A Parallel Algorithm for Zero Skew Clock Tree Routing

Zhaoyun Xing and *Prith Banerjee*

Technical Report No. CPDC-TR-9801-007

Feburary 1997

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

Abstract

In deep sub-micron fabrication technology, clock skew is one of the dominant factors which determine system performance. Previous works in zero skew clock tree routing assume that the wires have uniform size, and previous wire-sizing algorithms for general signal nets do not produce the exact zero skew. In this paper, we first propose an algorithm to get the exact zero skew wire-sizing by using an iterative method to make the wire size improvement. Our experiments on benchmark clock trees show that the algorithm reduces the source sink delay more than 3 times that of the clock trees with uniform wire sizes and keeps the clock skew zero. Motivated by the computation intensive nature of the zero skew clock tree construction and wire-sizing, we propose a parallel algorithm using a cluster-based clock tree construction algorithm and our zero skew wire-sizing algorithm. Without sacrificing the quality of the solution, on the average we obtain speedups of 7.8 from the parallel clustering based clock tree construction algorithm on an 8 processor SUN SPARC Server 1000E shared memory multi-processor.

1 Introduction

In deep sub-micron fabrication technology, clock skew is one of the dominant factors which determine system performance. As a result, numerous researchers have worked on the zero-skew clock tree construction [2, 9, 3, 4, 7, 13]. Among those, two models have been used to approximate clock source sink delays. The first model assumes that the delay is proportional to the path wire-length, and it is therefore called the linear delay model. The second is called the Elmore delay model which gives much more accurate approximation than the linear model.

Most of the zero skew clock tree construction procedures use variations of the Deferred-Merge Embedding (DME) algorithms [2, 3, 7]. The DME algorithm usually takes one bottom-up phase and one top-down phase. In each step of the bottom-up phase, one selected pair of the zero skew subtrees will be merged into a larger zero skew tree. Instead of only one point, the feasible positions of the root of the new large tree are a set of points. In each step of the top-down phase, the position of one internal node will be determined based on the objective of the optimization. The DME algorithm can be applied to both the linear and the Elmore models. The zero skew merging under the Elmore model is based on the work of Tsay [13]. Among various zero-skew clock tree construction schemes, CL, a clustering based construction algorithm [7], is the best in terms of total wire-length.

Under the Elmore delay model, the delay of a clock tree not only depends on its topology, but also on the size or width of each connection wire. Cong et al. [6] optimize the weighted source sink delay assuming that there are only a finite number of wire-sizes available. This wire-sizing method is called discrete wire-sizing. Chen et al. [5] generalize the optimization assuming a wire can take a real range of sizes. This method is called continuous wire-sizing. Kay et al. [11] proposed an algorithm for wire sizing which can handle zero skew clock tree wire sizing. But it requires linear relaxation for the nonlinear optimization problem. When the number of wires is large, it may take excessive amount of running time to solve the linear programming problem.

In this paper, as described in Section 6, though it can be extended to the multi-staged clock routing, we consider the single stage clock tree routing problem, i.e., the buffer insertion problem is not considered. Also, we assume there is only one clock pulse source and there is no coupling capacitance. we first propose an algorithm to get the exact zero skew wire-sizing. This approach uses an iterative method to make wire size improvement. Each time when we make an alternate choice of wire size for some segment, we propagate this information to the root of the tree by zero skew merging to make sure that we indeed get an improvement. Our experimental results show that this algorithm can reduce the clock delays more than 3 times over the CL algorithm using uniform wire sizes while the skew of clocks are set to zero.

Like all the other layout synthesis problems such as placement and routing, the clock routing process is a computation intensive process. For large clock tree nets of the complex designs, it may take hours to route a clock tree. Parallel processing is one of attractive methods to reduce the increasing running time. There has been a lot of research on developing parallel algorithms for a wide range of VLSI CAD problems[1]. Until now there has been no work on parallel clock tree routing. In this paper, we propose a parallel algorithm for clock tree construction and its wire-sizing. Our parallel clock tree construction is based on the CL algorithm, and the parallel zero skew clock tree wire-sizing is based on our zero skew wire-sizing algorithm. Our experimental results show good performance in the parallel algorithms of clock tree construction and zero skew wire-sizing.

