Massively Parallel Searching for Better Algorithms or How to Do a Cross Product with Five Multiplications

JOHN GUSTAFSON[1] **AND SRINIVAS ALURU**[2]

[1]*Ames Laboratory, Iowa State University, Ames, IA 50011; email: gus@scl.ameslab.gov*
[2]*Department of Computer and Information Science, Syracuse University, Syracuse, NY 13244; email: aluru@top.cis.syr.edu*

ABSTRACT

A number of "tricks" are known that trade multiplications for additions. The term "tricks" reflects the way these methods seem not to proceed from any general theory, but instead jump into existence as recipes that work. The Strassen method for 2 × 2 matrix product with seven multiplications is a well-known example, as is the method for finding a complex number product in three multiplications. We have created a practical computer program for finding such tricks automatically, where massive parallelism makes the combinatorially explosive search tolerable for small problems. One result of this program is a method for cross products of three-vectors that requires only five multiplications. © 1996 John Wiley & Sons, Inc.

1 INTRODUCTION

Humans have talents that are hard to program. Driving a car, recognizing continuous speech, and playing chess have proved far more challenging to computers than they have to people. To this class we can probably add *algorithm optimization.* How is it that a mathematician can look at a simple algorithm for complex product like

$$u \leftarrow a \times c - b \times d$$
$$v \leftarrow a \times d + b \times c$$

and discover that the number of multiplication operations can be reduced to three. One method, which seems far from obvious, is

$$s_1 = a - b \qquad m_1 = c \times s_1$$
$$s_2 = a + b \qquad m_2 = d \times s_2$$
$$s_3 = c - d \qquad m_3 = b \times s_3$$
$$u = m_1 + m_3$$
$$v = m_2 + m_3$$

The conventional method for 2 × 2 matrix products calls for eight multiplications and four additions. (In this article, we equate additions and subtractions in assessing operation count because they are computationally similar).

The trick behind the Strassen method for 2 × 2 matrix products [20] is even more abtruse and baffling than the shortcut for complex products (Yuval presents a possible derivation, see [22]):

Received December 1993
Revised August 1995

© 1996 John Wiley & Sons, Inc.

Scientific Programming, Vol. 5, pp. 203–217 (1996)
CCC 1058-9244/96/030203-15

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\begin{array}{ll} s_1 = a_{12} - a_{22} & m_1 = s_1 \times s_6 \\ s_2 = a_{11} + a_{22} & m_2 = s_2 \times s_7 \\ s_3 = a_{11} - a_{21} & m_3 = s_3 \times s_8 \\ s_4 = a_{11} + a_{12} & m_4 = s_4 \times b_{22} \\ s_5 = a_{21} + a_{22} & m_5 = a_{11} \times s_9 \\ s_6 = b_{21} + b_{22} & m_6 = a_{22} \times s_{10} \\ s_7 = b_{11} + b_{22} & m_7 = s_5 \times b_{11} \\ s_8 = b_{11} + b_{12} & \\ s_9 = b_{12} - b_{22} & \\ s_{10} = b_{21} - b_{11} & \end{array}$$

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7 \end{aligned}$$

This algorithm was later improved by Winograd [1, p. 247], who reduced the number of additions from 18 to 15:

$$\begin{array}{lll} s_1 = a_{21} + a_{22} & m_1 = s_2 \times s_6 & t_1 = m_1 + m_2 \\ s_2 = s_1 - a_{11} & m_2 = a_{11} \times b_{11} & t_2 = t_1 + m_4 \\ s_3 = a_{11} - a_{21} & m_3 = a_{12} \times b_{21} & \\ s_4 = a_{12} - s_2 & m_4 = s_3 \times s_7 & \\ s_5 = b_{12} - b_{11} & m_5 = s_1 \times s_5 & \\ s_6 = b_{22} - s_5 & m_6 = s_4 \times b_{22} & \\ s_7 = b_{22} - b_{12} & m_7 = a_{22} \times s_8 & \\ s_8 = s_6 - b_{21} & & \end{array}$$

$$\begin{aligned} c_{11} &= m_2 + m_3 \\ c_{12} &= t_1 + m_5 + m_6 \\ c_{21} &= t_2 - m_7 \\ c_{22} &= t_2 + m_5 \end{aligned}$$

Where is the pattern in these methods? Although the theory of trilinear forms [16, 17] has helped guide some shortcuts, it appears that these methods are the result of inexplicable intuitive leaps by some very bright people.

In August 1991, the authors began an experiment to see if computers, especially massively parallel computers, could discover these tricks, and perhaps find new ones. Beginning with brute force search methods that ran for days on very small algorithms, the program became gradually more sophisticated and able to handle interesting problems within the limits of our patience. Recently, an nCUBE 2 with 256 processors revealed that it is possible to compute the cross product of two 3-dimensional vectors using only five multiplications, and this method is new to the best of our knowledge. The conventional method requires six multiplications and three subtractions:

$$\begin{aligned} &(c_1\vec{i} + c_2\vec{j} + c_3\vec{k}) \\ &\quad = (a_1\vec{i} + a_2\vec{j} + a_3\vec{k}) \times (b_1\vec{i} + b_2\vec{j} + b_3\vec{k}) \\ &\quad = (a_2b_3 - a_3b_2)\vec{i} + (a_3b_1 - a_1b_3)\vec{j} \\ &\qquad + (a_1b_2 - a_2b_1)\vec{k} \end{aligned}$$

The algorithm with five multiplications, found by the computer program, is:

$$\begin{array}{lll} s_1 = a_1 - a_2 & m_1 = a_3 \times b_1 & t_1 = m_3 - m_2 \\ s_2 = b_2 + b_3 & m_2 = a_1 \times b_3 & \\ s_3 = s_1 - a_3 & m_3 = s_1 \times s_2 & \\ s_4 = b_1 - s_2 & m_4 = b_2 \times s_3 & \\ & m_5 = a_2 \times s_4 & \\ & c_1 = m_4 - t_1 & \\ & c_2 = m_1 - m_2 & \\ & c_3 = t_1 - m_5 & \end{array}$$

The rest of the article describes our search program. Shortcut ways of computing the given expressions are found by doing a search among all possible expressions that can be derived from the given set of variables and operations. Because this exhaustive enumeration has a combinatorially large number of expressions to explore, strategies are developed which reduce the search. Several "pruning" strategies are used to avoid the exploration of unpromising subtrees. Parallel computers are employed to conduct the search in parallel, to achieve higher speed.

