools for human genome sequencing.

Bansal obtained his B. Tech in electrical engineering in 1979 and his M. Tech in computer science in 1983 from IIT Kanpur, and his PhD in computer science from Case Western Reserve University in 1988.



Chokchai Leangsuksun is a PhD candidate in the Department of Mathematics and Computer Science at Kent State University. His research interests include parallel and heterogeneous computing, networking, operating systems, parallel languages and compilers, user interfaces, and multimedia.

Leangsuksun received the B. Eng. degree in agricultural/civil engineering from Khon Kaen University, Thailand, in 1983 and the MS degree in computer science from Kent State University in 1989. He is a student member of the ACM.



Chandra Asthagiri is a visiting assistant professor at the Computer Science and Engineering Department at Wright State University in Dayton, Ohio. Her areas of interest are parallel processing, optimizing compilers, and databases.

She received her MS and PhD degrees from Kent State University in 1986 and 1992, as well as BS and MS degrees in botany from Madras University in 1970 and 1972.

Readers can contact the authors at the Math Sciences Building, Kent State University Kent, OH 44242; e-mail {potter, jbaker, sscott, chokchai, asthagir, arvind}@mcs.kent.edu.