Research report 131

DYNAMIC EXPRESSION EVALUATION IS ONE OF A CLASS OF PROBLEMS WHICH ARE EFFICIENTLY SOLVABLE ON MESH-CONNECTED COMPUTERS

A M Gibbons

(RR131)

Let P be a recursively described and distributable problem of size n. We describe a general strategic approach which will often make an O(√n) parallel time solution of P possible on a 2-dimensional mesh-connected computer with O(n) PEs. Such a solution is time-optimal. It is likely that the class of such problems is large and will contain many that are non-trivial. We show that such a problem is dynamic expression evaluation. The methodology is easily extended to mesh-connected computers of arbitrary dimension to again obtain time-optimal solutions using O(n) processors.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

September 1988

Dynamic expression evaluation is one of a class of problems which are efficiently solvable on mesh-connected computers

A.M. Gibbons
Department of Computer Science,
University of Warwick, Coventry CV4 7AL, U.K.

Key Words: efficient parallel algorithms, mesh-connected computer, dynamic expression evaluation, list ranking problem, recognition of bracket and input driven languages, divide and conquer.

§1. Introduction.

Within the well-known P-RAM model of parallel computation, the class NC defines the class of efficiently solvable problems (see [3] for example). Using a polynomial number of processors, such problems can be solved in polylogarithmic parallel time. Within the constraints of current technology, it is not always possible to attain such time complexities because a lower bound for feasible parallel architectures (e.g. shared memory SIMD machines such as the mesh-connected computer) is naturally O(r), where r is the maximum path length within the architecture over which messages have to be passed between the co-operating processors. We show in §4 that the problem of dynamic expression evaluation belongs to a (probably large) class of problems (defined in §3) which are efficiently solvable on mesh-connected computers. Here, an *efficiently solvable problem* is a problem which is solvable in O(r) parallel time using O(n) processors. The mesh-connected computer as a model for parallel computation is well known [1,2,6,7,8,10]. Our presentation will be for two-dimensional mesh-connected computers (*MCC^2s*), where $r=O(\sqrt{n})$. Generalisation to arbitrary dimension is straightforward.

Let P, a problem of size n, be distributed across a MCC^2. The nodes of a graphical representation of P (the '*problem graph*') are mapped onto the processing elements (*PEs*) of a $\lceil\sqrt{n}\rceil\times\lceil\sqrt{n}\rceil$ mesh. Initially each PE stores at most one node of the problem graph. Along with such a node are addresses of PEs where nodes adjacent in the problem graph are stored (the problem graph is usually of constant-bounded degree). There may also be capacity to store a constant volume of additional information associated with a node. Now let P be recursively reducible to m similar problems, $1\leq m\leq b$, each of size ($\lceil n/b\rceil+c$) where $b\geq2$ and $c\geq0$ are integer constants. From the point of view of determining the time complexity T(n) for P, we generally need to solve a recurrence relation. The details of this recurrence, and therefore the explicit form of T(n), will depend upon precise details of the strategy employed within this framework. In §3, for any problem such as P, we emphasise certain strategic details that will often make an $O(\sqrt{n})$ parallel time implementation possible on a MCC^2 with O(n) PEs.

§2. An introductory example.

As a simple introductory example we take the problem of evaluating, at $x=h$, the general polynomial $p(x)$ of degree n, where:

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

For ease of presentation we assume here that $n=2^k-1$ for some integer k. We adopt a familiar mode of evaluation in which $p(x)$ is recursively described as follows:

$$p(x) = p'(x)+x^{(n+1)/2}p''(x)$$

where $p'(x)$ and $p''(x)$ are similar polynomials of degree $2^{k-1}-1$. The following provides an (equivalent) iterative evaluation of $p(x)$ at $x=h$.

```
1  $x \leftarrow h$
2  $d \leftarrow (n-1)/2$
3  **repeat until** $d=0$
          **begin**
4         **for** $0 \le i \le d$ **in parallel do** $a_i \leftarrow a_{2i}+x\ a_{2i+1}$
5         $x \leftarrow x^2$
6         $d \leftarrow (d-1)/2$
          **end**
```

