this message is thus, for every j, $0 \leq j \leq k-1$, the sum of the n jth bits. The originator of the message now computes a number x such that bit j in x is 1 if and only if the corresponding jth sum is greater than $\frac{1}{2}n$. If there exists a majority M, then, in particular, each bit in M appears more than $\frac{1}{2}n$ times; hence, M must be equal to x. We only need to verify that a majority exists. This can be done by sending another active message with the basic command of counting the number of times x appears.

C. Emulating Tree Computation

Consider a computation which is performed on a (logical) tree such that the operands are in the leaves and each internal node represents an arithmetic or logical operation. For example, the tree can be a parse tree of an arbitrary expression. Assume, for simplicity, that the tree is balanced, that it contains n leaves (n is a power of 2), each of them resides at a different node in the network, and that the tree is fixed. Assume, furthermore, that all the operations can individually be performed by shift arithmetic with 1 bit delay. The computation can be performed with only one active message in a similar way to the sorting algorithm. Conceptually, all even nodes perform the operations at height 1, all the nodes with indexes divided by 4 perform the operations at height 2, and so on. The last node performs $\log_2 n$ operations ending with the root. The computation above is pipelined, and hence all operations at the same level of the tree can be performed concurrently. The maximal number of times that an operand is "buffered" is equal to the height of the tree. This is optimal since operations on a path from leaf to root must be carried out sequentially (unless the tree can be reduced). We omit the details of the algorithm.

VII. Conclusions

We studied in this paper the feasibility of designing network protocols that allow efficient implementations of active messages. Active messages are simple commands that are performed on operands which are located at the network interfaces. The new protocols make the communication channel together with the interfaces an environment in which simple computation can be carried out very efficiently. We have shown that these protocols can be implemented without a significant overhead and with little additional hardware, and that they enhance the performance of distributed algorithms on ring networks.

References

[1] D. W. Andrews and G. D. Schultz, "A token ring architecture for local area networks—An update," in *Proc. COMPCOM Fall 1982*, Washington DC, Sept. 1982, pp. 615–624.

[2] A. Barak and A. Shiloh, "A distributed load balancing policy for a multicomputer," Dep. Comput. Sci., Hebrew Univ., Jerusalem, Israel, 1984.

[3] J. A. Bush, G. J. Lipovski, S. Y. W. Su, J. K. Watson, and S. J. Ackerman, "Some implementations of segment sequential functions," in *Proc. 3rd Symp. Comput. Architecture*, Jan. 1976, pp. 178–185.

[4] D. L. Eager, E. D. Lazowska, and J. Zahorjan "Dynamic load sharing in homogeneous distributed systems," Dep. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 84-10-01, Oct. 1981.

[5] ——, "A comparison of receiver-initiated and sender-initiated dynamic load sharing," Dep. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 85-04-01, Apr. 1985.

[6] R. A. Finkel and U. Manber, "DIB—A distributed implementation of backtracking," in *Proc. 5th Int. Conf. Distributed Comput. Syst.*, Denver, CO, May 1985, pp. 446–452.

[7] M. J. Fischer and S. L. Salzberg, "Finding majority among n votes," Dep. Comput. Sci., Yale Univ., New Haven, CT, Res. Rep. 252 Oct. 1982.

[8] D. Jefferson and H. Sowizral, "Fast concurrent simulation using the time warp mechanism," in *Distributed Simulation 1985*, Simulation Series, vol. 15, no. 2, Jan. 1985.

[9] L. Kleinrock *Queueing Systems, Volume 1: Theory*. New York: Wiley, 1975.

[10] P. Krueger and R. A. Finkel, "An adaptive load balancing algorithm for a multicomputer," Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 539, Apr. 1984.

[11] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Caltech Conf. VLSI: Architecture, Design, Fabrication*, 1979, pp. 65–90.

[12] G. J. Lipovski, A. Goyal, and M. Malek, "Lookahead networks," in *Proc. 1982 NCC*, June 1982, pp. 153–165.

[13] M. T. Liu "Distributed loop computer networks," in *Advances in Computers*, M. C. Yovits, Ed. New York: Academic, 1978, pp. 163–221.