In summary, our key contributions in this paper are: (1) Integrating the wire-sizing operation with the zero skew clock tree routing, (2) A parallel algorithm to speedup the zero skew clock tree construction and wire-sizing problems.

The remainder of the paper is organized as follows. Section 2 defines the zero skew clock tree routing problem using the Elmore delay model. Section 3 gives a zero skew wire-sizing algorithm and the experimental results on some benchmark clocks. In Section 4, we propose the parallel algorithms for the zero skew clock tree construction, topology improvement, and wire-sizing. Also, we report the experimental results on those parallel algorithms in Section 5. We conclude this paper in Section 6.

2 Zero Skew Clock Tree Routing

Throughout this paper, we make the following assumptions and use the following notations. Assume T is a clock routing tree with n wire segments. The driver or root of T is s_0, which has resistance r_0. T has a set of sinks $\{N_1, \cdots, N_s\}$ with load capacitances $c_i^s, 1 \leq i \leq s$. For each wire segment s_i, let l_i be the length of s_i and w_i be the width of s_i. Assume $\{W_1, \cdots, W_r\}$ is the set of discrete choices of wire width for each wire. Let $dec(s_i)$ be the set of descendants of s_i excluding s_i. Let $ans(w_i)$ be the set of ancestors of s_i excluding s_i. Let $\bar{r}$ be the resistance and $\bar{c}$ be the capacitance per unit square.

Based on the above assumption, under the π-model [5], the resistance of wire segment s_i is $r_i = \bar{r}l_i/w_i$, and the capacitance of s_i is $c_i = \bar{c}l_i w_i$. The down-stream capacitance of s_i is given by $C_i = \sum_{s_j \in dec(s_i)} \bar{c}l_j w_j + \sum_{N_j \in dec(s_i)} c_j^s, 1 \leq i \leq n$. Under the Elmore delay model the signal delay of sink N_i is given by

$$D_i = \sum_{s_j \in ans(s_i)} r_j(C_j + c_j/2) = \sum_{s_j \in ans(s_i)} \bar{r}l_j/w_j(\sum_{s_k \in dec(s_j)} \bar{c}l_k w_k + \sum_{N_k \in dec(s_j)} c_k^s + \bar{c}l_j w_j/2) \quad (1)$$

If $D_i = D_j, 1 \leq i, j \leq s$, then T is called a zero skew clock tree.

In this paper we consider the following zero skew clock tree routing problem

$$\begin{aligned} &\text{minimize } f(l_1, \cdots, l_n, w_1, \cdots, w_n) = D \\ &\text{subject to } D_i = D, \text{ and } w_i \in \{W_1, \cdots, W_r\}, 1 \leq i \leq n \end{aligned} \quad (2)$$

3 A Zero Skew Wire-Sizing Algorithm

When the topology of a clock tree is given, wire-sizing can further reduce the delay of the clock trees. In this section, we look at one wire-sizing algorithm which generates the exact zero skew clock trees with less source-sink delays than those with the uniform wire sizes.

The zero skew wire-sizing algorithm uses an iterative approach. In each step, one wire segment is selected and an alternate wire-size is tried. Because of the change of wire-size of this segment, the zero skew property will not hold. To make the skew of the tree still zero, we have to re-merge the subtree rooted at the current wire with its sibling. This re-merging