2 PROBLEM SPECIFICATION

The problem can be defined as follows:

Given: A set of variables

a budget of M multiplications and A additions
and a set of goal expressions

Find: A sequence of ($exp1$ op $exp2$) triples where $exp1$ and $exp2$ are selected from either the set of variables or previously computed expressions, and op is chosen from the remaining multiplications or additions that compute the goal expressions, and that minimize the total number of operations or the number of operations of a particular type.

We are interested in minimizing the number of operations of a particular type because the relative cost of the operation types is different in general. For example, in Strassen's matrix product algorithm, the variables involved could themselves be matrices, and matrix products are costlier than matrix sums. Any algorithm for finding the product of two $k \times k$ matrices in M multiplications can be used recursively to find the product of two $n \times n$ matrices ($n > k$) in $O(n^{\log_k M})$ time [1, 20]. For such a problem, we are obviously interested in minimizing the number of multiplications, even at the cost of increasing the total number of addition operations.

For the class of problems addressed in this article, addition (+), subtraction (−), and multiplication (×) operations are sufficient. Addition and multiplication are commutative, while subtraction is not. For the purposes of uniformity, we introduce a notation "reverse subtraction" (~), with the associated meaning that $exp1 \sim exp2$ is the same as $exp2 - exp1$. With this, we can impose an ordering on $exp1$ and $exp2$ and explore only cases with $exp1 < exp2$ without omission. Also, we use the term "add" to denote any of addition/subtraction/reverse subtraction. Throughout the article, we assume that operation cost is data independent. We also use the term "product" to refer to multiplication of entities like complex numbers and matrices and reserve the term "multiplication" for real numbers.

3 MODELING AS A SEARCH PROBLEM

Consider the set of all possible expressions that can be derived from the given set of variables and operations. These can be thought of as a graph with each node representing an expression (Fig. 1).

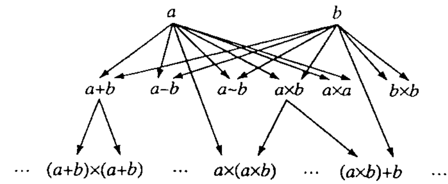


FIGURE 1 Partial search graph for a two-input problem.

We use the term "expression" for a node as long as it results in no confusion, because each node can be specified by the expression it represents. The graph $G(V, E)$ can be defined as follows:

1. Each of the given set of variables forms a node in the graph.
2. If two expressions $e_1, e_2 \in V$, then (e_1 op e_2) $\in V$ for every choice of operation op.

We can also think of edges from e_1 and e_2 to the node (e_1 op e_2). Clearly, this graph represents all possible expressions that can be generated from the given variables and operation types. Note that each node represents a unique expression and the way of generating it starting from the variables can easily be found by following the edges into the node representing this expression. However, two or more expressions could be mathematically equivalent.

The number of operations required to compute any expression in the graph can be found by recursively following the edges coming into the expression. The number of operations for computing e_1 op e_2 is one more than the operations required for computing e_1 and e_2, except that expressions common to the paths for e_1 and e_2 need be computed only once.

A set of nodes in this graph whose expressions are mathematically equivalent to the goal expressions constitutes a way of computing the goal expressions with the associated number of operations. The problem then translates to finding the appropriate set of nodes such that the associated number of operations is the minimum over all such possible sets.

Because there is no budget for the number of operations, the graph has an infinite number of nodes. The nodes of the graph can easily be ordered according to the number of operations required to compute the expressions at the nodes. The nodes at level 0 constitute the given set of variables. The nodes at level i consist of expressions that can be computed using exactly i opera-

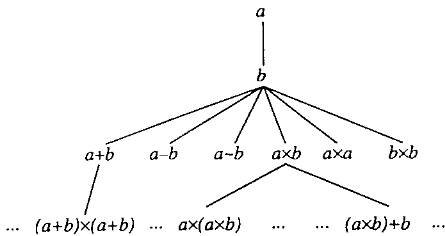


FIGURE 2 Partial search tree for a two-input problem.

tions. We can easily place a bound on the search because we are interested in minimizing the number of operations. If a known way of generating the goal expressions needs k operations, we need only look at nodes at levels less than k.

Because the graph has a combinatorially explosive number of nodes, it would be impractical to generate and store the graph during the search process. Also, solutions are difficult to identify, because matching subgraphs to the goal expressions (as algebraic equivalents) is itself a combinatorial problem. To avoid these problems, it is convenient to do a depth-first search on a tree as shown in Figure 2.

The tree is constructed as follows: The given set of variables forms the first m levels of the tree, one node at each level. The children of node i are all the expressions that can be formed using an operation and any two expressions on the path from the root to node i. The goal is to search for a path in the tree which contains expressions equivalent to goal expressions. Because each node accounts for one operation, paths containing more than the budgeted number of operations need not be explored.

4 THE BASIC SEARCH ALGORITHM

The *Search* algorithm is simply a depth-first search on the tree described in the previous section. Note that we do not allow operations involving specific integers; only letters involving unknown quantities. The naive form of the basic algorithm is easily stated:

Input: m variables, a budget consisting of M multiplications, and A adds, and n goal expressions to be determined.
Output: A way of computing the goal expressions without exceeding the budgeted number of operations or a statement that no such method exists.
Method: Initialize the first m levels to the input variables. Initialize *level* to $m + 1$.

```
Search()
For input1 = 1 to level − 1:
    For input2 = 1 to level − 1:
        For each operation type left in the budget:
            Apply operation to the expressions at input1 and input2.
            Remove the operation from the budget.
            level ← level + 1.
            If the new expression is a goal expression not yet found,
                mark the goal expression as found.
            If all the goals are found, return the solution.
            If level < m + M + A, Search().
            Unmark the goal expression found, if any.
            Restore operation to the budget.
        End For.
    End For.
End For.
```

In principle, questions like, "Is there a method of finding the product of two complex numbers involving three real multiplications and five real adds?" can be answered by exhaustive search. This approach is naive in general because there are so many possible algorithms with these constraints. Because the two input expressions can be taken from any previously computed expression, the number of input combinations is $(A + M + m - 1)!^2/(m - 1)!^2$. There are $\binom{A+M}{M}$ ways to place the multiplications in the set of steps. Because the "add" operations can be any of addition, subtraction, or reverse subtraction, there are 3^A possible sets of add operations for any specified placing of multiplications. The total number of elements in the search space by the naive

method is

$$\frac{(A+M+m-1)!^2}{(m-1)!^2}\binom{A+M}{M}3^A$$

For the complex product with four inputs and a budget of three multiplications and five adds, this value is approximately 6.02×10^{17}.

A massively parallel collection of 8000 processors, each checking one million nodes per second, would take more than 2 years in the worst case that there is only one such algorithm and that it is the last one checked. The existence of multiple algorithms in the search space and termination of the search when the first is found might reduce this time by an order of magnitude, but it would still not be the sort of problem casually attempted with 1992 technology!

