Allocation of multiple processors to lazy boolean function trees – justification of the magic number 2/3

Alan Dix[†]

Computer Science Department
University of York, York, YO1 5DD, U.K.
alan@uk.ac.york.minster

First draft May 1991
Revised April 1992

In simulations of lazy evaluation of boolean formulae, Dunne, Gittings and Leng found that parallel evaluation by two processors achieved a speed-up of 2/3 compared to single processor evaluation. The reason that this factor was only 2/3 rather than a half was because the function nodes used for the simulation were all 'and' type functions, and hence sometimes the evaluation by the second processor proved unnecessary. This paper is intended to provide a justification for this 'magic number' of 2/3, by proving that it is exactly the speed-up obtained for pure balanced trees. A similar speed-up figure result is obtained for non-binary balanced trees and an equivalent speed-up figure for multiple processors is also given. Unbalanced trees have worse behaviour and by way of example the speed-up figure for an AVL tree is given (0.6952). Finally, numerical solutions are given for the case of arbitrary random binary trees. Since this report was first written Dunne *et al.* have proved analytically that the 2/3 figure does indeed hold for random trees generated by a reasonable distribution.

1. Background

This report springs from work by Dunne, Gittings and Leng on the computer-aided design of VLSI logic circuits.[1,2] They are in interested in efficient parallel evaluation strategies for simulation of Boolean expression trees. Demand-driven lazy evaluation of such trees means that, for instance, at 'and' nodes, if the first evaluated sub-tree yields 'false' the other need not be evaluated. In general, this will mean that parallel evaluation will not achieve 100% process utilisation. At the Seventh British Colloquium on Theoretical Computer Science in March 1991, Chris Gittings presented their simulation results which suggested a speed-up figure of 2/3 for two processors on random trees with 'and' type nodes.[3]

This magic number 2/3 arose out of empirical studies of randomly generated trees. Is this speed-up factor for two processors exact? If so is it a feature of the particular random trees chosen, or is it more general? Further, what is the generalisation to three or more processors? The following report is an attempt to analytically justify the figure of 2/3 and partially generalise the result. I say 'justify' as the report primarily deals with deterministic rather than random trees. Since this report was originally written Dunne *et al.* have produced a more complete analytic treatment of random trees both proving the speed-up of 2/3 can be achieved, on average, for 'and' type trees and giving similar figures for formula trees which also include 'xor' type nodes.[4]

† work funded by SERC Advanced Fellowship B/89/ITA/220

The following is a summary of the results in this report:

type of tree	nos. of processors	speed-up factor
balanced binary tree	2	2/3
balanced n-ary tree	2	2/3
balanced binary tree	$2^k + m$	$\frac{3.2^k - m}{3^{n+1}}$
AVL tree	2	$\frac{5+\sqrt{17}}{9+\sqrt{17}} = 0.6952$
random tree	2	tabulated for small trees

The main sections which follow cover these results in turn. The remainder of this section describes various assumptions about scheduling and similar issues.

Scheduling and costing assumptions

For the deterministic trees, the structure of the branches of a node are determined entirely by the number of nodes they contain. In the case of random trees, it is assumed that processor allocation (whether to assign both processors to one branch or to evaluate both in parallel) is based purely on the size of the sub-trees.

For the calculation of costs I have assumed that the costs are purely in terms of the terminal nodes evaluated, the function nodes are ‘free’. Assuming the converse or assigning costs to both leaves and internal nodes appears to make no difference to the assymptotic speed-up figures.

To simplify initial analyses I assume that the trees are not in fact randomly generated, but are perfectly balanced. For this case, with the above scheduling assumptions, the figure of 2/3 is exactly correct for the 2 processor case, and also equivalent figures can be calculated for a general p processor allocation regime.

There are a few additional scheduling issues that arise with multiple processors and with unbalanced trees. If two branches are evaluated, what happens if one completes before the other, and evaluates to such a value that the second becomes unnecessary? I assume that it is possible to abort the computation of the second branch so that the processor(s) become available for further computation. Other options are to wait for both, or to add the processors for the first branch to those of the second.

This is further complicated with non-binary branching factors. With three children, we may allocate processors to two branches. If one finishes before the other, can we allocate its resources to the third branch, or do we have to wait for both?