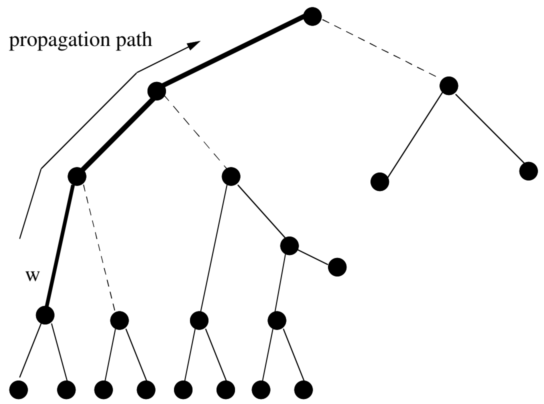

Figure 1: When the size of a wire is locally optimized, the effect of the wire size change is propagated by zero skew merging to the root of the clock tree. The length of all the wires along the propagation path and their siblings may change but their wire-sizes remain unchanged. The thick wires are on the propagation path, and dashed wires are their sibling.

generates a new subtree rooted at the parent of the current wire. In this step, we assume that the sibling wire uses the same wire size. Moreover, the zero skew re-merging may change the wire-length of its sibling. Then the new parent wire needs to re-merge with its sibling. This propagation continues until all the wire segments on the path from the current wire to the root wire s_0 are re-merged. If the new generated tree has less delay then we accept the new wire size, otherwise this wire is ignored, and we proceed to the next optimization step. Figure 1 illustrates this propagation process.

3.1 Analysis of the Zero Skew Wire-Sizing Algorithm

Throughout the analyses of the paper, assume M stands for the number of iteration in the zero skew wire-sizing algorithm. Assume the tree generated by the CL algorithm is well balanced. Then the height of the tree is $O(\log(n))$. In the zero skew wire-sizing algorithm, each propagation will take $O(\log(n))$ time. In each iteration, we will try $r-1$ different wire sizes for each wire. This takes $O(n(r-1)\log(n))$. Therefore, the total running time is

$O((r-1)Mn\log(n))$.

In the worst case, it may take n^r iteration to make wire-sizing algorithm converge. To avoid the iteration explode, in the wire-sizing algorithm, a number M is used as the number of iterations.

3.2 Experimental Results

We implemented and tested the zero skew wire-sizing algorithm on several benchmark clocks r1 - r5 [13] on a SUN SPARCstation 5. The topology of the clock tree is generated by the CL algorithm without local optimization. In our experiments, the set of wire widths is $\{1\mu m, 2\mu m, 3\mu m, 4\mu m\}$. The algorithm is executed until there is no change. Table 1 lists the delay and the run time results of the zero skew wire-sizing algorithm. Table 2 lists the results of scaled delays of r1-5 by the CL algorithm with uniform minimum wire width ($1\mu m$), and the CL algorithm with uniform maximum wire width ($4\mu m$). The scale is based on the zero skew wire-sizing algorithm. Our experiments show that our zero skew wire-sizing algorithm can generate clocks with source sink delay three times less than those generated by CL algorithm with either minimum or maximum uniform wire size. The results of running times indicate that our algorithm is efficient.

circuit	wire #	delay (ns)	runtime(s)
r1	533	0.35	9.7
r2	1195	1.12	22.3
r3	1723	0.82	86.4
r4	3805	4.12	91.2
r5	6201	5.39	672.4

Table 1: Delay and run-time results of the zero skew wire-sizing algorithm.

circuit	zero skew	uniform $1\mu m$	uniform $4\mu m$
r1	1.0	3.71	2.80
r2	1.0	2.88	2.14
r3	1.0	6.35	4.45
r4	1.0	3.73	2.61
r5	1.0	4.25	3.56

Table 2: Comparisons of clock delays by wire-sizing algorithms.

4 Parallel Zero Skew Clock Tree Routing

Even though the run times in Table 1 show that our algorithm is very efficient, the run times are expected to be in the hours for very large clock trees having 10,000 to 100,000 wire segments as is expected in future complex microprocessor designs. It is therefore important to look at parallel algorithms to reduce the run times. In this section, we propose a parallel algorithm for the zero skew clock tree routing. This algorithm consists of zero skew clock tree construction and wire-sizing. The clock tree construction is based on the clustering algorithm CL with local improvement. The wire-sizing is based on our zero skew wire-sizing algorithm. To illustrate the parallelism differences inside CL and local improvement, we discuss their parallel algorithms separately.