[14] M. Livny, "The study of load balancing algorithms for decentralized distributed processing systems," Ph.D. dissertation, Weizmann Inst. Sci., Rehovot, Israel, Aug. 1983; also Dep. Comput. Sci., Univ. Wisconsin, Madison, Tech. Rep. 570, Dec. 1984.

[15] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proc. ACM Comput. Network Perform. Symp.*, Apr. 1982, pp. 47–55.

[16] M. Melman and M. Livny, "The DISS methodology of distributed system simulation," *Simulation*, pp. 163–176, Apr. 1984.

[17] *Operation and Maintenance Manual of PRONET UNIBUS HSB*, Proteon Associates, Inc., Natick, MA, Aug. 1982.

[18] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[19] R. H. Thomas, "A Majority consensus approach to concurrency control for multiple copy database," *ACM Trans. Database Syst.*, vol. 4, pp. 180–209, June 1979.

Speculative Computation, Parallelism, and Functional Programming

F. WARREN BURTON

Abstract — **Many problems can be solved more quickly on parallel machines if some work can be started before it is known to be necessary. If work which is known to be necessary (mandatory work) is given priority over other work (speculative work), then performing speculative work can only speed computation. A simple functional language feature to control speculative work is proposed.**

Index Terms — **Backtracking, combinatorial searching, functional programming, parallelism, priorities, speculation computation.**

I. Introduction

There are many algorithms which have a much higher potential for parallelism if some wasted work will be tolerated. For example, all NP-complete problems can be solved in polynomial time with unbounded parallelism, but cannot be solved sequentially in polynomial time in the worst case, unless $P = NP$. Therefore, these problems have a high potential for parallelism. On the other hand, all NP-complete problems have solutions with a best case polynomial time performance. Therefore, the extra work performed with a high degree of parallelism may be wasted.

Nondeterminism [7], [9], [12] may be used to facilitate speculative evaluation in functional programming languages. With this approach, a problem may be solved in several ways at once. Once a solution is found, other attempts at solving the problem may be terminated. OR-parallelism in Prolog [5], [6] supports speculative evaluation in the case of backtracking. We propose a deterministic feature which has simple semantics and gives the programmer a higher degree of control over speculative work.

II. Speculative Evaluation

As a foundation, let us take a functional language in which parameters can be passed by name or value, with value as the default [2]. The language will support two forms of parallelism. Arguments

Manuscript received April 18, 1985; revised July 29, 1985. This work was supported by the National Science Foundation under Grant ECS-8312748.

The author is with the Department of Electrical Engineering and Computer Science, University of Colorado at Denver, Denver, CO 80202.

0018-9340/85/1200-1190$01.00

to a function may be evaluated in parallel. In addition, streams provide for consumer/producer parallelism. The stream constructor *cons* and selectors *first* and *rest* satisfy the usual stream (list) axiom that $cons(first(x), rest(x)) = x$ whenever x is a stream and $x \neq nil$. However, *cons* may return a result while its second argument is still being evaluated.

We propose the addition of a single primitive function:

$$priority: real \times \text{anytype} \rightarrow \text{anytype}.$$

Informally, $priority(r, e)$ will set up a process to evaluate the expression e with priority r and immediately return the location where the value of e will be placed, once its evaluation has been completed. The result returned by $priority(r, e)$ may be used as a substitute for the value of e until the actual value is required. For example, it may be placed in a stream or other data structure, or passed as a parameter, before the evaluation of e has been completed.

When a computation needs the value of e, for example, to add it to something else, the computation must be suspended until the value has been computed. If it is found that the value of e is not needed, for example, because there are no remaining references to the location where the value is to be placed, then the evaluation may be aborted. This gives the system the ability to terminate unneeded speculative computation.

Semantically, $priority(r, e)$ is equivalent to e, except that a program may terminate even if the evaluation of e does not, provided the value of e is never needed.

We will use the term *speculative computation* for any computation initiated by *priority*, and *mandatory computation* for other computation, except that once the result of a speculative computation is found to be needed, it will become a mandatory computation. Mandatory computation should always be run in preference to speculative computation, so that the use of speculative computation will never slow down a program (except for additional overheads).