Random tree generation

Where random binary trees are used they are assumed to be generated by the following procedure:

1. The size of tree is chosen n. This is the number of terminal nodes in the tree.
2. A split is chosen for the children of the root node so that one child had size r and the other size $n-r$. The split r is chosen uniformly in the range 1 to $n-1$.
3. Each child is split in the same fashion until trees of size 1, that is terminal nodes, are reached.

An alternative is to choose the tree randomly from all the possible trees with n nodes. The number of binary trees with n nodes is:

$$B(n) = \frac{(2n-2)!}{n! \times (n-1)!}$$

This would give the probability of an $r, n-r$ split of:

$$\frac{B(r) \times B(n-r)}{B(n)}$$

However, as Dunne *et al.* point out, this yields unnaturally unbalanced trees. (In fact, asymptotically a half of all such trees are completely degenerate with a 1;n-1 or n-1;1 split.)

Effect of laziness

An ‘xor’ operation or its negation always requires both its sub-formulae branches to be evaluated, however ‘and’ type operations including or, nand etc. may sometimes be evaluated using only one branch. All the analyses in this report are based on a tree of purely ‘and’ type nodes and hence, without loss of generality, we can assume that the nodes are all ANDs, and that the probability of any sub-tree being 1 or 0 is 50:50. Thus after evaluating one child tree, there is only a 1/2 chance of requiring the other child tree. Thus we can see at once why the speed-up is 2/3 rather than 1/2. If we calculate both branches in parallel half the time the evaluation of the second branch will have been in vain – that is wasting 25% of the combined processor effort. The next section makes this argument more rigorously.

2. Binary balanced trees – two processors

This is the simplest case. Call the expected time to calculate a node at height h with one processor w_h, and with two processors w_h^2. We assume that there is a constant k such that:

$$w_h^2 = k\, w_h$$

We can generate a recurrence relation for w_h. With one processor we must evaluate one child first, with cost w_{h-1}. With probability 1/2 this is all the work that is needed and we can return an answer. If not then we calculate the second child with additional cost w_{h-1}. Hence the expected cost of the parent is:

$$w_h = w_{h-1} + \frac{1}{2} w_{h-1}$$

$$w_h = \frac{3}{2} w_{h-1}$$

Of course, w_1 is the cost of calculating a terminal node.

We can evaluate the above expression fully and obtain that:

$$w_h = \frac{3}{2}^{h-1} w_1$$

However, this is unnecessary, as we can just use the recurrence relation.

With two processors we have two choices, we can either allocate both to the first branch, then both to the second (if necessary). This is a sequential allocation for the node. In this case we get the same recurrence as for one processor:

$$w_h^2 = w_{h-1}^2 + \frac{1}{2} w_{h-1}^2$$

$$w_h^2 = \frac{3}{2} w_{h-1}^2$$

If we use the constant speed-up assumption this gives us:

$$k\, w_h = \frac{3}{2} k\, w_{h-1}$$

Which doesn’t tell us much about k. For notational purposes we can call this allocation strategy 2;2.

The second possibility is to calculate both branches in parallel, allocating one processor each. We can call this 1||1. Each branch then takes time w_{h-1}, but these are done in parallel, hence the times overlap and the total time is:

$$w_h^2 = w_{h-1}$$

This is more interesting as we then get:

$$k\, w_h = w_{h-1}$$

and hence (using the recurrence for one processor) that:

$$k = \frac{2}{3}$$

So, for balanced binary trees the magic figure 2/3 is exact.

Notice that it didn't in fact matter whether we did the 2;2 or the 1||1 split, both give the same speed-up. At some stage we do have to make the decision however, and split the processors, otherwise we could get to a leaf of the tree and still have two processors, of which one would would be 'wasted'. It is quite reasonable however to wait until we get to the last non-leaf node and do the split at this level.

Non-terminal node costs

Although there was no cost added for the evaluation of the non-terminal nodes, the same result would have been obtained for large n. This is seen most easily by solving the recurrence relation with added costs. If the cost of the terminal nodes (w_1) is a and the cost of evaluation a non-terminal node is b, the recurrence relation for w_h is:

$$w_h = \frac{3}{2} w_{h-1} + b$$

$$w_1 = a$$

This has solution:

$$w_h = \alpha^h a + \frac{\alpha^{h-1} - 1}{\alpha - 1} b$$

where α is 3/2.