To make the problem more tractable, we began the accumulation of "pruning rules" for eliminating subtrees in the search without any danger of missing a solution. These pruning rules are conservative because they still allow the resulting tree to be called an exhaustive search. Later we consider pruning rules that seem to greatly assist in finding clever methods that trade multiplications for adds, but cannot be used to prove the nonexistence of a method with a given operation budget.

5 CONSERVATIVE PRUNING RULES

To motivate the need for pruning rules, look at what is probably the simplest trick in all of elementary algebra: $a^2 - b^2 = (a+b)(a-b)$. The factoring reduces the number of multiplications from two to one, at the cost of increasing the number of adds from one to two. If we set up a search tree using Algorithm A, the first step could be any of

$$\begin{array}{cccc} a \sim a & a-a & a+a & a\times a \\ a \sim b & a-b & a+b & a\times b \\ b \sim a & b-a & b+a & b\times a \\ b \sim b & b-b & b+b & b\times b \end{array}$$

Because addition and multiplication are commutative, and we include both subtraction and reverse subtraction, we need not include both *input1 op input2* and *input2 op input1*. Therefore, requiring the first input to be from the level on or above that of the second input is a conservative pruning rule. This pruning rule is easily implemented by changing loop control variables instead of using explicit "if" statements. Some of the pruning strategies are rather straightforward and obvious. Nevertheless, they are very important in view of the significant amount of reduction in the search.

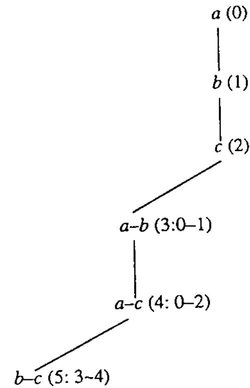


FIGURE 3 A typical path showing expressions and how they are derived.

Before we enumerate the pruning rules, it is necessary to introduce some notions. The program always keeps track of the path from the root of the tree to the current node along with the way each expression on the path is derived. Expressions are represented as polynomials in the input variables. The terms of the polynomials are ordered to make it easy for addition and subtraction operations and to check mathematical equality of expressions. The expressions on the path are numbered starting from the root. The way an expression is derived is represented by the numbers denoting the parent expressions and the operation used. Figure 3 makes the ideas clear.

We can also define a lexicographic ordering on the expressions based on the way they are derived. Let $f_1\ op_1\ g_1$ and $f_2\ op_2\ g_2$ denote two expressions e_1 and e_2 on a path. We say e_1 precedes e_2 in lexicographic order if $g_1 < g_2$ or if $g_1 = g_2$ and op_1 precedes op_2 in lexicographic order or if $g_1 = g_2$, $op_1 = op_2$, and $f_1 < f_2$. The ordering on operations is defined to be "$\sim$" < "$-$" < "$+$" < "$\times$". There are several possible choices for defining a lexicographic ordering and there is nothing special about the specific order chosen. The ordering is useful in defining some pruning rules. A number of pruning strategies are discussed below. For each pruning rule that is used, we state the rule along with a brief justification wherever it is appropriate.

1. *Explore only paths of length less than or*

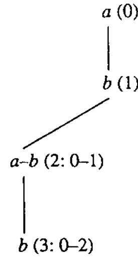


FIGURE 4 A path showing two identical expressions.

equal to the total number of operations plus the number of variables.
Each expression on a path (except the variables) is formed by one operation. Therefore, paths of length more than the budgeted number of operations plus the number of variables cannot provide the desired solution.

2. *Restrict the number of operations of each particular type.*
If the budget of operations of a particular type is consumed on a path from the root to a node, all children of the node using the particular operation can be pruned.
3. *Subtracting an expression from itself is not useful, and can be excluded from the search tree.*
This assumes that 0 is not a goal expression.
4. *If the expression at a node is the same as the expression at one of its parent nodes, delete the node and the subtree under it.*
This is obvious because no shortcut will have the same expression computed twice. In Figure 4, the node labeled $b(3:0-2)$ is deleted because it represents the same expression as the node labeled $b(1)$.
5. *Eliminate expressions with a leading negative term, except when one of the goal expressions contains a leading negative term.*
Recall that the terms of expressions are ordered and hence there is no ambiguity in deciding if an expression has a leading negative term. An expression is negative if it has a leading negative term and positive otherwise. For each path $P^{(1)}$ containing a negative expression, we show the existence of another path $P^{(2)}$ which does not contain such expressions and is a solution if $P^{(1)}$ is a solution. $P^{(2)}$ satisfies the property that every expression in it is the same as the corresponding expression in $P^{(1)}$ or its negative. We construct $P^{(2)}$ starting from the root, such that at each stage the constructed partial path satisfies the above property. To start, the first m levels constitute the variables and hence satisfy the property. Let $e_3^{(2)}$ be the next expression to be added to $P^{(2)}$ and $e_3^{(1)}$ be the corresponding expression on $P^{(1)}$. Let $e_3^{(1)} = e_1^{(1)}$ *op* $e_2^{(1)}$. We can show that for every choice of *op* and for every possible combination of signs of $e_1^{(1)}$ and $e_2^{(1)}$, $e_3^{(2)}$ can be constructed such that $e_3^{(2)} = -e_3^{(1)}$ if $e_3^{(1)}$ is negative and $e_3^{(2)} = e_3^{(1)}$ otherwise. For example, if $e_3^{(1)} = e_1^{(1)} + e_2^{(1)}$ and $e_3^{(1)}$ and $e_2^{(1)}$ are negative, let $e_3^{(2)} = e_1^{(2)} \sim e_2^{(2)}$. Because $e_1^{(2)} = e_1^{(1)}$ and $e_2^{(2)} = -e_2^{(1)}$, $e_3^{(2)} = -e_2^{(1)} - e_1^{(1)} = -e_3^{(1)}$, as desired. If all goal expressions are positive and $P^{(1)}$ contains a solution, then $P^{(2)}$ must also contain a solution by the above construction. Therefore, we can prune the subtree under any expression that is negative.
6. *The level of the second input node should be the same as or greater than the level of the first input node.*
This rule was explained in the beginning of this section.
7. *Eliminate exploring paths with an identical set of expressions, by requiring expressions on each path to be in increasing lexicographic order starting from the root.*
For each path $P^{(1)}$ that does not contain expressions in increasing lexicographic order, we can show the existence of another path $P^{(2)}$ containing the same expressions as $P^{(1)}$, but in the proper order. Let $e_1^{(1)}$ and $e_2^{(1)}$ be two expressions on $P^{(1)}$ that do not conform to the lexicographic order. Without loss of generality, let $e_1^{(1)}$ be the expression nearer to the root. Because $e_2^{(1)}$ precedes $e_1^{(1)}$ in lexicographic order, the expressions needed to compute $e_2^{(1)}$ appear before $e_1^{(1)}$ on the path, and therefore $e_1^{(1)}$ and $e_2^{(1)}$ can be swapped. We can construct $P^{(2)}$ by systematically swapping every two expressions on $P^{(1)}$ that do not conform to the lexicographic order. Because $P^{(1)}$ and $P^{(2)}$ contain the same expressions, $P^{(2)}$ represents a solution if $P^{(1)}$ represents a solution.
Using this rule, we can prune each node that precedes its parent in lexicographic order. In Figure 5, the two paths shown have identical expressions and hence the

subtrees under them are identical. The tree is pruned at node labeled $a - b(3:0 - 1)$ because it precedes its parent in lexicographic order. This avoids computing the same subtree twice.