The problem of aborting unneeded processes in a tree of processes is considered by [8] and [11]. The same approach can be used to upgrade speculative computation to mandatory computation. A functional program can be run on a network of processors [3]. In this case the priorities should be used for guidance by the scheduler. However, a low-priority process may run on one processor while a high-priority process is inactive on another, busier processor.

III. Examples

In this section we will consider several examples where speculative computation may be used in solving combinatorial problems.

In each example we will search a tree of problem states for an answer node. We will assume that all answer nodes are leaves, but a leaf may not be an answer node. Each node that is not a leaf will have two children.

For example, consider the sum of subsets problem, which is NP-complete. The problem is: given a set of n integers, $A = \{a_1, a_2, \cdots, a_n\}$, and an integer m, determine if there is a subset of A which sums to exactly m. A subset of A will be associated with each node. The root will be the empty set. If the subset of a node sums to at least m or the node is at depth n, then the node is a leaf. Otherwise the node will have two children. If subset B is associated with a node at depth k, then the subsets associated with the children of the node will be B (again) and $B \cup \{a_{k+1}\}$. Hence, the nodes at level k will correspond to the subsets of $\{a_1, a_2, \cdots, a_k\}$ which have not been pruned. A node is an answer node if its subset sums to exactly m.

In each example we will assume that we have the following functions:

leaf: node → Boolean
answer: node → Boolean
left: node → node
right: node → node.

The function *leaf* tells whether a node is a leaf. If it is a leaf, then *answer* tells whether it is a solution to the problem. If a node is not a leaf, the *left* and *right* will generate its children.

```
solve(node) =
    if leaf(node) then
        if answer(node) then node else FAIL
    else
        or(solve(left(node)), solve(right(node)))
        where or(a, name b) = if a = FAIL then b else a
```

Fig. 1. A sequential backtracking algorithm.

Backtracking

Fig. 1 contains a sequential backtracking algorithm which will search a tree rooted at the node passed to it and return the leftmost answer node, if the tree contains an answer node, or the special value *FAIL* otherwise. Notice that the second parameter of *or* is passed by name, so the right subtree is searched only if the search of the left subtree returns the value *FAIL*.

There is an obvious opportunity for speculative computation in this problem. The left and right subtrees can be searched in parallel. This can be repeated recursively, with the left and right subtrees of each subtree being searched in parallel. The number of searchable subtrees will grow exponentially with the depth of the search. On any realistic system, the degree of parallelism will be bounded, so the potential for parallelism in a large problem will far exceed that attainable in the system. It is important that computing resources be concentrated on those subproblems where they will do the most good.

We will require the speculative algorithm to be deterministic and to return the leftmost answer node, just as the sequential algorithm did. (Nondeterminism complicates the logic of a functional program. While *priority* may be used to control nondeterministic computations, we will restrict our attention to deterministic algorithms in this paper. In fact, one of the primary advantages of our approach is that, in many situations, it allows speculative evaluation without introducing the problems normally associated with nondeterminism. However, we admit that there appear to be problems where nondeterminism offers a real advantage.)

Since we require the leftmost answer node, the solution to the search of a right subtree is of interest only if no answer can be found in the left subtree. Therefore, whenever we initiate a search of a pair of subtrees, we want all searching in the left subtree to have priority over searching in the right subtree. We can achieve this by assigning a range of priorities to each subtree so that each lower level recursive search of a subtree is assigned a value in the range. Whenever we split a search into two lower level searches, we divide the range into half, assigning the higher half to the left subtree. Fig. 2 gives the complete search function. All work associated with searching the subtree passed to *solve* will have priorities in the range $(low, high]$. When we initiate a search of a subtree, we give the processing of the root node the highest priority in the range. Notice that the *or* function has been modified so both parameters are passed by value. This is necessary to initiate the speculative processing of the right subtree.

In the case of a single processor, the algorithm in Fig. 2 will expand the same nodes as the algorithm in Fig. 1. With unbounded parallelism, all paths to leaves will be examined in parallel. With bounded parallelism, the nodes to the left will tend to be expanded first.