So asymptotically

$$w_h = \alpha^h a + \frac{\alpha^{h-1}}{\alpha - 1} b$$

The formula for w_h^2 is similar to before:

$$w_h^2 = w_{h-1} + b$$

The term w_{h-1} clearly dominates b and hence asymptotically $w_h^2 = w_{h-1}$ and hence there is again a speed-up of α^{-1} that is 2/3.

Balanced non-binary tree

In addition the 2/3 figure persists for non-binary trees. Take the case of a ternary tree, with fully lazy nodes. That is all the nodes are ternary ANDs or ternary ORs. We first obtain a recurrence relation for one processor:

$$t_h = t_{h-1} + \frac{1}{2} t_{h-1} + \frac{1}{4} t_{h-1}$$

We now look at two processors. Again we assume a constant speed-up factor k'

$$t_h^2 = k' t_h$$

The possible allocation strategies are:

Fully sequential: 2;2;2

Parallel then sequential: (1||1);2

Sequential the parallel: 2;(1||1)

As before the fully sequential case gives us no information about the speed-up factor. The (1||1);2 and 2;(1||1) cases give us:

$$t_h^2 = t_{h-1} + \frac{1}{4} k' t_{h-1}$$

$$t_h^2 = k' t_{h-1} + \frac{1}{2} t_{h-1}$$

respectively. In both cases we solve for k' and get 2/3. So again it doesn't matter which allocation strategy you use, and we retain the magic number 2/3.

3. Balanced binary tree with many processors

If we have p processors, we can do a similar analysis to the two processor case. We already have the recurrence relation for 1 processor:

$$w_h = \frac{3}{2} w_{h-1}$$

We call the time taken by p processors w_h^p, and assume a constant speed up factor for p processors k_p:

$$w_h^p = k_p w_h$$

We now work out a recurrence relation for p processors. There are two major possibilities:

Fully sequential: $p ; p$.

That is just use all p for one branch and then if necessary use all p for the other.

Parallel: $r \| p-r$.

For some r we use r processors to evaluate one branch and the remaining $p-r$ processors to evaluate the second branch in parallel.

As with the binary case, the sequential case gives us no information (but is an acceptable allocation decision so long as it is not to close to the leaves), so we concentrate on the parallel cases.

Assume we have chosen r. The time for the whole node is:

$$w_h^p = \frac{1}{2} (w_{h-1}^r + w_{h-1}^{n-r})$$

This is because half the time the first branch to finish is sufficient and the slower branch can be aborted, and

the other half of the time we must wait for the slower branch. Hence the expected time for the whole node is the average of the times for each branch.

We can substitute for the w_h^p using the constants k_p:

$$k_p\ w_h \ = \ \tfrac{1}{2}\,(\,k_r\ w_{h-1} \ + \ k_{n-r}\ w_{h-1}\,)$$

and then divide by the recurrence relation for w_h:

$$k_p \ = \ \tfrac{1}{3}\,(\,k_r \ + \ k_{n-r}\,)$$

Of course, we want the best choice of r and hence we choose r to minimise the left hand side.

Hand calculations

This is now basically a dynamic programming problem.

$$k_p \ = \ \tfrac{1}{3}\ \mathbf{inf}_{1\le r<n}\,(\,k_r \ + \ k_{n-r}\,)$$

Each value of k_p can be calculated knowing all the values of k_r for r less than p. I evaluated these by hand, tabulating the values in a triangle.

p		1	2	3	4	5
	k_p	1	2/3	5/9	4/9	11/27
1	1	2/3	5/9	14/27	13/27	
2	2/3	5/9	4/9	11/27		
3	5/9	14/27	11/27			
4	4/9	13/27				
5	11/27					

The tableau is labelled by pairs of p values. The column next to the indices has the values of k_p. The central value at the cell (i,j) is $(k_i + k_j)/3$ calculated from the margin values of k_i. Finally the diagonals can be scanned to choose the value of the next k_p.

For example at cell (2,3) we have the value 11/27 which is (2/3+5/9)/3. The values along this diagonal (13/27, 11/27, 11/27 and 13/27), represent the different choices of r for splitting 5 processors (1||4, 2||3, 3||2, 4||1). The smallest of these is 11/27 corresponding to a choice of 2 or 3 for r. This is the value filled in for k_5. We could then use this value to calculate the next diagonal etc.

This was getting a little tedious, and I was approaching the limit of my arithmetic accuracy. However, there was a tendency for the best strategy to be the even or near even split of resources. The next stage is to prove this hypothesis.