Consider implementation on an MCC^2. We provide each PE of the $n^{1/2} \times n^{1/2}$ mesh with one of the constants a_i and with the values of n and h. Within the computation each processor repeatedly recomputes its a_i, x and d. When $d=0$ the result of the computation is found in the register storing a_0. Each recomputation of x and of d takes constant time. Each recomputation of a_i requires the values of a_{2i} and a_{2i+1}. These can be acquired from their associated PEs by successive applications of the *Random Access Read* procedure (RAR) of Nassimi and Sahni [6]. (We adopt the input convention that PE_i is associated with the parameter a_i). Each application of RAR takes $O(n^{1/2})$ parallel time. Since the number of repititions involved in the repeat statement is $O(\log n)$, we have described an $O(n^{1/2} \log n)$ implementation. We can however do better than this. Notice that each iteration of the repeat statement reduces the 'size' of the problem (that is, the number of processors that need to be active) by a factor of 1/2. It will be convenient to contrive reduction by a factor of 1/4. This is easily done by making each iteration contain two assignments of the kind indicated in line 4. Now take the PEs to be indexed according to *shuffled row major* order (see for example Nassimi and Sahni [6]) then after each iteration in which the problem size is reduced by a factor of 1/4, the active processing elements are constrained to occupy a square of the mesh which is 1/4 the original area. See figure 1. The affect of this in the ith iteration is to replace the cost $(O(n^{1/2}))$ of applying the RAR procedure by $O(n^{1/2}/2^{i-1})$. The complexity of the algorithm is now $O(n^{1/2}(1+1/2+1/4+.....))$ which is $O(n^{1/2})$ and which is time-optimal and an improvement by a factor of log n.

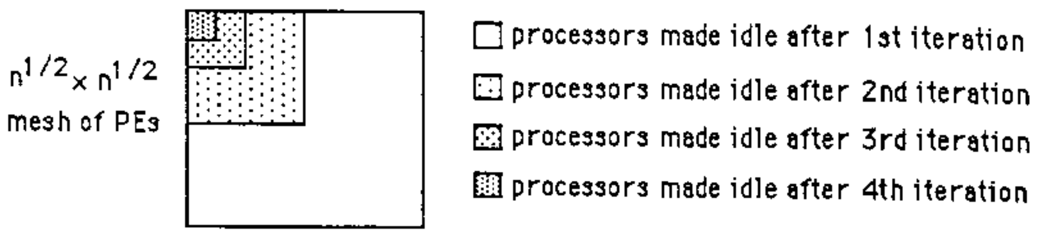
n^1/2 x n^1/2
mesh of PEs
□ processors made idle after 1st iteration
▨ processors made idle after 2nd iteration
▨ processors made idle after 3rd iteration
▦ processors made idle after 4th iteration

Figure 1

§3. A class of efficiently solvable problems

We show here that many problems of the general description given to P in §1 are likely, by observing certain strategic details, to be efficiently solvable on a MCC^2. It is these problems that form the class defined here.

The technique employed in the efficient implementation of the algorithm described in §2 relied essentially upon shrinking (by a constant factor) the space physically occupied by the problem during each iteration of the body of a **repeat** statement. If at an arbitrary time the problem size is s, the recurrence relation for the parallel computation time, T(s), is given by:

$$T(s)=T(s/4)+O(s^{1/2}), \qquad s>1, \qquad \text{(i)}$$
$$T(s)=0, \qquad s=1$$

As we saw $T(n)= O(n^{1/2})$. Within our example the compression of successively produced problems into smaller squares was made possible by indexing the processing elements according to *shuffled row major* order. For this problem we contrived that compression along with new problem creation took O(1) time. New problem creation and compression may in general be separate processes. In general (for similarly solvable problems in which $m=1$, $b\geq2$, $c=0$) we can perform the compression in $O(s^{1/2})$ parallel time after new problem creation using the techniques of [6]. Provided new problem creation also takes $O(s^{1/2})$ parallel time, as will often be the case, then (i) still holds and we have algorithms of optimal time complexity using n PEs.