8. *Among all the children of a node with mathematically equivalent expressions, choose the one that is smallest in the lexicographic ordering.*
   This rule is also useful to avoid computing duplicate subtrees. The expression that is smallest in the lexicographic ordering is chosen because by rule 7, the subtree under such a node contains the subtree under a node representing the same expression and having the same parent node. For example, in Figure 6, the node labeled $b - c(5:3 \sim 4)$ is pruned as it succeeds the node labeled $b - c(5:2 - 3)$ in lexicographic order and has a common parent.
9. *Levels to explore should be at least as many as goal expressions to be found.*
   A path representing a solution should contain all the goal expressions. Paths with not enough room for all the goals need not be explored.
10. *In a shortcut, each expression is used at least once.*
    Each expression is formed by at most two expressions on the current path. Also, the number of expressions on the path is bounded. This places a limit on the number of expressions that can be used in forming subsequent expressions on the path. If there is no possibility that all the expressions (except the goal expressions) are used at least once, the tree can be pruned at this node.

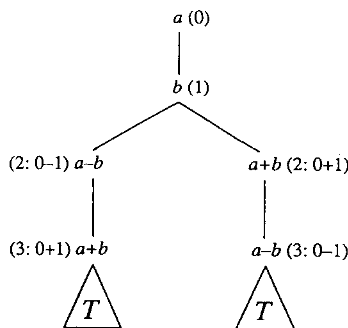


FIGURE 5 Search tree with two paths having identical set of expressions.

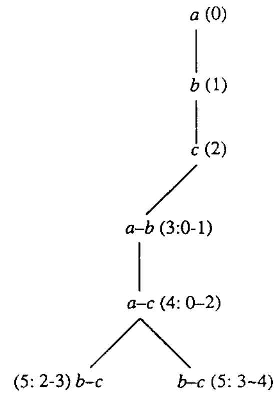


FIGURE 6 Search tree with two children of a node representing the same expression.

Let l = length of the path constructed so far, g = number of goals yet to be found, and u = number of unused expressions on the path. As only paths of length at most $m + M + A$ are explored, the constructed path can be extended to contain $(m + M + A) - l$ more expressions and at most $2 \times (m + M + A - l)$ different expressions could be used to construct these. There are u expressions on the path explored that are unused and $(m + M + A - l - g)$ expressions on any extended path that have to be used. Therefore, any extension of the path could be a shortcut and a solution only if $u + (m + M + A - l - g) \leq 2 \times (m + M + A - l)$ or if $u \leq m + M + A - l + g$. Otherwise, the subtree under the path constructed so far can be pruned.

It should be noted that some of the pruning rules are more expensive than the others. This can be viewed as moving the tree traversal cost to the node expansion cost, and the trade off in cost should be considered. For example, rule 4 involves determining whether the expression is identical to one of its ancestors. Implementing this rule is fairly cheap and effective when the node is close to the root because there are few nodes to compare and large subtrees to prune. However, when it is close to the maximum level (as defined by rule 1), the cost of comparison increases dramatically but the payoff decreases significantly. Rule 8 involves comparing the expression to its siblings generated before, making it impractical to implement even for nodes fairly close to the root. In fact, the siblings are not available in a depth-first search as only

the path from the root to the current node is stored. In such a case, a restricted version of the rule might be more useful. For example, rule 8 implies that any expression in first degree of length two should be constructed using only the given variables and this can be checked with just two comparisons. Except for rules 4 and 8, the remaining rules can be implemented using a constant number of operations irrespective of the position in the tree of the node being tested.

There are other pruning rules that guarantee that if a solution were to be found in one of the deleted subtrees, then an equally economical solution is guaranteed to be found in another subtree to be explored. These rules usually stem from the commutativity of the operation the goal expressions represent or from symmetry considerations.

As an example, consider the complex product, which is a commutative operation.

$$(a + \mathbf{i}b)(c + \mathbf{i}d) = (c + \mathbf{i}d)(a + \mathbf{i}b)$$

Exchanging a with c and b with d in any solution that computes the complex product also gives a solution for the same. Hence, in the search program, if a path from the root to a node can be derived from another path by the above exchange, the subtree under that node need not be explored. In fact, we can easily derive the missed solutions from the solutions found.

Such a pruning rule is very useful because it prunes the tree at the first level. For, if a path can be derived from another, the first node in the path can also be derived and the tree should have been pruned at the first node on the path itself.

As another example of a similar rule, consider the 2 × 2 matrix product. Interchanging the rows of the first matrix and/or the columns of the second matrix does not affect the goal expressions to be computed. This prunes three of four nodes at the first level of the tree.

6 AGGRESSIVE PRUNING RULES

As the size of the search tree is exponential both in the number of variables and the number of operations, we need as many pruning rules as possible to be able to tackle interesting problems in reasonable time. Due to this, we added some pruning rules that seem to be intuitively appealing, without the guarantee that they eliminate only unpromising subtrees. These rules greatly assist in finding clever

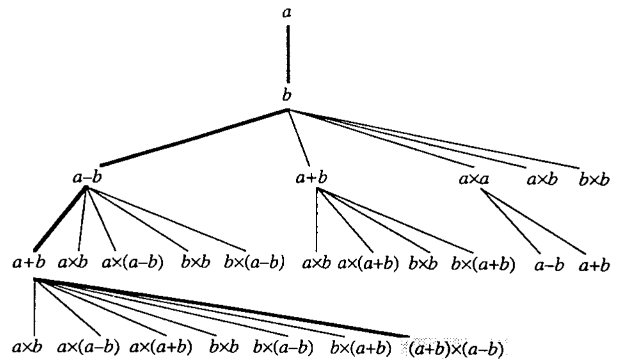


FIGURE 7 Aggressively pruned search tree for $a^2 - b^2$.

solutions but cannot be used to prove the nonexistence of any methods with a given operation budget:

1. Eliminate expressions with degree higher than the highest degree among goal expressions.
2. If the goal expressions are all homogeneous, do not allow expressions with terms of different degree.
3. If the goal expressions do not contain terms of the form $n \times expression$, where n is an integer, $|n| > 1$, and *expression* is a product of input variables, then we can also exclude all operations that result in such terms.