Proof that even splitting is optimal

I want to prove that the optimal strategy is: if $p = 2m$, split $m||m$, or if $p = 2m + 1$ split $m||m+1$. In fact, a sneak look at the next section gives a formula for k_p given this assumption. We could calculate this, and then by induction prove that even splitting is optimal for p processors given the formula is correct for all r less than p. In fact, we only need one property of the series k_p, that it is convex. This is sufficient to prove that even splitting is optimal.

To show that even splitting is optimal, we need to show that:

$$\forall\, r < \hat{r} \quad k_r + k_{p-r} \geq k_{\hat{r}} + k_{p-\hat{r}}$$

where $\hat{r} = \left\lfloor \frac{p}{2} \right\rfloor$, the even split of p. This is clearly true if k_p is convex.

So we set up a double induction.

(i) Prove that even splitting is optimal for p processors, given k_r is convex for $r < p$.

(ii) Prove that k_p is convex given k_r is convex for $r < p$ and that even splitting is optimal for p processors and less.

The first part of the proof we have already declared obvious by examination of the above formulae. We are thus left with part (ii). We could at this stage appeal directly to the formula in the next section, but we will use a more abstract proof.

LEMMA: k_r convex on $[1, p-1]$ and even splitting correct on $[1, p]$

$\Rightarrow$ k_r convex on $[1, p]$

PROOF:

To prove convexity we need to show that:

$$\forall\, r \in [2, p-1] \quad k_{r+1} + k_{r-1} - 2k_r \geq 0$$

We look at two sub cases:

CASE: r even: $r = 2m$

Using formula for even splitting we get:

$$\begin{aligned} k_{r+1} &= \tfrac{1}{3}(k_{m+1} + k_m) \\ k_r &= \tfrac{2}{3} k_m \\ k_{r-1} &= \tfrac{1}{3}(k_m + k_{m-1}) \end{aligned}$$

We expand the convexity condition:

$$\begin{aligned} k_{r+1} + k_{r-1} - 2k_r &= \tfrac{1}{3}(k_{m+1} + k_m + k_m + k_{m-1} - 2\times 2k_m) \\ &= \tfrac{1}{3}(k_{m+1} - 2k_m + k_{m-1}) \\ &\geq 0 \quad - \text{ by inductive convexity hypothesis} \end{aligned}$$

CASE: r odd: $r = 2m + 1$

This proceeds in a similar fashion:

$$\begin{aligned} k_{r+1} &= \tfrac{2}{3} k_{m+1} \\ k_r &= \tfrac{1}{3}(k_{m+1} + k_m) \\ k_{r-1} &= \tfrac{2}{3} k_m \end{aligned}$$

We expand the convexity condition:

$$\begin{aligned} k_{r+1} + k_{r-1} - 2k_r &= \tfrac{1}{3}\left(2k_{m+1} + 2k_m - 2(k_{m+1} + k_m)\right) \\ &= 0 \end{aligned}$$

These are the only two cases, so the lemma is proved.

So we have proved the induction lemma, and as the convexity condition is vacuously true for $r \in [1, 2]$, we have also proved that even splitting is always optimal and that k_p is always convex.

Calculating times given even splitting

We now know that the optimal strategy is even splitting. That is,

$$k_p = \frac{1}{3}\left(k_{\hat{r}} + k_{n-\hat{r}}\right)$$

where $\hat{r} = \left\lfloor \frac{p}{2} \right\rfloor$

We can easily calculate this series:

p	1	2	3	4	5	6	7	8	9	10	…
k_p	1	2/3	5/9	4/9	11/27	10/27	9/27	8/27	23/81	…	

We notice first that at the values $p = 2, 4, 8$, k_p is successive powers of 2/3. Further, the series drops linearly between these values. That is, for powers of two:

$$p = 2^n \qquad k_p = \left(\frac{2}{3}\right)^n$$

and in the general case

$$p = 2^n + m \qquad k_p = \left(\frac{2}{3}\right)^n - \frac{m}{3^{n+1}}$$

$$= \frac{3.2^n - m}{3^{n+1}}$$

The proof by induction of this formula is quite easy. We assume that p is of the general form:
in which case, we calculate $\hat{r}$:

$$\hat{r} = 2^{n-1} + \hat{m}$$
$$p - \hat{r} = 2^{n-1} + m - \hat{m}$$

where $\hat{m} = \left\lfloor \frac{m}{2} \right\rfloor$

We can then use the recurrence formula for k_p:

$$k_p = \frac{1}{3}\ \frac{3\times 2^{n-1} - \hat{m} + 3\times 2^{n-1} - (m - \hat{m})}{3^n}$$

$$= \frac{3.2^n - m}{3^{n+1}}$$

Which is just what we were after. The base case with $p = 1$ is clearly correct with $n = 0$ and $m = 0$. So the formulae is correct.