4.1 Parallel Zero Skew Clock Tree Construction

The clustering based algorithm CL is the best zero skew clock tree construction algorithm in terms of the total wire-length. Based on this algorithm, in this section we propose a parallel algorithm to reduce the extremely large running times when the number of sinks increases.

For a given set of zero skew subtrees which need to be merged into a large zero skew tree, CL first finds the nearest neighbor graph. The nodes in this graph are the roots of the zero skew subtrees, and an edge is given if one node is the nearest neighbor of the other. The edge weight is the distance between two nodes. The edges are sorted in an increasing edge

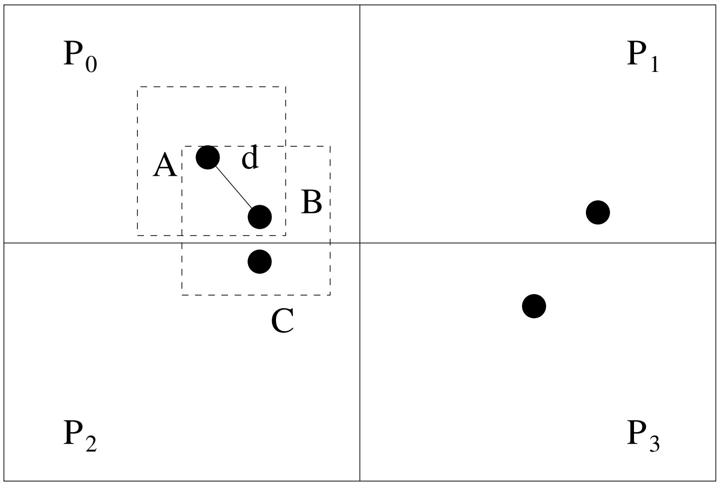

Figure 2: Node partition and parallel nearest neighbor computation. Solid dots are the roots of the zero skew subtrees. After grid partition, each processor has two nodes. After the local nearest neighbor computation, node A has the nearest neighbor B with a distance of d. Since the bounding square of A with side length of $2d$ does not overlap with territories of the other processors, B is also the global nearest neighbor of A. Moreover, since the bounding square of B overlaps with the territory of processor p_2 we have to check all the nodes in p_2 to get the nearest neighbor of B which is C. In addition, since the bounding square of B does not overlap with the territories of processors p_1 and p_3, we do not have to check any of their nodes.

weight order. The nodes (roots of the subtrees) are zero skew merged in that order. A node can have multiple edges and if either of the nodes of an edge have been merged, the edge is ignored. This process continues until there is only one node (one tree) left.

The dominant parts of the computation in CL are the nearest neighbor graph construction and edge sorting. The most efficient implementation of the nearest neighbor computation is by Delaunay triangulation, which has a complexity of $O(n \log(n))$.

Assume that all the sinks are randomly distributed in a box $\mathcal{B}$ and there are p processors. To get an efficient parallel nearest neighbor computation, we partition $\mathcal{B}$ evenly into p two dimensional grids such that each process is assigned a rectangular area and has its own set of nodes. The parallel CL algorithm proceeds as follows. First, each processor independently constructs the nearest neighbor graph based on all the nodes it owns. Second, we have to extend this local nearest neighbor to the global nearest neighbor. For a given node N, assume its edge weight, the distance to its nearest neighbor, is d. Form a bounding square with N as the center and $2*d$ as the side length. For any processor, if this square does not overlap with its territory, then all the nodes owned by the processor have a distance of at least d from N. Thus the nearest neighbor of node N cannot be owned by that processor. Otherwise, we need to compare the distance between N and each node owned by the processor against the current edge weight to get the nearest neighbor of N. Third, after we construct the nearest neighbor graph, each processor sorts all the edges it owns. Fourth, like in merge sort, the edges are merged such that they are in a increasing weight order. Figure 2 illustrates an example node partition and nearest neighbor graph construction.