This set of pruning rules, along with the conservative pruning rules, reduces the number of possibilities for the $a^2 - b^2$ search space from 15,552 to a much more manageable 24 expressions. The savings are more dramatic for larger search trees, of course. The resulting search tree for the computation of $a^2 - b^2$ is shown in Figure 7.

A few more pruning rules were found useful in dealing with goal expressions denoting the product of two mathematical entities like complex numbers, vectors, and matrices. None of the goal expressions for such a product contains a term with two variables drawn from the same operand and none of the known shortcuts contains any intermediate expressions with such terms.

We define the structure of a term to be the sequence of operand names from which the variables forming the term are drawn. For example, with respect to the complex product $(a + \mathbf{i}b) \times (c + \mathbf{i}d)$, the term bc has the structure $(exp1, exp2)$, where $exp1$ and

*exp*2 refer to the first and the second operand, respectively. The following pruning rules are based on the structures of expressions. Due to these pruning rules, each expression is made of the terms of the same structure and the structure of any term can be taken as the structure of the entire expression.

4. +, − operations are allowed only on operands of the same structure.
5. × is allowed only on operands of different structures.

For example, in the search tree for computing the complex product, $(a - b) \times (a + b)$ is not allowed whereas $(a + b) \times (c + d)$ is allowed.

7 THE PARALLEL PROGRAM

Besides the speed gained by using the pruning rules, the search may be done in parallel to achieve further speed. The parallel approach is simply that of master-slave load allocation. In the serial version, a depth-first search of the tree is done until a path containing goal expressions is found. In the parallel program, several processors can be used with each processor exploring a subtree of the entire tree. The master-slave approach is inherently suited to multiple instruction multiple data (MIMD) computers. The primary obstacle to single instruction multiple data (SIMD) computers is the large number of branching tests from the pruning rules resulting in disparate control flow [3].

For the MIMD approach, one processor is used as the master processor delegating work (subtrees) to the other processors. The master processor does a depth-first search of the entire tree. However, on reaching a specified number of levels, it delegates the underlying subtree to an idle processor instead of exploring it. The master processor then backtracks and continues the search. Each of the slave processors performs a depth-first search on the subtrees assigned to it and reports the solution to the master, if found. To avoid waiting for work, each slave processor is given its next subtree while it is computing the current one. With this, all the slave processors are busy most of the time. On 256 nodes, efficiencies of 99.8% are typical.

A key parameter in the parallel program is the number of levels the master explores before distributing the subtrees. If this is too small, there might not be enough subtrees to allocate to all the processors. Also, there is greater danger of load imbalance with fewer subtrees. If this is too large, the master processor needs to do most of the work and the slaves remain idle. The choice depends on the size of the problem and the number of processors used. An appropriate value can easily be found by experimentation. For problems like complex product or matrix product on a system of 16 to 1024 nodes, exploring three levels on the master processor seems to be a good choice.

Notice that a superlinear speedup is possible, if we stop as soon as a solution is found. Because the tree is searched in parallel by many processors, there is a good possibility that some processor might get "lucky" and find the solution [19]. We have found that this is indeed the case, most of the time.

8 EXAMPLES OF USE

8.1 Integer Powers of a Number

An example of an algorithm optimization discussed in Knuth [10] is that of raising a number to an integer power by repeated multiplications. By saving and reusing partial products, the number of multiplications to compute a^n is usually much less than the $n - 1$ multiplications that would be done by a simple loop. for example, a^{15} can be computed with five multiplications using $a^2 = a \times a$, $a^4 = a^2 \times a^2$, $a^5 = a^4 \times a$, $a^{10} = a^5 \times a^5$, $a^{15} = a^{10} \times a^5$.

Even this simple problem illustrates that minimizing operations results in tricks, i.e., methods that do not seem derivable by straightforward stepwise thinking. A straightforward method might be to express the exponent as a binary number, and compute products corresponding to every "1" in that number. For example, $a^{15} = a^{1+2+4+8}$, so $a^2 = a \times a$, $a^4 = a^2 \times a^2$, $a^8 = a^4 \times a^4$, $a^{12} = a^8 \times a^4$, $a^{14} = a^{12} \times a^2$, $a^{15} = a^{14} \times a$. Because this method uses six multiplications, it is inferior to the "clever" method of more subtle derivation.

A further complication is that divide operations can sometimes reduce the work to get to a power, depending on the relative cost of multiplications and divides (and on the cost of handling divisions by zero). For instance, a^{31} can be computed by repeated squaring to get a^{32}, and then dividing by a to get a^{31}. The five multiplications and one division required will be faster than the seven operations needed using multiplications alone, if there is no cost for the $a = 0$ exception and division takes less time than two multiplications.

The optimization program of Section 4 easily generates optimal algorithms for the first powers of a number up to an exponent of 100. We simplified the program to consider only one initial input, a, and used the mathematical equivalence of the ring formed by multiplication and exponentiation to the ring formed by addition and multiplication. That is, the goal expression of $15a$ using only addition and subtraction is equivalent to a goal expression of a^{15} using only multiplication and division.

To automate the discovery of minimal operation counts, the algorithm was surrounded by the following control structure, where $m(n)$ is the number of operations to find a^n:

```
n = 2.
While n ≤ nmax
    mtest = ⌈log2 n⌉.
    While Search(mtest) fails
        Increment mtest.
    End While
    Record method with mtest operations.
    Increment n.
End While
```

Only conservative pruning rules involving adds were applied. The aggressive pruning rules are either irrelevant or inappropriate, since we wish goal expressions of the form na, where n is an integer. For extra speed, the *Search* program was simplified.

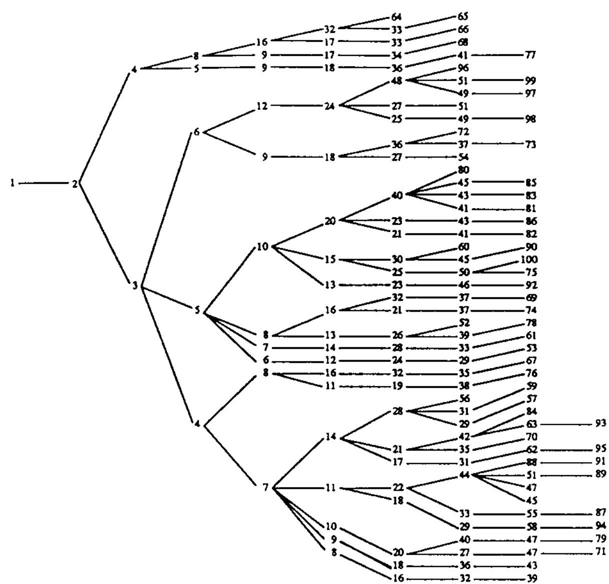


FIGURE 8 Power tree with multiplications.