Conclusions: balanced tree – general processors

We have shown that for any number of processors p, there is a constant speed-up factor k_p given by

$$k_p = \frac{3.2^n - m}{3^{n+1}}$$

where $p = 2^n + m$.

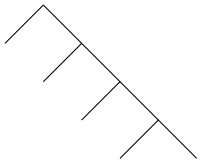
This speed-up is achieved by near equal splitting of the processors between the two child branches at any node.

4. Unbalanced trees – the AVL tree and random trees

The magic figure of 2/3 for two processors does not hold for unbalanced trees. This is because without dynamic reallocation of processors to evaluating branches, we will often end up with processors idle waiting for others to finish.

Degenerate binary tree

An extreme example of this is the degenerate binary tree.



If this is of height h the time taken with one processor is $h+1$ and with two processors is h. That is there is NO asymptotic speed-up. This is because when we decide to do parallel computation, one processor evaluates a single node and then is left idle whilst the other evaluates the whole right hand branch. Obviously it is here that dynamic reallocation of processors would be helpful. In this case, the degenerate binary-tree of height h is exactly equivalent to a $h+1$-ary tree of height 1 and the speed-up factor is indeed 2/3.

AVL tree

A less unbalanced tree to look at is the maximally unbalanced AVL tree. The tree of height h has two branches, one of height $h-1$ and one of height $h-2$. The number of terminal nodes in the tree of height h is $fib(h)$, which is asymptotically proportional to γ^h, where γ is the golden ratio:

$$\gamma = \frac{1+\sqrt{5}}{2}$$

If we have one processor, then it is clearly better to evaluate the shorter branch first and then the longer, hence the time for one processor is given by:

$$w_h = w_{h-2} + \frac{1}{2} w_{h-1}$$

Thus w_h is asymptotically proportional to β^h where β is the solution of the characteristic polynomial:

$$\beta^2 = 1 + \frac{1}{2}\beta$$

that is

$$\beta = \frac{1+\sqrt{17}}{4}$$

With two processors we again find that sequential 2;2 splitting tells us nothing about the speed-up factor, so we look at parallel 1||1 splitting. In this case, one branch finishes first, and half the time is sufficient. hence the average time is:

$$w_h^2 = \frac{1}{2}(w_{h-2} + w_{h-1})$$

Now taking the asymptotic form of w_h we have:

$$\begin{aligned} w_h &= w_{h-2} + \tfrac{1}{2} w_{h-1} \\ &= (1 + \tfrac{1}{2}\beta)\, w_{h-2} \end{aligned}$$

and

$$\begin{aligned} w_h^2 &= \tfrac{1}{2}(w_{h-2} + w_{h-1}) \\ &= \tfrac{1}{2}(1+\beta)\, w_{h-2} \end{aligned}$$

Hence the speed-up ratio is:

$$\frac{\frac{1}{2}(1+\beta)}{1+\frac{1}{2}\beta} = \frac{5+\sqrt{17}}{9+\sqrt{17}} = 0.6952$$

which is less than but not too different from the magic 2/3.

Speed-ups for greater numbers of processors can be calculated using dynamic programming as for the balanced binary tree. My guess is that the optimal split for $p = fib(n)$ processors is to allocate $fib(n-2)$ to the smaller and $fib(n-1)$ to the larger branch, but that is just a guess.

General unbalanced trees

In general we can see that we expect a figure of somewhat more than 2/3 for unbalanced trees, however not necessarily very much more, as the AVL tree has its terminal nodes split in the ratio γ:1, which is fairly unbalanced, but still has a speed-up not much less than 2/3. In fact, in some ways the AVL tree could be seen as particularly bad, as every node including all the 'nearly' leaves are unbalanced. In a random tree there will be some places where the nodes are 'almost' balanced, and at these points you can choose to schedule one processor per branch and thus almost acheive a 2/3 speed-up. By looking ahead down the tree you can choose to split if the balanced-ness of the node in question is the best you are going to get. Thus 2/3 is an upper† bound for the speed-up, but it is not unreasonable to assume that the actual speed-up is much short of it.