Now consider the case in which $m>1$ ('divide and conquer'). It is trivial to observe that any problem of size s which at each level of recursion is divided into 4 (or less) similar problems of size at most s/4 will satisfy (i) provided that this division of the problem and the subsequent assignment of each created problem to its own quarter of the PE mesh can be achieved in $O(s^{1/2})$ parallel time. In §2 a specious method was used to concentrate the newly created problem. In the general case, the problem of assigning each of the created problems to their own quarter of the processing mesh can be solved by an application of the connected components algorithm of Nassimi and Sahni [7]. This is because the connected components of the problem graph correspond to the subproblems. Unfortunately this takes $O(s^{1/2} \log s)$ parallel time on a MCC^2 for graphs of arbitrary but fixed maximum degree Δ. There is an exception (see [7]) when $\Delta=2$, for then the complexity is $O(s^{1/2})$ and (i) still holds. However in general, the use of this connected components procedure will lead to a complexity of least $O(s^{1/2} \log s)$ which is no improvement over the case without 'compression'. Using the best known connected components algorithm brings no improvement unless $\Delta=2$ for which case we shortly give (in a more general setting) an example. Of course, individually, there may exist specious methods for efficient compression just as there was for the example of §2.

A modified shuffled row major indexing suitable for b=3

0	1	2	9	10	11	18	. . .
3	4	5	12	13	14	21	
6	7	8	15	16	17	. . .	
27	28	29	. . . .				
30	. . . .						
. . .							

Figure 2

A natural generalisation of the foregoing with m>1 (we have dealt with m=1) is to replace the factor of 1/4 in (i) by 1/b where $b\geq 2$ is a constant integer. Here, within each level of the recursion, a problem of size s is replaced with (upto) b similar problems in $O(s^{1/2})$ parallel time, each problem is of size at most s/b (without loss of generality we can take $s=b^j$ for some integer j). The initial difficulty with this is how to partition the processing element mesh systematically so that each recursively created problem occupies a square and such that overall n PEs are still sufficient for an $O(s^{1/2})$ time computation. Now no longer does the *shuffled row major* indexing based on binary considerations with an attendant recursive division of the processing element mesh into a 2×2 matrix serve us well. However a *modified shuffled row major* indexing suits our needs. What we require is an indexing such that the square area (of s PEs) is recursively divided into a $b\times b$ matrix, each sub-square of the mesh occupying an area of consecutively indexed s/b^2 PEs. Each such sub-square may then be occupied by one of the problems produced after a double application of the problem division procedure without an overall demand for more PEs. Figure 2 shows the indices of the first few processing elements for such an indexing with b=3.

We now consider the general case that each problem of size s is recursively replaced by at most b similar problems each of size at most $\lceil s/b\rceil+c$ where b and c are constant integers, $b\geq 2$ and $c\geq 1$. We motivate our considerations by describing a simple but nevertheless architypal example known as the *list ranking* problem.

Given a list of elements, the list ranking problem is to associate with each element i a parameter L(i) such that L(i) is the distance from i to the head of the list. The standard P-RAM technique that places the problem of list ranking in NC is that of recursive doubling on the list pointers. Figure 2 shows the technique for a list of seven elements. The ith element has an associated pointer P(i) which initially points to the next element of the list. With P(i) we associate L(i), which is the current distance along the list from the element i to the element which P(i) points to. At the outset of the computation the situation is then as illustrated at the top of figure 3. Here

L(i) labels P(i). The same figure then shows the P(i) and L(i) after successive iterations within the algorithm. In each iteration, for all i in parallel and provided each particular P(i) does not yet point to the head of the list, the processor associated with element i makes the

