Efficient Construction of Balanced Binary Trees

D. J. Evans and F. Abdollahzadeh
Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire, UK

An improved algorithm for balancing arithmetic expression trees is presented and its efficiency considered. When binary trees are constructed to represent arithmetic expressions and vacant nodes occur then by making use of the vacancy in the tree we are able to insert a smaller tree of lower height resulting in a balanced binary tree (b.b.t.) which is more efficient than those given by existing methods.

INTRODUCTION

In this note the construction of a b.b.t. for arithmetic expressions which is applicable in a parallel processing environment is presented. The problem of parallel evaluation of arithmetic expressions has been investigated by several authors, and various methods have been proposed. These methods determine which part of the expressions are most suitable for execution in parallel. Since any operation that appears at the same level in a tree representation of an expression, may be executed in parallel, an attempt to construct a balanced binary tree of minimum height is the ultimate goal of many of these methods.

However, in the construction of the tree for a general expression the likelihood of vacancies occurs and a strategy of using these vacancies in a most adaptable and flexible manner is necessary in order to achieve a binary tree representation of an arithmetic expression which is of minimum height wherever possible.

The algorithm is compared with those of Refs 1 and 2 and the results show that not only is the level of the tree reduced but also the method of the construction of the tree and its execution are more efficient.

DEFINITION AND CONCEPTS

1. An arithmetic expression is called a sum (or product) term, if it is a simple variable, or the final operation is '+ ' (or '*').

2. If T is a subexpression such that the root node is ' + ' (or '*') and T1 and T2 are left and right-subtrees respectively, then $T = T_1 + T_2 (T = T_1 \cdot T_2)$ is the algebraic notation for $T$.

3. The level of a tree is denoted by $h(T)$ which is an integer.

4. Vacancies of an execution tree T, denoted as $V_T$ is a set obtained as follows:

   (a) if T is a leaf of the tree then $V_T = \emptyset$ (the empty set)

   (b) if $T = T_1 + T_2$, two cases can be considered,

      (i) $|h(T_1) - h(T_2)| \geq 1$, then $V_T = \{h(T) - 1, h(T) - k, \ldots, h(T) - n > \min (h(T_1), h(T_2))\}$

      and all of the vacancies belong to one of the subtrees.

      (ii) $|h(T_1) - h(T_2)| = 0$. This means that the sub-trees have the same height and $h(T_1) = h(T_2) = h(T) - 1$.

5. If $T_1$ and $T_2$ are execution trees, then $T_1$ accommodates $T_2$ if and only if $h(T_1) + h(T_2) = h(T)$, i.e. there is at least a suitable vacancy in $T_1$ into which $T_2$ can be inserted, in other words $h(T_1) > h(T_2)$.

These definitions are clarified by the following examples.

If $T = T_1 + T_2$, to find $h(T)$, there are three cases to be considered:

   (1) $h(T_1) > h(T_2)$, there may be a vacancy in $T_1$ which can accommodate $T_2$.

   (2) $h(T_1) < h(T_2)$, there may be a vacancy in $T_2$ which can accommodate $T_1$.

   (3) $h(T_1) = h(T_2)$, there is not a suitable vacancy in either of them to accommodate the other.

THE FORMATION OF A BALANCED BINARY TREE

To construct a tree representing an arithmetic expression, three cases must be considered for an expression of the form $T_{1_{op}} T_{2_{op}} \cdots T_{n_{op}}$.

   (a) where $T_i, 1 \leq i \leq n$ are simple operands or other subexpressions and $op$ is a non-associative operator.

   (b) where $T_i, 1 \leq i \leq n$ are simple variables (trees of $h(T_i) = 1, 1 \leq i \leq n$ and $op$ is an associative operator.

   (c) where $T_i$ can be either a simple operand or another subexpression (a tree of $h(T_i) \geq 1$) and $op$ is an associative operator.

An expression can have any combination of the above cases.

In case (a), even if there exists a vacancy of the correct size in one of the trees, it is not possible to insert any other subexpressions into it. Therefore, a tree, say $T = T_{1_{op}} T_{2_{op}} \cdots T_{n_{op}}$, with $h(T) = h(\max(h(T_1), h(T_2), \ldots, h(T_n))) + n - 1$ is constructed. We should explain here that any available vacancies, or in other words wasted processors, are available for use in other computations within the multiprocessor at the same time that this tree is being computed.

We will outline a new algorithm which makes a balanced binary tree of minimum height for case (b). Then, we introduce a further new algorithm, to construct a tree for case (c).

To construct a tree for case (b), $T_1$ and $T_2$ can be joined by forming a new tree, say AD1, whose left and right subtrees are $T_1$ and $T_2$, similarly $T_3$ can be joined to AD1 by forming another tree, AD2, whose right subtree is
AD1, and left subtree is $T_3$. Then $T_4$ can be inserted into the right subtree of AD2 (without increasing the height). Thus, the right subtree of AD2 is $T_{3 \, op} \, T_4$.

Similarly $T_5$ can be added by forming another tree, AD3 with AD2 and $T_4$ as its left and right subtrees. This process is continued either by inserting each $T_i$, where $i \leq n$, into the constructed tree or by forming another tree whose left subtree is the constructed one of $T_1$, $T_2$, ..., $T_{i-1}$ and right subtree is $T_i$. Then, the constructed tree is a balanced binary tree of height $\log_2 n$.

