A Parallel Steiner Tree Heuristic for Macro Cell Routing

Christian Fobel and Gary Gréwal
Deptartment of Computing and Information Science
University of Guelph
Guelph, Ontario, Canada
N1G 2W1
cfobel@uoguelph.ca, gwg@cis.uoguelph.ca

***Abstract*— Global routing of macro cells remains an important but time-consuming step in the VLSI design cycle. Macro cells are large, irregularly sized parameterized circuit modules that typically contain large numbers of terminals that must be interconnected. The interconnection pattern for each set of terminals (net) that must be connected is a Steiner tree, and the primary sub-problem in the global routing of macro cells is to find a set of dissimilar, low-cost Steiner trees for each net that must be routed. In this paper, a two-phase, parallel (multiprocessor) algorithm is proposed for quickly constructing a diverse pool of high-quality Steiner trees for routing of multiterminal nets. In the first phase, a single Steiner tree is constructed using a heuristic, called *Shrubbery*. Then, in the second phase, a pool of dissimilar, high-quality trees are created from the original tree, by running multiple instances of a local search in parallel. Computational experiments performed on over 800 commonly used benchmarks show that running multiple instances of the local search in parallel results in near-linear speed-up over the serial case. Most importantly, the trees produced are both high-quality and dissimilar, allowing for numerous routing possibilities for each net.**

I. Introduction

Macro cells are large, irregularly sized parameterized circuit modules that are typically generated by a silicon compiler as per a designers selected parameters. The macro-cell design style allows for very compact and high-performance designs. However, the chip-level routing process has a much higher complexity than other methods [1]. The traditional approach to routing divides the routing into two phases. The first phase, called *global* routing, assign list of routing regions to each net without specifying the actual geometric layout of the wires. The second phase, called *detailed* routing, determines the actual geometric lay-out of each net within the assigned regions.

Many global routing methods can be found in [2]. Usually graph-based methods are used for global routing problems. The routing model is based on a routing graph, which is extracted from the given placement and the routing-region definitions. In practice, channel graphs [3] have been found to provide the most general and accurate model. Given a layout or placement, the edges of the graph correspond to future routing regions, while the vertices of the graph correspond to the intersection of routing regions. In order to ease the task of detailed routing, several optimality criteria may be used by the global router, such as minimizing the total wire length, minimizing the total area, minimizing local densities, etc. Therefore, every edge in the graph is assigned one or more cost values, typically representing the length of the associated routing regions and/or an upper bound on the number of connections that can pass through the region. To compute a global route for a specific net, vertices representing the terminals are added at appropriate locations. Finding a global route then becomes equivalent to finding a minimum-cost subtree in the routing graph that spans all of the terminal vertices. For two-terminal nets, the problem reduces to the shortest-path problem. However, for multiterminal nets, the problem has traditionally been viewed as a minimum Steiner tree Problem in a Graph (SPG).

Given n points (terminals) on a graph, a Steiner tree connects these points through some extra (Steiner) points such that the total length (cost of edges) of the tree is minimized. The global (macro-cell) routing problem [2] can be viewed as a problem of finding a set of Steiner trees for each net in the routing graph. More than one candidate Steiner tree is required for each net, as the capacity of the edges in the graph must not be violated. If the capacity constraints of some edges cannot be satisfied, routes must be ripped up and then rerouted using alternate Steiner trees compatible with the Steiner trees used for connecting other nets. Unfortunately, finding even a single high-quality Steiner tree for each net can be time-consuming because the minimum SPG problem is NP-hard [4]; therefore, all existing solution methodologies are heuristics. The focus of this paper is on the macro-cell routing problem [2]. The main aim of the paper is to develop an efficient parallel algorithm to generate a set of Steiner trees for each net.

The remainder of this paper is organized as follows. Section II presents related work. Our parallel approach is given in Section III. We introduce the graph terminology that will be used throughout the remainder of the paper in Section IV. Shrubbery, and Insert are described in Sections V, and VI, respectively. In Section VII we present our results, followed by our conclusions in Section VIII.

II. Related Work

The primary subproblem in macro-cell routing is to find a minimal cost Steiner tree for a net. The Steiner problem in graphs can be formulated as an integer or a continuous non-convex optimization problem. Many exact algorithms for small size problems are based on these formulations [5].

978-1-4244-2658-4/08/$25.00 ©2008 IEEE

Several heuristics are available for the approximate solution of SPG; see Duin and Voss[6] for recent surveys. Constructive methods that build feasible solutions from scratch have been proposed by Takahashi and Matsuyama [7], Plesnik [8], Minoux [9], and Rayward-Smith and Clare [10]. Node-based [9], path-based [11] and spanning-tree based [5] *improvement* heuristics have also been proposed for solving the SPG. More recently, we find implementations of meta-heuristics including simulated annealing [12], genetic algorithms [13] [14], tabu search [15], and GRASP [16] [17]. For surveys on heuristic techniques for solving the SPG see Voss [18], Hwang, et. al [19], and Duin and Voss [6].