Breadth-First Search

In the sum of subsets problem, the tree we search is of finite depth. This is not the case with all problems of this general type. For example, suppose we are trying to solve a puzzle where a "move" takes us from one problem state to another. If a sequence of moves can lead to a problem state identical to a previous problem state, then we have a tree with at least some infinitely long paths. The

```
solve(node, high, low) =
    if leaf(node) then
        if answer(node) then node else FAIL
    else
        or(priority(high, solve(left(node), high, mid)),
           priority(mid, solve(right(node), mid, low)))
        where mid = (high + low) / 2
        and or(a, b) = if a = FAIL then b else a
```

Fig. 2. A speculative backtracking algorithm.

backtracking algorithm will fail to find an answer node if it comes across an infinite path prior to the leftmost answer mode. We can overcome this disadvantage by using a breadth-first search. We will search for the leftmost minimal depth mode.

With our previous example, in Fig. 2, we were only interested in the result of the search of the right tree in cases where no node was found in the left tree. In this example, we will search both trees in parallel and look at part of the result of each search. When *solve* is applied to a node, a stream of values will be returned. For each level on which no answer node is found, a *FAIL* element will be placed in the stream. When an answer node is found, the leftmost answer node at its level will be placed in the stream as the final element.

The breadth first *solve* function is shown in Fig. 3. Notice that the priority of the search decreases with the depth, encouraging a breadth-first expansion. The function *combine* merges the solutions of two subproblems, reporting a single *FAIL* for each level on which both fail. Once an answer node has been found, neither stream is inspected further, so the expansion of deeper nodes may be aborted. This algorithm involves more communication than a sequential breadth search algorithm. However, in [4] it is shown that the total amount of communication is proportional to the number of nodes expanded.

Least-Cost Search

With many problem, all leaf nodes are answer nodes and the best answer node it desired. For example, with various formulations of the *traveling salesman problem,* the answer nodes represent complete tours of n cities. A least-cost answer node is a tour of minimal length. With problems of this type, it is often possible to establish a lower bound on the cost of any answer node in the subtree rooted at a given node. We will assume that the function *cost* will return this lower bound for any node, and will give the true cost of a leaf node.

In a sequential least-cost search, the root node is the only *live node* at the start of the search. The algorithm repeatedly selects the least-cost live node, removes it from the set of live nodes, generates its children, and adds the children to the set of live nodes. The first answer node selected is the least-cost answer node. The reader is referred to [10] for a more detailed discussion of least-cost searching.

A speculative least-cost search algorithm can now be defined. As before, we will have a process for each node, and each process will return a stream of results.

Let us consider the information we need from *solve*. Suppose that we find that the least-cost answer node in one subtree has cost c. In order to ensure that this is the least-cost answer node overall, we must expand every node in the other subtree that has a cost bound less than c. At the same time, we would like to be able to avoid expanding nodes with a cost greater than c. To achieve this, *solve* will return a stream of increasing cost lower bounds. The cost of each node expanded by *solve* will be placed in the stream before the node is expanded. Once a least-cost answer node is found, its cost will be the final item in the stream. (If the answer node is desired as well as its cost, this should also be returned.)

A speculative least-cost search algorithm is given in Fig. 4. Notice how *short_merge* drops all items in one stream which cost more than the optimal solution (final item) in the other stream. This allows the system to kill the tasks which would expand the unneeded nodes.

```
solve(node, depth) =
    if leaf(node) then
        if answer(node) then cons(node, nil) else cons(FAIL, nil)
    else
        cons(FAIL, combine(do(left(node)), do(right(node))))
        where do(node) = priority(1 / depth, solve(node, depth + 1))
        and combine (a, b) =
            if a = nil then b
            else if b = nil then a
            else if first(a) ≠ FAIL then a
            else if first(b) ≠ FAIL then b
            else cons(FAIL, combine(rest(a), rest(b)))
```

Fig. 3. A speculative breadth-first search algorithm.

```
solve(node) =
    if leaf(node) then nil
    else short_merge(generate(left(node)), generate(right(node)))
    where short_merge(a, b) =
        if a = nil or b = nil then nil
        else if first(a) < first(b) then cons(first(a), short_merge(rest(a), b))
        else cons(first(b), short_merge(a, rest(b)))
    and generate(node) = cons(bound, priority(bound, solve(node)))
    where bound = cost(node)
```

Fig. 4. A speculative least-cost search algorithm.