4.1.1 Analysis of the Parallel CL Algorithm

Assume there are n subtrees to be zero skew merged. If the distribution of those subtrees is even, then each processor owns n/p nodes. The local nearest neighbor computation takes $O(n/p \log(n/p))$ time. Assume n nodes reside on a $\sqrt{n} \times \sqrt{n}$ grid. After the local nearest neighbor computation, the distance of each node to its nearest neighbor is 1. For each processor, the ownership checking of the nodes owned by the other processors is only necessary at most $4\sqrt{n/p}$ on the boundary of the territory. This takes $O(4\sqrt{n/p}n/p) = O((n/p)^{\frac{3}{2}})$. The sorting of the local edges takes $O(n/p \log(n/p))$. Assuming the sending of one edge information takes time c, then the merging of the sorted edges takes $O(cn)$ time. In summary, the total running time of the parallel CL algorithm is $O((n/p)\log(n/p)) + O((n/p)^{\frac{3}{2}}) + O(cn)$. Recall that the serial CL algorithm takes $O(n \log(n))$. When the term $O((n/p)^{\frac{3}{2}})$ is not dominant (which is the case for even when n is fairly large, as shown by our experiments in Section 5), we may get super-linear speedup from the parallel CL algorithm.

4.2 Parallel Improvement Algorithm

The improvement algorithm in CL accounts for the best quality of CL in terms of total wire-length. In the improvement algorithm, for any internal node v, assume T_v is a subtree of v with at most $2m-1$ nodes for some positive integer $4 \leq m \leq 6$. Then T_v has at most m leaves. Without changing the capacitances and delays of those leaf nodes, using exhaustive search an optimal zero skew merging sequence can be given. A different merging sequence generates a different tree topology. Though the original objective of improvement is to minimize the total wire-length, the optimization approach can also be applied to the optimization of source sink delay. In every iteration, the nodes of the tree will be optimized in a bottom up order. The improvement algorithm terminates after a fixed number of iterations, or when there is no delay change after execution of two consecutive iterations. If m is 4, then the search is equivalent to a search tree with depth of 2. Using the depth number instead of the number of nodes can save some effort in implementation.

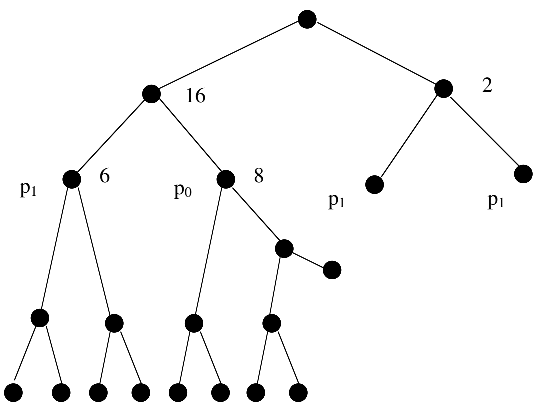

Figure 3: Subtree partition. Assume there are two processors. The subtree assignment will not occur on nodes of depth 1 since it will make an assignment of 16 nodes and 2 nodes on two processors. This is too off balance. But in depth of 2, we have 4 subtrees. After execution of our partitioning scheme, processor 0 gets one subtree with 8 nodes and processor 1 gets three subtrees with total number of nodes also 8. This is a well balanced partition.

The improvement over one internal node will not effect the improvement over another internal node with higher depth counting from the root of the tree. If all processors hold some subtrees from the same depth, then the improvement over those subtrees can be done independently. Since the enumeration takes almost the same time for all the nodes, the parallelism depends on the number of nodes owned by each processor. Unfortunately, the CL algorithm can produce unbalanced trees. To make a good parallelization, we have to design a good subtree partitioning scheme.