III. Our approach

Our goal is to develop an algorithm that is fast enough to generate not one, but an entire pool of high-quality trees. Our proecedure consists of two separate phases: a construction phase, followed by an improvement/diversification phase (see Fig. 1).

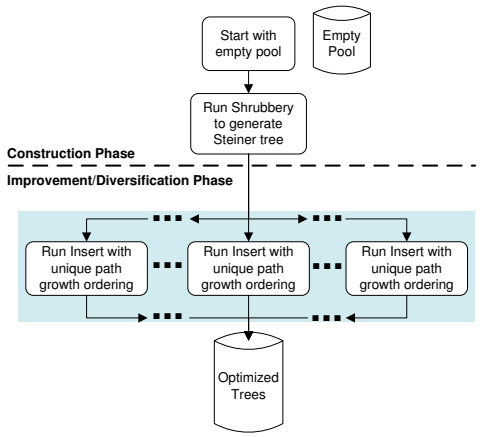

Fig. 1: Approach taken to generate pool of Steiner trees.

The goal of the construction phase is to use our constructive heuristic, Shrubbery, to construct a single minimal cost Steiner tree for each net. Unlike previous heuristics, Shrubbery builds a Steiner tree much like shrubs grow in a garden. Shrubbery begins the construction of a Steiner tree by simultaneously growing separate shortest-path trees rooted at every terminal vertex. Each tree is called a "shrub". These trees grow towards each other in a coordinated manner. Whenever nearby trees touch each other for the first time, they join the same "grove". Their connecting paths are merged into a "hedge" within the grove. A hedge is a tree (in a graph) that connects all roots (terminals) belonging to the same grove. Eventually all the shrubs become connected and belong to a single grove. Each "dangling" branch is pruned, and the remaining hedge is a Steiner tree.

Although conceptually simple, Shrubbery is able to find high-quality solutions to very large problem instances in small amounts of time. Most importantly, Shrubbery has running time complexity of $O(|E|\log|V|)$, with a constant no larger than 2.

In the diversification phase, local search is used to modify the Steiner tree generated in the construction phase to obtain numerous unique, high-quality trees. We have refined a local search strategy, called *Insert*, that iteratively replaces an existing (*key*) path in the tree with a new path of lesser cost. The behaviour of the Insert algorithm is deterministic, based on the order in which it evaluates the nodes in the graph. Diverse solutions are generated by running Insert multiple times with different node traversal orderings. Our results show that Insert can further reduce the cost of the seed Steiner tree, while introducing diversity among tree pool members. We employed the most obvious form of parallelism scheme by distributing the repetitions of Insert among the processors (see shaded region in Fig. 1). In a homogenous, parallel environment each processor performs a fixed number of Insert executions equal to the number of trees required for a net divided by the number of available processors. Once all processors are finished their computations, the entire pool of solutions are now available for routing the net.

IV. Terminology

Throughout the remainder of this paper, we use the following definitions and terminology when describing Shrubbery and Insert. The following symbols index scalar objects or sets of scalar objects: i and j are used to index vertices; a and b are used to index root vertices of shrubs; thus shrub, grove, or hedge identifiers. An edge (e_{ij}) in a graph is identified by the pair of vertices v_i and v_j it connects. The cost or weight of an edge e_{ij} in a graph is identified by w_{ij}. We also make use of the following definitions (see Fig. 2 for assistance):

Definition 1:

A **shrub** S_a consists a set of edges S_a^e and a set of vertices S_a^v where v_a is the only terminal in S_a^v, the edges in S_a^e form a tree rooted at v_a, meeting all vertices of S_a^v, and all $e_{ij} \in S_a^e$ have end vertices $v_i, v_j \in S_a^v$. Once an edge or a vertex is included in a shrub, its shrub membership does not change. The function $s(v_i) = undefined$, if vertex v_i does not belong to any shrub. Otherwise, $s(v_i) = a$, the root of the containing shrub.

Definition 2:

A **path** P_{ij} is a sequence of distinct edges connecting vertices v_i and v_j. Two paths P_{ij} and P_{kj} with no common vertices except for vertex v_k can be concatenated to form a single, longer path; that is, $P_{ik} \| P_{kj}$ is the union of distinct edges and vertices in P_{ik} and P_{kj}. A path P_{ai} represents the unique path by which vertex v_i was reached from root vertex v_a during shrub growth. The function $N(P_{ai})$ returns a list of all vertices in the path P_{ai}. The function $E(P_{ai})$ returns a list of all edges in the path P_{ai}.

Definition 3:

A **grove** G_a is a union of shrubs $\bigcup_{b \in M} S_b$ where M is an index set of member shrubs and $a = \min_{\mathrm{k}} \{k \in M\}$. More specifically, $G_a^e = \bigcup_{b \in M} S_b^e$ and $\bigcup_{b \in M} S_b^v$.