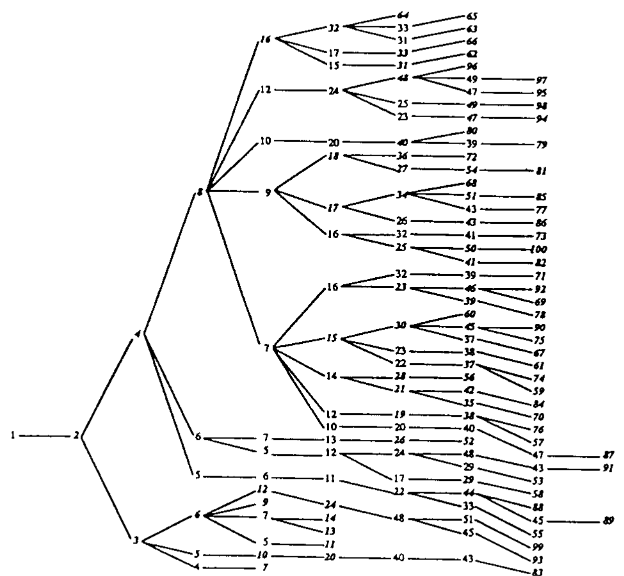


FIGURE 9 Power tree with divisions weighed the same as multiplications.

An nCUBE 2 with 64 processors took 1.5 minutes to find all of the optimal methods tabled in Knuth, assuming no divisions. Divisions were then introduced with various weightings, because division takes longer than multiplication on most 1992 computers, by amounts that vary from computer to computer. The optimal sequence of operations with no divisions is shown in Figure 8 and with a division weighted the same as a multiplication is shown in Figure 9. The sequence of optimal operations when a division is weighted as two or more multiplications is the same as that with no divisions, for integer powers less than or equal to 100.

Note that division is much less help than one might think. Given the high cost of division on many machines, plus the cost of exception handling when $a = 0$, the use of multiplications alone appears very attractive for low integer powers. For integer powers higher than 200 or so, other methods such as those given in [6] become advantageous.

8.2 Complex Product

We have already mentioned the trick for computing the product of complex numbers $(a + ib)$ and $(c + id)$ with only three multiplications. The search program found 12 different ways of doing this, 2 of which are given below:

$$m_1 = c \times (a - b) \quad m_1 = (a - b) \times (c + d)$$
$$m_2 = d \times (a + b) \quad m_2 = b \times c$$
$$m_3 = b \times (c - d) \quad m_3 = a \times d$$
$$u = m_1 + m_3 \quad u = m_1 + m_2 - m_3$$
$$v = m_2 + m_3 \quad v = m_2 + m_3$$

An exhaustive search with a budget of three multiplications and five adds took 21 h and 10 min on a 256-processor nCUBE 2. This represents approximately 10^{14} integer operations. Reduction of the budget to four adds revealed no way to compute the goal expressions $ac - bd$ and $ad + bc$, providing a computational "proof" of the nonexistence of a method with fewer adds.

Aggressive pruning rules can be used to reduce the search time: Expressions involving ab or cd are disallowed, as are expressions involving sums $\pm a \pm c$, $\pm a \pm d$, $\pm b \pm c$, or $\pm b \pm d$ (aggressive pruning rule 4). Nonhomogeneous expressions, expressions with degree higher than 2 (highest degree among goal expressions), and expressions having terms with coefficients other than ± 1 (see Section 6) are disallowed. Although we have no proof that optimal methods cannot use such steps, we empirically observed the efficacy of these pruning rules and make use of them in related problems such as cross product and matrix-matrix product. For the complex product, a search with the aggressive pruning rules took only 0.18 s on a 64-processor nCUBE, yet found all 12 methods. Aggressive pruning rules destroy proof of optimality, but speed the discovery of new methods. Without aggressive pruning rules, the method that forms the alternative title of this article would probably not have been tractable with current high-speed computers.

8.3 Cross Product

Among many applications, computer graphics uses the cross product of three-vectors extensively to determine intersections of rays and polygons or to create plane normals. The conventional algorithm is as follows:

Given three vectors $a_1\vec{i} + a_2\vec{j} + a_3\vec{k}$ and $b_1\vec{i} + b_2\vec{j} + b_3\vec{k}$, find

$$a_2 \times b_3 - a_3 \times b_2,$$
$$a_3 \times b_1 - a_1 \times b_3,$$
$$a_1 \times b_2 - a_2 \times b_1$$

as the three goal expressions.

While working on a synthetic scene generation program, we wondered if there might be a way to compute a cross product in less than six multiplications, at the cost of extra adds. The Sandia nCUBE 2, with 1024 total processors, was invaluable for this search; we used 256 of its processors for a period of 5 h and 40 min and discovered the method with the first set of stems distributed by the master. A search that went through all the stems would have taken a month, even with aggressive pruning rules. The method is shown at the end of Section 1. We believe this is the first publication of the method.

9 FUTURE APPLICATIONS

9.1 Strassen Products

The Strassen algorithm for 2×2 matrix products [20] was cited in the Introduction. Applied recursively to matrices of size N, it reduces the complexity of matrix-matrix products from order N^3 to order $N^{\log_2 7}$. The lack of an obvious pattern in the algorithm was the original motive for the work described in this article. We intended to generate the Strassen method by exhaustive search, and then attempt to find simlar tricks for 3×3 and 4×4 matrix products. For 3×3 matrix products, a method using only 23 multiplications has been found [13], presumably by hand. With optimal methods for 2×2, 3×3, and 4×4, we hoped to find a pattern that would reduce the exponent for matrix products without requiring very large values of N, the matrix size.

The budget for a Strassen product, using the Winograd improvement, is 22 operations (7 multiplications and 15 additions). Because the search grows exponentially in the number of operations in the budget, the search is significantly harder than those described in previous sections (Table 1).

Based on the number of "stems" generated by the parallel algorithm and the rate at which progress is made through that set of stems, we estimate that exhaustive search for the Strassen matrix product on the 1024-processor nCUBE 2, even with aggressive pruning rules, would take many centuries. We continue to seek exploitable symmetries and search orderings that will result in discovery of methods, if not search of the entire space.