For case (c) this one-pass algorithm constructs a tree of minimum height for any arithmetic expression of the form $(a_0, T_1, T_2, \ldots, T_n)$, where $a_0$ is an associative operator and each $T_i$, $1 \leq i \leq n$ is a simple operand (a tree of height one) or another subexpression.

To make a tree of minimum height for the expression, the following cases will be considered:

1. If $T_1, T_2, \ldots, T_i, i < n$ are simple operands then the algorithm for case (b) will be applied to make a tree of height $\log_2 i$. The constructed tree is being used instead of $T_i$.
2. If the two successive operands, say $T_i$ and $T_{i+1}$, are other subexpressions of different height, then if $h(T_i) > h(T_{i+1})$, the operator of the root node of $T_i$ is

---

**Figure 1.** Execution tree for $E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + b_1 + b_2 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + b_{10}$ by the new algorithm.

---

**Figure 2.** Execution tree for $E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + b_1 + b_2 + (c_1 * c_2 + c_3) * c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + b_{10}$ by Williams's algorithm.

---

**Figure 3.** Execution tree for $E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + b_1 + b_2 + (c_1 * c_2 + c_3) * c_4 + c_5 + c_6 + c_7 + c_8 + c_9 + b_{10}$ by Ashoke's algorithm when $\theta = 1$, $\theta = 2.5$ and $\theta = 3.2$.
the same as \( h \) and \( h(\text{left subtree of } T_i) > h(\text{right subtree of } T_i) \), then if there exists a suitable vacancy on the right subtree of \( T_i \), \( T_{i+1} \) can be inserted into it. In this case \( T_i \) is the resulting tree.

(3) If \( h(T_i) < h(T_{i+1}) \) and all the above conditions for the right subtree of \( T_i \) hold for the left subtree of \( T_{i+1} \), then \( T_i \) can be inserted into the left subtree of \( T_{i+1} \). In this case \( T_{i+1} \) is the resulting tree.

(4) If \( h(T_i) = h(T_{i+1}) \) or the conditions described in (1) and (2), do not exist. Then, the resulting tree of \( h(T_i) + 1 \) is a tree whose left and right subtrees are \( T_i \) and \( T_{i+1} \), respectively.

This process will be continued until the stack becomes empty, i.e. all the operands have been used.

Example

Let

\[
E = a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + b_9 + \uparrow \\
\quad \quad \quad \quad \quad b_9 \cdot c_1 + b_9 \\
\quad \quad \quad \quad \quad (b_9 + b_9) \cdot b_9 + a_9 + b_1 + b_2 + (c_9 \cdot c_9 + c_9) \uparrow c_2 + \\
\quad \quad \quad \quad \quad b_3
\]

then the execution tree for this expression is shown in Fig. 1. The height of the tree is \( h(T) = 7 \).

### COMPARISON OF THE ALGORITHMS

From Ref. 2 it can be observed that the algorithm described there (Ashoke’s) makes use of the commutativity of subexpressions, whereas Evans and Williams’s algorithm\(^1\) and the method proposed here do not reorder any subexpression or single variable of the expression. Figs 1–3 show the different execution trees for the arithmetic expression in the given example by the present, Evans and Williams’s and Ashoke’s algorithms, respectively, with the corresponding trees being of heights 7, 8 and 8.

Ashore’s algorithm uses the distribution law to reduce the height of the tree (assuming different execution times for various operators), whereas Abdollahazadeh\(^3\) shows that distribution may or may not decrease the tree height and may increase the number of processors even if it reduces the height.

Hence, in the speeding up of numerical computations, the application of the distribution law is still debatable. If a certain expression has to be evaluated for many different values, it might be worthwhile to apply distribution to reduce the height of the execution tree. Since each machine takes a different execution time for each operation, Ashoke’s algorithm is machine dependent.

The new algorithm presented here attempts to reduce the height of the execution tree by using ‘vacancies’ in all the subexpressions. Comparing Figs 1, 2 and 3 we see that the new algorithm is better because it might give smaller trees. Also, if there were more subtrees with height less than 6, they could be inserted into the right subtree of \( T \) in Fig. 1 without increasing the height of the tree, but in Fig. 2 the height of the tree would be increased. Williams\(^4\) claims that ‘this situation is in line with the tree being formed systematically from components’. This is not the case, because, in the algorithm, insertions are always at the next available position in the tree, i.e. the algorithm takes the subexpressions successively from its right hand side, so any suitable positions available earlier in the tree are not accessible (see Fig. 2). The new algorithm takes subexpressions always from its left hand side for insertion in the tree and whenever any vacancy is available in the tree, the new subexpression is accommodated into it.

The time to construct the trees for 9 simple arithmetic expressions by the new algorithm is four time units less than the time by Evans and Williams.\(^1\) Also, the time to execute the trees for fifty arithmetic expressions is three time units less than the time by Evans and Williams. Therefore, the new algorithm may be more efficient in terms of time as well as height level.

### REFERENCES


Received August 1981