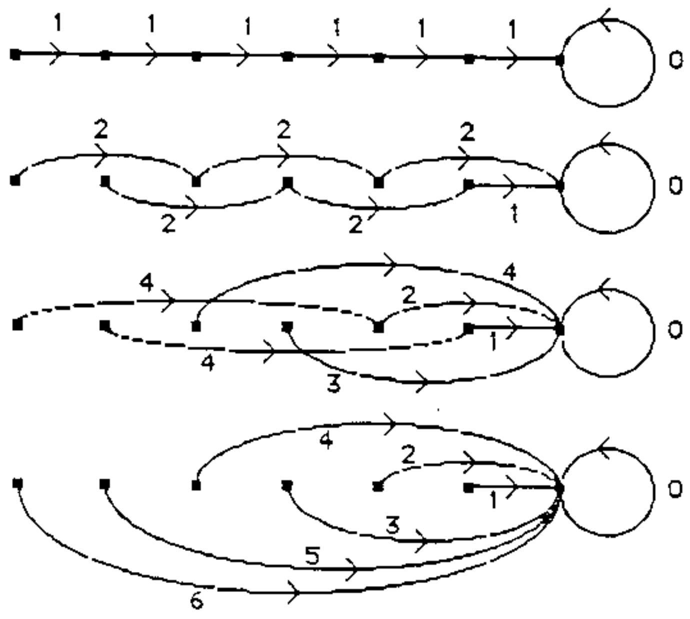

Figure 3

assignments $L(i) \leftarrow L(i)+L(P(i))$ and $P(i) \leftarrow P(P(i))$. After i iterations we have that P(k), unless it points to the head of the list, points 2^i elements along the list from i. Thus after $\lceil \log_2 n \rceil$ iterations all P(i) point to the head of the list. Moreover, R(i)=P(i) for all i. Thus on a P-RAM the list ranking problem is easily solved in O(log n) time with n processors (details of the implementation are obvious).

If j is at the head of the list, after each iteration within the algorithm described all the P(i) except P(j) form a number of directed paths terminating at j. After each doubling operation, the number of these paths is at most doubled. It is natural to utilise this in a recursive solution to the list ranking problem. Each of the problems of size s at one level of the recursion get divided into at most two problems of size at most $\lceil s/2 \rceil+1$. To obtain strictly disjoint problems we just have to copy the element at the head of the original list into each newly created list. Notice also that the problem graph will have $\Delta=2$. Here then is a simple example of the type of problem which has parameters b=2 and c=1. As for the problem in §2 we can easily contrive that b=4.

Let us return to the generalised form of the problem. At the outset we have a problem of size n. At the first level of recursion each of the (up to) b problems have size $(\lceil n/b \rceil+c) \leq (n/b+c+1)$, at the second level of size at most $n/b^2+(c+1)(1+1/b)$ and so on. In general, at the ith level of recursion each of the b^i problems has size bounded by ps(i):

$$ps(i)=n/b^i+(c+1)\sum_{j=0 \text{ to } i-1}(1/b)^j = n/b^i+k(1-1/b^i) \qquad \text{where } k = b(c+1)/(b-1) \qquad \text{(ii)}$$

Notice that below a certain level of the recursion, this estimate does not reduce the (integral) problem size. A minimum is reached when $(n\text{-}k)/b^i \leq 1/2$, at which point:

$$i \approx \log_b 2(n\text{-}k) \quad \text{and} \quad ps_{min} \approx \lceil k \rceil \qquad \text{(iii)}$$

This provides the value of i at which the recursion bottoms-out and at which the residual problem is solvable in constant time. Consider now implementation on a MCC^2. At the ith level of recursion, as for the case with c=0, we store each problem of size ps(i) over n/b^i PEs arranged in a sub-square of the mesh. In the case that $c \neq 0$ however, instead of a single node of the problem graph being stored at each PE, we now store up to k nodes of this graph at each PE. In other words the problem, of size $ps(i) \leq n/b^i + k$, is stored over n/b^i PEs and we note that it is an easy technical problem to store the additional $\leq k$ nodes evenly over these PEs. Now, within each application of the problem division procedure, each PE (in parallel with the other PEs) processes in sequential fashion the nodes of the problem graph that are stored at that PE. Since there are at most a constant number of nodes stored at each PE, the order of the time complexity of executing the problem division procedure will be the same as if a single node of the problem graph were stored at each PE. Notice the inconsequential technical difficulty that within the process of sequentially handling nodes of the problem graph, care has to be taken that the effect is the same as if they had been handled in parallel. This is easily achieved (at the expense of increasing the storage space required at each PE by at most a constant) by (in effect) keeping and not corrupting a copy of the problem graph output from the previous application of the problem division process whist constructing the output from the current application.