Table 1. Minimum Operation Budgets Required for Various Tasks

Task to Optimize	Operation Budget
$a^2 - b^2$	3
Conventional complex product	6
Three-multiply complex product	8
Conventional cross product	9
Conventional matrix product	12
Five-multiply cross product	13
Strassen matrix product	22

9.2 Buneman-Type Methods Using Extra Identities

The conventional method for rotating a two-vector (x, y) by θ radians is to multiply it by a matrix of the form

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

to obtain (x', y'). This requires four multiplications and two additions. This operation is critical, e.g., in the fast fourier transform (FFT) algorithm. If the rotation is to be done for many two-vectors, as it is for the FFT, one regards the $\cos\theta$ and $\sin\theta$ values as constants instead of as input variables. Therefore, it is legitimate to consider precomputing alternative constants based on θ, but not consider the effort to do so in the operation count. Buneman [5] discovered a trigonometric identity that accomplishes a two-vector rotation with three multiplications and three additions, assuming zero cost for computing $\tan(\theta/2)$:

$$t = y + x \times \tan(\theta/2)$$
$$x' = x - t \times \sin\theta$$
$$y' = x \times \tan(\theta/2) + t$$

A similar approach exists for reducing the number of multiplications for computing a fourth-degree polynomial [18] from 4 to 3, where precomputation of constants based on the polynomial coefficients can be amortized over many evaluations of the polynomial for various arguments.

The *Search* program described in this article does not currently have any way of discovering this type of trick. We do not yet see a brute force approach likely to generate such tricks automatically.

9.3 Massively Parallel Compiler Optimization

Current commercially available MPP systems either compile source programs on a single node of the ensemble or use a front-end computer that has a traditional serial architecture. Small-scale parallelism for the phases of compilation can be accomplished via pipelining, but that approach does not scale to thousands of processors.

The *Search* method described here suggests a strategy for a way of using large numbers of processors during compilation. A small number of processors can do conventional compilation using pipeline parallelism, but basic blocks that look amenable to optimization can be farmed out to the rest of the ensemble and run through the *Search* process. When conventional compilation is completed, the processors that were given sections to optimize can be polled for the best optimization found so far, and the optimizations can be collected as a final pass. The user might supply a time constraint on compilation, so the search for reduced operation counts can be limited by the user's patience instead of the completion of conventional compilation. Here, "operation" includes any instruction in the target computer's repertoire, not just arithmetic operations. Also, overlap of instructions and the number of clock cycles for each instruction would have to be used in the cost metric instead of the simplistic operation counts used in Search. It might make more sense to budget clock cycles than operations or instructions for MPP compiler optimization. This is fertile ground for future research.

As a test of the massively parallel compiler optimization concept, we did the following experiment: Given the C expressions

$$f = a*a*a + a*a*b + a*b*b + b*b*b \text{ and}$$
$$g = a*a*a + 3*a*a*b + 2*a*b*b + b*b*b,$$

we used the *Search* program to optimize the calculation of f and g. Then, we tried various C compilers with optimizations enabled.

Unoptimized compilation gave a method to compute f using eight multiplies and three adds. The *Search* program found that f could be computed with three multiplies and two adds in 0.18 s on a 16-processor nCUBE. The Shortcut found by the *Search* program is $(a + b)(a^2 + b^2)$. The SUN (Sun Release 4.1) and DEC (Ultrix) compilers were only able to reduce the op-

eration count to seven multiplies and three adds. The Gnu compiler managed six multiplies and three adds.

A naive computation of g requires 10 multiplications and 3 adds, as found by unoptimized compilation. The SUN and DEC compilers could not find any improvements. The Gnu compiler found a method with nine multiplies and four adds. The optimum method ($(a + b)^3 - ab^2$) taking 4 multiplications and 2 adds is found by the *Search* program in 0.36 s on a 64-processor nCUBE.

To test our approach on expressions that appear in real production codes, we have chosen the following subroutine by Sisira Weeratunga of NASA Ames Research Center from the application benchmarks given in the NAS parallel benchmarks [4].

It appears that even simple algebraic expressions can be improved by a factor of 2 over current compiler technology using the *Search* approach.

10 LIMITATIONS OF USE

10.1 Numerical Stability

The *Search* program can be used to derive methods for computing expressions with minimum number of operations but such a method is not guaranteed to be numerically stable. Thus, analysis of the numerical stability of the derived algorithm is necessary before advocating its use.

Numerical stability of algorithms for real and complex matrix multiplication is discussed in [7, 8, 15]. Miller [15] analyzes the tradeoff between computational complexity and numerical stability. He defines the notion of *Strong stability* and *Brent stability*, which is a much weaker requirement with Strong stability implying Brent stability. For a thorough discussion of these notions, see [15]. It is shown that any algorithm for matrix multiplication

```
subroutine setiv
do k = 2, nz-1
   0zeta = (dfloat(k-1))/(nz-1)
   do j = 2, ny-1
      eta = (dfloat(j-1))/(ny-1)
      do i = 2, nx-1
         xi = (dfloat(i-1))/(nx-1)
         do m = 1, 5
               :
            u(m,i,j,k) = pxi + peta + pzeta - pxi * peta - peta * pzeta
                         - pzeta * pxi + pxi * peta * pzeta
         end do
      end do
   end do
end do
return
end
```

We tried to optimize the core of the computation given by the statement inside the loops, which, after a convenient renaming of the variables, is

$$a + b + c - ab - bc - ca + abc.$$

Table 2 shows the number of operations used by various compilers in computing this expression. An optimal way of computing this expression, as found by the *Search* program in 24.6 s on 32 processors, is

$$c(ab - a - b) - ((ab - a - b) - c).$$

Table 2. Number of Operations Used by Various Compilers and the *Search* Program in Computing $a + b + c - ab - bc - ea + abc$

Compiler	Unoptimized	Optimized
SUN (V 4.1)	5 muls, 6 adds	4 muls, 6 adds
Ultrix (V 4.2a)	4 muls, 6 adds	4 muls, 6 adds
Gnu (V 1.36)	5 muls, 6 adds	4 muls, 6 adds
Search	2 muls, 4 adds	

that possesses Strong stability should have at least n^3 multiplications and that the Strassen's method possesses Brent stability.

Applying a similar analysis to the new Croos Product method discovered by the *Search* program, we found that six multiplications are necessary for Strong stability and that the method discovered by the *Search* program possesses Brent stability. A similar result is true for the complex multiplication methods.