A lower bound for random trees can be calculated by using more extensive dynamic programming. If we ignore the structure of the branches of a node, we can simply choose whether to do 2;2 or 1||1 splitting on the basis of the average time to completion for branches with the appropriate number of sub-nodes.

Let a_n be the average time taken for nodes of size n (i.e. with n leaves) with one processor and a_n^2 be the equivalent figure for 2 processors.

We look first at the 2 processor case at a node of size n. If the two children are of size r and $n-r$, we decide to split 1||1 if:

$$\tfrac{1}{2}(a_r + a_{n-r}) < a_r^2 + \tfrac{1}{2} a_{n-r}^2$$

(assuming $r \le n - r$).

N.B. This decision is dependent on the number of nodes in each branch but ignores the structure of these sub-trees.

We call the minimal value of these two $a_{r,n-r}^2$ which represents the average time taken for a node of size n **given** it has children of size r and $n-r$. (If $r > n - r$ we simply let $a_{r,n-r}^2 = a_{n-r,r}^2$.)

---

† in the context of a speed-up fraction a small number is good and a large number bad. So when I say upper bound I mean that the actual speed-up fraction will be greater than 2/3!

Finally we can calculate a_n^2 as the average of these.

$$a_n^2 \quad = \quad \frac{1}{n-1} \sum_{r=1}^{n-1} a_{r,n-r}^2$$

To get the speed-up we do the similar calculation of a_n

$$a_n \quad = \quad \frac{1}{n-1} \sum_{r=1}^{n-1} a_{r,n-r}$$

where

$$a_{r,n-r} \quad = \quad a_r + \tfrac{1}{2} a_{n-r}$$

if $r \leq n - r$ and $a_{r,n-r} = a_{n-r,r}$ otherwise.

The asymptotic speed-up factor is then $\lim a_n^2 / a_n$

The first 20 terms in these series are tabulated below together with the values at 40, 60, 80 and 100. The speed-up figure is just over 0.7 for most of the range up to 20, and is still above 0.69 at $n = 100$, but is still dropping albeit slowly. It is this ratio which Dunne *et al.* have since proved to be assymptotically 2/3. Incidentally, the point at which parallel evaluation (1||1) rather than sequential (2;2) is chosen appears to be at about $r > n/3$.

Speed-up figures for random trees n = number of leaves			
n	a_n	a_n^2	speed-up
1	1.0000	1.0000	1.0000
2	1.5000	1.0000	0.6667
3	1.7500	1.2500	0.7143
4	2.0000	1.4167	0.7083
5	2.1875	1.5625	0.7143
6	2.3625	1.6708	0.7072
7	2.5083	1.7792	0.7093
8	2.6512	1.8738	0.7068
9	2.7762	1.9663	0.7083
10	2.8977	2.0509	0.7078

n	a_n	a_n^2	speed-up
11	3.0071	2.1296	0.7082
12	3.1145	2.2031	0.7074
13	3.2129	2.2734	0.7076
14	3.3094	2.3397	0.7070
15	3.3990	2.4023	0.7068
16	3.4873	2.4626	0.7062
17	3.5702	2.5198	0.7058
18	3.6519	2.5749	0.7051
19	3.7290	2.6271	0.7045
20	3.8052	2.6781	0.7038
40	4.9803	3.4738	0.6975
60	5.8245	4.0510	0.6955
80	6.5073	4.5171	0.6942
100	7.0909	4.9146	0.6931

References

1. Paul E. Dunne, Chris J. Gittings and Paul H. Leng, ‘‘Parallel demand-driven evaluation of logic networks’’, in *Proc. 28th annual Allerton Conf. on Communication, Control and Computing* (1990).
2. Chris J. Gittings, ‘‘Parallel Demand-Driven Simulation of Logic Networks’’, Ph.D. Dissertation, Department of Computer Science, University of Liverpool (1991).
3. Paul E. Dunne, Chris J. Gittings and Paul H. Leng, *Parallel algorithms for logic simulation*, The Seventh British Colloquium on Theoretical Computer Science (March 26th-28th 1991).
4. Paul E. Dunne, Chris J. Gittings and Paul H. Leng, ‘‘An Analysis of Speed-up ratios in Demand-Driven Multiprocessor Simulation Algorithms’’, CS/91/22, Department of Computer Science, University of Liverpool (August 1991).