Assume there are p processors. We want to have each processor have some subtrees from the same depth and make the total node count balanced. Let n be the total number of nodes in a tree. Assume the subtree assignment occurs in depth d and there are s subtrees in this depth. Also, assume that the subtree T_v has n_v nodes. The optimal partitioning strategy will be to find a partition of s weighted objects into p subsets such that the total weight difference is minimized. Since the two set partition problem can be easily reduced to this problem, this problem is NP-complete [8]. Therefore, we propose a heuristic to give a fairly good partition. To balance the load, we use $\alpha > 0$ be the maximum percentage difference of total node counts among all processors. Assume the nodes with depth less than d to be m. Since those nodes will not be distributed, the total number of distributed nodes is $n-m$. On the average, each processor should expect to have subtrees with a total $l = (n-m)/p$ number of nodes. To make a balance assignment, we sort the subtrees in depth d in decreasing order of their node counts. The subtree assignment is processed one processor a time. If the total number of nodes assigned to the current processor does not exceed the average and adding the next subtree does not make the total node count over $(1 + .5 * \alpha)l$, then assign the next subtree to the current processor. If the current processor is under-loaded and the addition of the next subtree will make the total node count over $(1 + .5 * \alpha)$, then we do the same test on the subtree at the end of the unassigned subtree list to see if we can assign that subtree to the current processor. This process continues until all the subtrees are assigned. After the assignment, if the difference among the loads of all processor exceeds α, the partition will go the next depth of tree unless the depth is maximum already.

After the partitioning, the tree will be cut into two parts. The top part consists of nodes with depth less than d, and the bottom part consists of the rest of nodes. The improvement in the bottom part is done in a distributed manner across processors first. Then the information about subtrees will be synchronized. Finally, the improvement of the top part can be done by all the processors simultaneously. This completes one iteration of the improvement. The improvement will terminate if no source sink delay improvement occurs after this iteration. In the next iteration, since the topology is changed we have to repartition the tree and repeat this process all over again. Figure 3 illustrates the process of subtree partitioning.

4.2.1 Analysis of the Parallel Improvement Algorithm

Assume there are m nodes in the top part and it takes Q to enumerate the optimal topology for the tree T_v. The improvement of the nodes in the top part takes $O(mQ)$ time. After partitioning, each processor has at most $(n-m)(1+\alpha)/p$ nodes. The improvement of those nodes takes $(n-m)(1+\alpha)Q/p$ time. Assume it takes c to send one node information. It takes $O(c(n-m))$ time to synchronize all the improvements. In summary, the total running time of the parallel improvement algorithm is $O((mQ+(n-m)(1+\alpha)Q/p+cn)M)$. As shown by our experiments in Section 5, m is small in most the cases. Recall that the serial improvement algorithm takes $O(nMQ)$ time. Though the overhead term $O(cnM)$ could become quite significant, we can still expect a good speedup from the parallel improvement algorithm.

4.3 Parallel Zero Skew Wire-Sizing Algorithm

In the section of parallel improvement algorithm, we give a subtree partitioning scheme. This scheme can also be used in parallel zero skew wire-sizing algorithm. When doing wire-sizing the topology will not be changing, but the length of the wires or their siblings that are on the path to the root may change. Recall that in the subtree partition, the tree is partitioned into the top part and the bottom part. Only the nodes in the bottom part are distributed among the processors, and the nodes in the top part are shared among the processors. There is a wire corresponding to each node except the root node. In each iteration of the zero skew wire-sizing algorithm, we first let each processor do the wire-sizing for the top part. Then similar to the parallel improvement algorithm, each process can do the wire-sizing for all the wires in the bottom part of the tree in a distributed manner. After this we need to synchronize all the information about the roots of subtrees owned by the other processors. Unfortunately, the serial algorithm of the zero skew wire-sizing cannot be simulated by a distributed memory parallel algorithms since the length of the wire can be changed by several processors at one time. But as shown in the experimental results, for any given iteration there is no direct correlation between the source sink delay and the number of processors.