The foregoing paragraph justifies the claim that, using the same number of processors, the order of the parallel time complexity of a problem with the integer parameter c>0 is the same as for a similar problem with c=0. Thus, for example, our list ranking problem has an efficient solution because its equivalent problem with c=0 has a problem graph with $\Delta=2$.

§4. Dynamic expression evaluation.

Consider now the problem of dynamic expression evaluation. We shall see that it falls within our class of efficiently solvable problems on a MCC^2. Dynamic expression evaluation is the problem of evaluating an expression with no free preprocessing. This problem has been considered in terms of the P-RAM by Miller & Reif [5] and by Gibbons & Rytter [4]. The algorithm of [4] can be made to run on a P-RAM in O(log n) parallel time using O(n/log n) processors. We briefly outline a simple version here which would run in the same time on a P-RAM using O(n) processors. As we shall see, such a version of the algorithm has a MCC^2 implementation taking $O(n^{1/2})$ parallel time with O(n) processors.

The input to the algorithm is the expression tree. The first task is to rank the leaves of the tree from left to right before presentation to the algorithm proper. Such a preprocessed input is shown in figure 5(a). Within that figure each leaf has an associated integer (its rank) and the number shown in brackets is the value associated with that leaf. The ranking of the leaves is easily achieved by an application of the Euler tour technique of Tarjan and Vishkin ([9], also described in [3]). Within this technique we first construct the so-called traversal list of the expression tree (if we associate a processor with each of the nodes of the binary-tree of the expression, this takes constant time on a MCC^2). Each leaf appears exactly once on this list (other nodes appear three times each) also the leaves appear in the same order as their left to right ranking in the tree. We 'mark' the leaf elements and then, by a simple adaptation of the list ranking procedure, we rank the marked elements on the list to complete the pre-processing. All this takes $O(n^{1/2})$ parallel time with $O(n)$ processors on a MCC^2.

The algorithm now consists of repeatedly applying a so-called leaves-cutting operation, within each such operation the number of leaves in the tree is reduced by a factor of one half. Eventually the tree is reduced to a single node at which time the expression has been evaluated. A single leaves-cutting operation consists (as we shall see) of the parallel removal of some leaves of the tree. We therefore introduce the operation by first describing how a single leaf may be removed

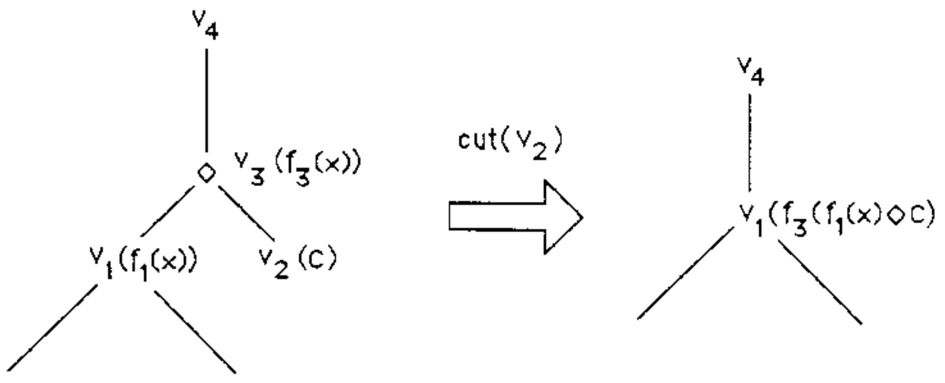

Figure 4

by a local reconstruction of the tree. Such a local reconstruction is illustrated in fig.4. This figure shows the removal the single leaf v_2. At each internal node v_i of the tree we store an associated function, $f_i(x)$, which (when evaluated at x=[value of the sub-expression associated with the sub-tree rooted at v_i]) represents the value to be passed to father(v_i) in the evaluation of the expression. In the course of the computation, it is an invariant (we only consider the operations +,-,* and /) that each $f_i(x)$ has the general form $f_i(x)= (a_ix+b_i)/(d_ix+e_i)$ where a_i, b_i, d_i and e_i are numerical constants which are recomputed whenever a leaf (which is a brother of v_i) is cut. Thus we can represent each $f_i(x)$ by storing four numbers. Initially, for all i, $a_i=e_i=1$ and $b_i=d_i=0$. Thereafter, if we have that $f_1(x)= (a_1x+b_1)/(d_1x+e_1)$ and $f_3(x)= (a_3x+b_3)/(d_3x+e_3)$ and if a single cutting of a leaf requires (see figure 4) that we recompute the constants defining $f_1(x)=f_3(f_1(x)\diamond c)$, then this is done in constant time as follows:

$a_1=((a_3 \diamond c)\ a_1+b_3d_1)$ $\qquad$ $b_1=((a_3 \diamond c)\ a_1+b_3d_1)$

$d_1=((a_3 \diamond c)\ a_1+b_3d_1)$ $\qquad$ $e_1=((a_3 \diamond c)\ a_1+b_3d_1)$

where $\diamond$ is the operation at father(v_2). When the tree is locally reconstructed (corresponding to the cutting of leaf v_2 of figure 4) by the movement of a (constant) number of father-son pointers, these recomputed constants ensure that the value represented by the new tree is the

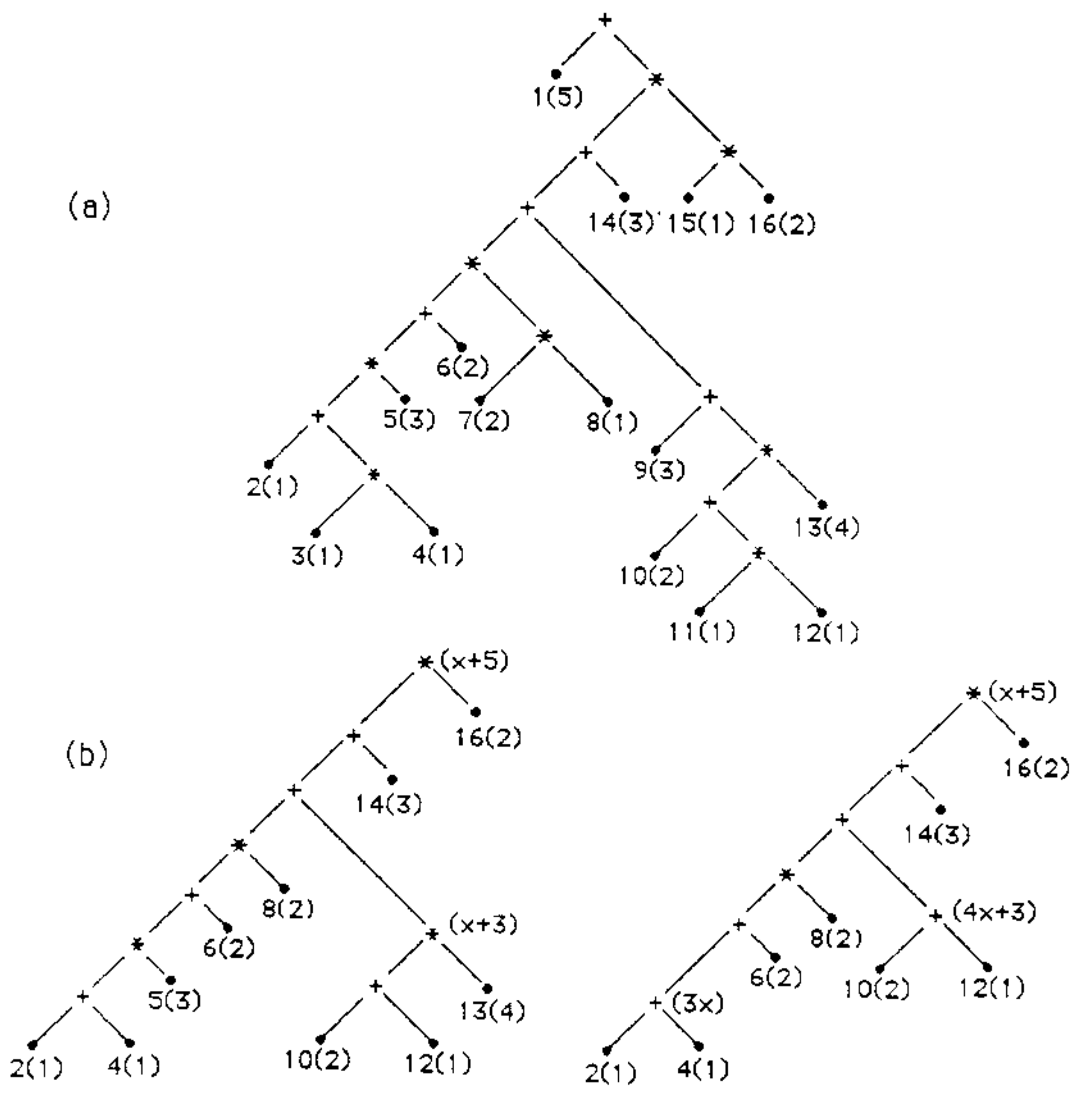

Figure 5

same as that represented by the original. Having described how a single leaf may be cut, we now describe the operation of parallel leaves-cutting. Referring to figure 4, if we define involved(v_2)=$\{v_1,v_2,v_3\}$ then we say that two operations cut(v) and cut(v') are independent if involved(v) $\cap$ involved(v)=$\emptyset$. Any set of pairwise independent cut operations can be performed in parallel. A sufficient condition that cut(v) and cut(v') are independent is that both v and v' are non-consecutive leaves and that both right or both left sons. We therefore define the operation of **parallel leaves-cutting** to consist of the three steps:

1. **in parallel** cut all odd numbered leaves which are left sons
2. **in parallel** cut all odd numbered leaves that are right sons
3. **in parallel** divide the 'rank' of each leaf by two.

On the left of figure 5(b) is the result of applying step 1 to the expression tree of figure 5(a). The result of then applying step 2 is shown on the right . Step 3 ensures that the leaves of the new tree are ranked from left to right before the next iteration. Steps 1 and 2 are clearly achievable $O(n^{1/2})$ parallel time with n processors on a MCC^2, since both are achieved with a finite number of the RAR and RAW operations of [6]. Step 3 takes constant time. In the execution of steps 1 and 2 we mark as 'dead' those nodes which do not figure in the newly constructed tree. It