The communication costs in the speculative least-cost algorithm are high. The sum of the lengths of the streams produced by *solve* is approximately equal to the sum of the depths of the expanded nodes rather than the number of expanded nodes. (In addition, some information about those unexpanded nodes immediately below expanded nodes must filter up to the level where such nodes are pruned.) If the tree is relatively well balanced, then the communication costs for n nodes is $O(n \log n)$. However, the worst cost time for unbalanced trees is $O(n^2)$.

The communication costs appear to be an inherent characteristic of parallel least-cost search algorithms, and are found in nonfunctional parallel versions [1]. The problem is that the least-cost answer node may occur anywhere in the tree. Once it is found, or is thought to be found, this information must be communicated to the rest of the tree. We note that even the sequential algorithm may require $O(n \log n)$ time (excluding the time required to test and expand nodes). This is because heap (priority queue) operations require $O(\log k)$ time for a heap of size k. If the expanded part of the tree is well balanced, then k may be $O(n)$.

It is interesting to note that well-balanced trees, where both the sequential and speculative algorithms require $O(n \log n)$ work, are the trees with the greatest potential for parallelism. On the other hand, it is possible for a sequential algorithm to go directly to the optimal answer node without expanding any node not on the path to the optimal solution. In this case we have a highly unbalanced tree, which is the worst case for communication costs in the speculative algorithm. Furthermore, any speculative work will have been wasted in this case.

Our primary concern is with functional programming, and more specifically with demonstrating what can be done with speculative

evaluation. We leave to the reader the problem of finding better parallel search algorithms.

REFERENCES

[1] F. W. Burton, M. M. Huntbach, G. P. McKeown, and V. J. Rayward-Smith, "Parallelism in branch-and-bound algorithms," Sch. Comput. Studies and Accountancy, Univ. East Anglia, Norwich, England, Rep. CSA3/1983, 1983.

[2] F. W. Burton, "Annotations to control parallelism and reduction order in the distributed evaluation of functional programs," *ACM Trans. Progr. Lang. Syst.*, vol. 6, pp. 159–174, Apr. 1984.

[3] F. W. Burton and M. M. Huntbach, "Virtual tree machines," *IEEE Trans. Comput.*, vol. C-33, pp. 278–280, Mar. 1984.

[4] F. W. Burton, "Controlling speculative computation in a parallel functional programming language," in *Proc. 5th Int. Conf. Distributed Comput. Syst.*, Denver, CO, May 1985, pp. 453–458.

[5] J. S. Conery and D. F. Kibler, "Parallel interpretation of logic programs," in *Proc. Conf. Functional Progr. Lang. Comput. Architecture*, Portsmouth, NH, Oct. 1981, pp. 163–170.

[6] J. S. Conery, "The AND/OR process model for parallel interpretation of logic programs," Ph.D. dissertation, Dep. Comput. Sci., Univ. California, Irvine, Tech. Rep. 204, June 1983.

[7] D. P. Friedman and D. S. Wise, "An indeterminate constructor for applicative programming," in *Conf. Rec., 7th ACM Symp. Principles Progr. Lang.*, 1980, pp. 245–250.

[8] D. H. Grit and R. L. Page, "Deleting irrelevant tasks in an expression-oriented multiprocessor system," *ACM Trans. Progr. Lang. Syst.*, vol. 3, pp. 49–59, Jan. 1981.

[9] P. Henderson, *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[10] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Woodland Hills, CA: Comput. Sci. Press, 1978.

[11] P. Hudak and R. M. Keller, "Garbage collection and task deletion in distributed applicative processing systems," in *Proc. ACM Symp. LISP and Functional Progr.*, Pittsburgh, PA, Aug. 1982, pp. 168–178.

[12] J. McCarthy, "A basic mathematical theory of computation," in *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg, Eds. Amsterdam, The Netherlands: North-Holland, 1963, pp. 33–70.