4.3.1 Analysis of the Parallel Zero Skew Wire-Sizing Algorithm

Assume there are m nodes in the top part. The zero skew wire-sizing algorithm in the top part takes $O(m\log(m))$ time. After our partitioning, each processor has at most $(n-m)(1+\alpha)/p$ wires. The improvement of those nodes takes $(n-m)(1+\alpha)\log(n)/p$ time. Assume it takes c time units to send one node information. It takes $O(cm))$ time to synchronize all the shared wires in the top part. In summary, the total running time of the parallel improvement algorithm is $O((m\log(m)+(n-m)(1+\alpha)\log(n)/p+cm)M)$. As shown by our experiments in Section 5, m is small in most the cases. Recall that the serial zero skew wire-sizing algorithm takes $O(Mn\log(n))$ time. We may get super-linear speedup from the parallel zero skew wire-sizing algorithm.

5 Experimental Results

Using the message passing interface (MPI)[12], which is portable across a wide range of parallel platforms, we implemented the parallel CL algorithm, the parallel improvement algorithm, and the parallel zero skew wire-sizing algorithm. We report results on the SPARC Server 1000E, an 8 processor shared memory multiprocessor. We report our results on benchmark circuits r4 and r5. Also, to make a projection of the advancing technology and

		r4	r5	t1	t2	t4	average
1 proc	run time (s)	9.12	19.43	80.93	196.40	331.92	
	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	1.00	1.00	1.00	1.00	1.00	1.00
2 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	2.45	2.30	1.86	2.14	1.65	2.20
4 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	5.49	5.94	3.28	4.27	3.19	4.40
8 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	10.13	9.52	8.96	6.04	4.44	7.82

Table 3: Run time and speedup results for the parallel CL algorithm.

to test the effectiveness of our parallel algorithms, we report our results on t1, t2, and t4. Clock t1, t2, and t4 consists of 10000, 20000, and 40000 randomly generated sink pins on an area of 10cm x 10cm. The unit square resistance is 0.033 Ω, and unit square capacitance is 1.9e-17F. In both the parallel improvement algorithm and parallel zero skew wire-sizing algorithm, we set $\alpha = 50\%$. All the scales are made by that of a serial run.

Table 3 lists the result for the parallel CL algorithm. It showed that in most of the cases, we obtained super-linear speedups while the quality of the results is identical to the serial run. To show the effectiveness of our subtree partition algorithm,

Table 4 lists the results of the run time and speedup of the parallel improvement algorithm using 6 iterations. Since the synchronization cost after each iteration to keep the same execution with the single processor is quite significant, we cannot get super-linear speedups. But the speedups are still significant. Again the quality of the results of the parallel improvement algorithm is exactly the same as the serial run.

Table 5 lists the results of the scaled source sink delays and the speedup of our parallel zero skew wire-sizing algorithm using 6 iterations. We do not force any synchronization between iterations. This result shows that there is no direct correlation between source sink

		r4	r5	t1	t2	t4	average
1 proc	run time (s)	17.67	27.20	90.18	181.67	392.11	
	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	1.00	1.00	1.00	1.00	1.00	1.00
2 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	1.78	1.76	1.63	1.68	1.83	1.73
4 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	3.01	2.96	3.03	2.98	3.34	3.11
8 proc	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	4.09	3.91	4.15	3.96	4.06	4.05

Table 4: Run time and speedup results for the parallel improvement algorithm.

delay and number of processors. The results of the parallel run are different from the serial run. Most of the time, the parallel execution generates shorter source sink delay in the same number of iterations. Since we do not synchronize between iteration the convergence is faster for the wire-sizing of fewer nodes. For the same reason, there are many super-linear multi-processor runs.