10.2 Use of Constants

Shortcuts for several expressions involve clever use of numerical constants. For example, the expression $abc - ab - ac - bc + a + b + c$ is equivalent to $(a - 1)(b - 1)(c - 1) + 1$, requiring only two multiplications and four adds. $a^4 - 8a^3 + 24a^2 - 32a + 16$ can be computed as $(a - 2)^4$ using only two multiplications and one add. Such clever methods involving the use of constants cannot be found by the *Search* program.

11 SUMMARY AND CONCLUSIONS

With the speed of computers continuing to increase about 60% per year, it is prudent to examine some of the problems and solution methods traditionally thought of as "intractable." Contrary to the philosophy taught in most computer science programs, even programs of combinatorially explosive complexity can yield interesting results for small problems. The computer-aided discovery of a cross product with five multiplications is an example of the kind of problem we can now pose to the fastest machines available.

In 1992, computers were capable of about 10^{11} integer operations per second, so a run within the limits of human patience might involve 10^{17} operations. At the current rate of performance improvement, computers will eventually be fast enough to "discover" the Strassen product trick in a 2 week run. As we find better rules for pruning the search tree, we might well move this date up by many years, and be able to attempt larger problems with the MPP computers of the future.

ACKNOWLEDGMENTS

We thank Dr. G. M. Prabhu for many suggestions during the development of this article and for verifying the proofs of the validity of the conservative pruning rules. We are also indebted to Sandia National Laboratories for generous amounts of time on their 1024-processor nCUBE 2 computer, on which many of our results were obtained. This work is supported by the Applied Mathematical Sciences Program of the Ames Laboratory-USDOE under contract No. W-7405-ENG-82. The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. W-7405-ENG-82. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so for U.S. Government purposes.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

[2] S. Aluru and J. Gustafson, "A massively parallel optimizer for expression evaluation," in *International Conference on Supercomputing,* 1993, p. 97–106.

[3] S. Aluru and J. Gustafson, "Subtle issues of SIMD tree search," *Proc. Parallel Computing,* 1993.

[4] D. Bailey et al., "The NAS parallel benchmarks," NASA Ames Research Center, Ames, IA, Tech. Rep. RNR-91-002, Jan. 1991.

[5] O. Buneman, "Inversion of the Helmholtz (or Laplace-Poisson) operator for slab geometry," *J. Computational Phys.,* vol. 12, pp. 124–130, 1973.

[6] W. J. Cody and W. Waite, *Software Manual for the Elementary Functions.* Englewood Cliffs, NJ: Prentice-Hall, 1980.

[7] N. J. Higham, "Exploiting fast matrix multiplication within the level 3 BLAS," ACM Trans. Math. Soft., vol. 16, pp. 352–368, 1990.

[8] N. J. Higham, "Stability of a method for multiplying complex matrices with three real matrix multiplications," *SIAM J. Matrix Anal. Appl.,* vol. 13, pp. 681–687, 1992.

[9] J. E. Hopcroft and L. R. Kerr, "On minimizing the number of multiplications necessary for matrix multiplication," *SIAM J. Appl. Math.,* vol. 20, pp. 30–36, 1971.

[10] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* 2nd ed. Reading, MA: Addison-Wesley, 1981.

[11] L. Kronsjö, *Algorithms: Their Complexity and Efficiency,* 2nd ed. New York: John Wiley & Sons, 1987.

[12] D. W. Krumme and D. H. Ackley, "A practical

method for code generation based on exhaustive search," in *Proceedings of the ACM SIGPLAN'82 Symposium on Compiler Construction,* 1982, pp. 185–196.

[13] J. D. Laderman, "A non-commutative algorithm for multiplying 3 × 3 matrices using 23 multiplications, *Bull. Am. Math. Soc.,* vol. 82, pp. 126–128, 1976.

[14] H. Massalin, "Superoptimizer-A look at the smallest program," *ASPLOS II,* pp. 122–126, 1987.

[15] W. Miller, "Computational complexity and numerical stability," *SIAM J. Comput.*, vol. 4, pp. 97–107, 1975.

[16] V. Pan, "Strassen algorithm is not optimal. Trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix multiplication, in *Proceedings of the 19th Annual Symposium on the Foundations of Computer Science,* Ann Arbor, MI, 1978, pp. 166–176.

[17] V. Pan, "How can we speed up matrix multiplication?" *SIAM Rev.*, vol. 26, pp. 393–415, 1984.

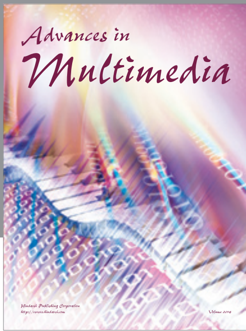
[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge, MA: Cambridge University Press, 1990.

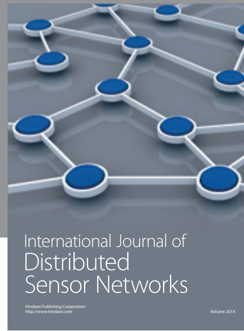
[19] V. N. Rao and V. Kumar, "Superlinear speedup in ordered depth-first search," in *Proceedings of the 6th Distributed Memory Computing Conference* (DMCC6), 1991.

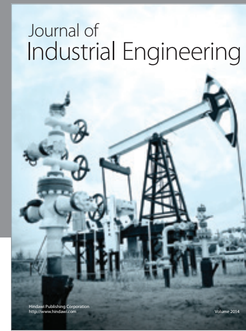
[20] V. Strassen, "Gaussian elimination is not optimal," *Numer. Math.*, vol. 13, pp. 354–356, 1969.

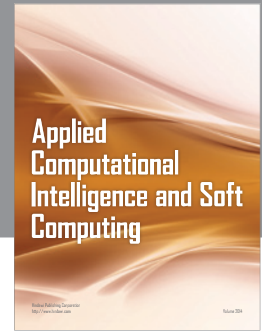
[21] S. Winograd, "On multiplication of 2 × 2 matrices," *Linear Alg. Appl.*, vol. 4, pp. 381–388, 1971.

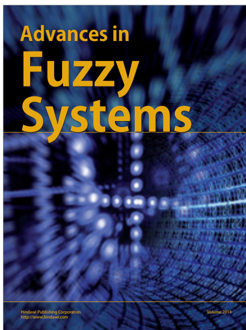
[22] G. Yuval, "A simple proof of Strassen's result," *Info. Proc. Let.*, vol. 7, pp. 285–286, 1978.

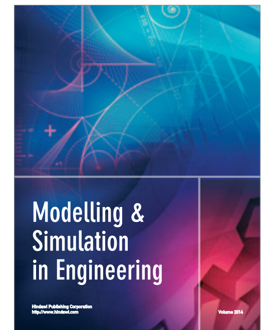












Hindawi

Submit your manuscripts at
http://www.hindawi.com