takes $O(n^{1/2})$ parallel time (using techniques from [6]) to 'compress' the new tree (consisting of the live nodes only) before the next iteration. In fact, if we perform two leaves-cutting operations before each compression operation, then the recurrence relation for the time-complexity T(s) of the whole algorithm (for a tree of size s), satisfies (i) and so we have the result for dynamic expression evaluation stated in §1. It follows that there are efficient MCC^2 algorithms also for the recognition of bracket and of input-driven languages (see [4]).

References

[1] M.J.Atallah & S.E.Hambrusch. Solving tree problems on a mesh-connected processor array. Proceedings of the IEEE Symposium on the Foundations of Computer Science (1985).

[2] M.J.Atallah & S.R.Kosaraju. Graph problems on a mesh connected processor array. JACM, Vol.31, No.3, 649-667 (1984).

[3] A.M.Gibbons & W.Rytter. Efficient Parallel Algorithms. Cambridge University Press (1988).

[4] A.M.Gibbons & W.Rytter. Optimal Parallel Algorithms for Dynamic Expression Evaluation and Context-Free Recognition. To appear in Information and Control (1988).

[5] G.L.Miller & J.Reif. Parallel tree contraction and its applications. Proceedings of the IEEE Symposium on the Foundations of Computer Science (1985).

[6] David Nassimi & Sartaj Sahni. Data Broadcasting in SIMD Computers. IEEE Transactions on Computers. Vol. c-30, No.2, February 1981.

[7] David Nassimi & Sartaj Sahni. Finding Connected Components and Connected Ones on a Mesh-Connected Computer. SIAM Journal of Computing, 744-757 (1980).

[8] Y.N.Srikant. Parallel parsing of arithmetic expressions. IEEE International conference on parallel computing (1987).

[9] R.E.Tarjan & U.Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. Proceedings of the IEEE Symposium on the Foundations of Computer Science (1984).

[10] C.Thompson & H.Kung. Sorting on a mesh-connected parallel computer. CACM, 263-271 (1987)