6 Conclusion and Future Work

In this paper, an algorithm for performing the wire-sizing of a zero skew clock tree is given using the Elmore delay model. Our experiments on benchmark clock trees show that this algorithm reduces the source sink delay more than 3 times that of the clocks with uniform wire sizes and keep the clock skew zero. Motivated by the computation intensive nature of the zero skew clock tree construction and wire-sizing, we propose a parallel algorithm based on cluster based clock tree construction algorithm and our zero skew wire-sizing algorithm.

This work may have the following extensions. After buffers are added in multi-staged clock tree generation, this scheme can be used to improve the clock delay in each stage

		r4	r5	t1	t2	t4	average
1 proc	run time (s)	61.06	27.20	90.18	181.67	392.11	
	scaled delay	1.00	1.00	1.00	1.00	1.00	1.00
	speedup	1.00	1.00	1.00	1.00	1.00	1.00
2 proc	scaled delay	0.93	0.902	0.82	0.84	1.09	0.92
	speedup	2.66	2.09	2.58	1.72	1.74	2.16
4 proc	scaled delay	1.21	1.24	0.86	0.40	0.79	0.90
	speedup	4.83	3.69	3.46	2.94	3.34	3.65
8 proc	scaled delay	1.28	1.24	0.75	0.98	1.22	1.09
	speedup	6.89	6.95	9.18	4.59	4.27	6.38

Table 5: Scaled delay and speedup results for the parallel zero skew wire-sizing algorithm. The delay is scaled by the result of one processor run.

by providing a way for effective topology construction and wire sizing. Generally, clock tree routing is done after placement and before detailed routing. The couple capacitance information is not available at this stage. Therefore, this scheme can not directly deal with coupling capacitance. But if clock tree routing is being carried out at the same time as detailed routing, it can handle the coupling capacitance easily.

References

[1] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design,* PTR Prentice Hall, Englewood Cliffs, New Jersey 07632, 1994.

[2] K. D. Boese and A. B. Kahng, "Zero-Skew Clock Routing Trees with Minimum Wire-length," *Proc. IEEE Intl. ASIC Conf.,* Rochester, NY, September 1992, pp. 17-21.

[3] T. H. Chao, Y. C. Hsu, and J. M. Ho, "Zero Skew Clock Net Routing," *Proc. ACM/IEEE DAC,* Anaheim, CA, June 1992, pp. 518-523

[4] T. H. Chao, Y. C. Hsu, J. M. Ho, K. D. Boese, and A. B. Kahng, "Zero Skew Clock Routing with Minimum Wire-length," *IEEE Trans. Circuits and Systems,* 39(1992), pp. 799-814.

[5] C. Chen and D. F. Wong, "A Fast Algorithm for Optimal Wire-Sizing under Elmore Delay Model," *Proc. IEEE ISCAS,* May 1996. pp 412 - 415.

[6] J. Cong and K. Leung, "Optimal Wire-sizing under the Distributed Elmore Delay Model," *Proc. IEEE ICCAD,* July 1993. pp. 634-639.

[7] M. Edahiro, "A Clustering-Based Optimization Algorithm in Zero-Skew Routings," *Proc. ACM/IEEE DAC,* 1993, pp. 612-616.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W. H. Freeman and Company, New York, 1979.

[9] A. B. Kahng, J. Cong, and G. Robins, "High-Performance Clock Routing Based on Recursive Geometric Matching," *Proc. ACM/IEEE DAC,* 1990, pp. 573-579.

[10] A. B. Kahng and G. Robins, *On Optimal Interconnections for VLSI*, Kluwer Academic Publishers, 1995.

[11] R. Kay, G. Bucheuv, and L. Pileggi, "EWA: Exact Wiring-Sizing Algorithm," *Proc. Intl Symp on Physical Design,* April 1997, pp 178 - 185.

[12] Message-Passing Interface Forum, "Document for a Standard Message-passing Interface," University of Tennessee, Knoxville, TN, Tech. Rep. CS-93-214, 1993.

[13] R. S. Tsay, "Exact Zero Skew," *Proc. IEEE Intl. Conf. Computer-Aided Design,* Santa Clara, November 1991, pp. 336-322