Definition 4 (hedge):

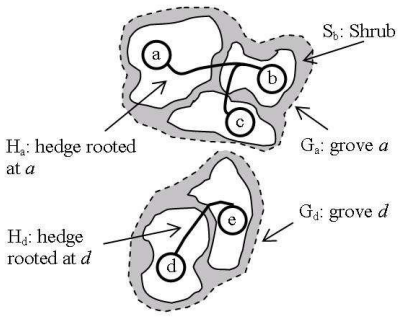

Fig. 2: Illustration of shrub, grove, and hedge.

A **hedge** $H_a$is the partial Steiner tree that connects the terminals of G_a. H_a consists of two sets $H_a^e \subseteq G_a^e$ and $H_a^v \subseteq G_a^v$.

Definition 5 (root distance):

Every **vertex** v_k within a shrub S_a has a root-distance d_i^a, which is the cost of the path by which vertex v_k was reached from root of S_a (vertex v_a), during shrub growth; that is, $d_k^a = \sum_{e_{ij} \in P_{ak}} w_{ij}$, where is the cost of edge.

Every vertex within a shrub except the root vertex has a parent. Let's suppose vertex v_i is vertex v_j's parent, the following relationship exists: $d_j^a = d_i^a + w_{ij}$, where w_{ij} is the cost of edge e_{ij}. To simplify the description in later sections, if vertex v_j is not in S_a, but it is adjacent to vertex v_j which is within S_a, vertex v_j also has a root-distance to shrub S_a through vertex $v_i : d_j^a = d_i^a + w_{ij}$.

V. Shrubbery

Shrubbery starts a Steiner tree construction by simultaneously growing individual shortest-path trees rooted at every terminal vertex. We refer to these trees as "shrubs". Shrubs grow using a modified version of Dijkstras shortest-path algorithm. Initially, each shrub consists of a single terminal vertex, its root. Then, shrubs are extended by adding one edge and vertex to a single shrub at each step of the algorithm. Shrubbery searches for a global optimum by coordinating growth among the shrubs. At every step, each shrub has one nearest adjacent vertex not connected by a path to this shrub and having the minimum-cost path to the shrubs root. Shrub growth is coordinated by selecting the shrub whose nearest vertex has the globally smallest path length to its root. The winning shrub expands to include its nearest vertex. That shrub will then determine its next nearest vertex and becomes a candidate for the next round. (Ties can be handled by any consistently applied rule.) When unrelated shrubs meet during this process, they merge into a containing "grove", a tree of adjacent shrubs.

During the growth, edges connecting vertices that belong to the same grove are ignored, so that shrubs in the same grove do not meet again. Consequently, the meeting of shrubs is the meeting of different groves containing them. When two

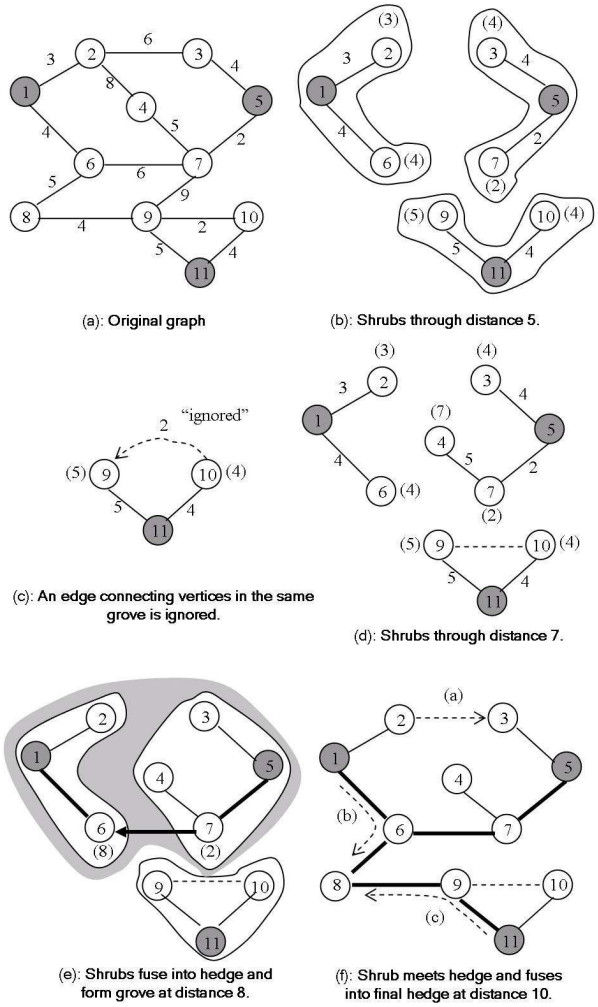

Fig. 3: Illustration of shrub growth.

shrubs meet, two groves also meet, and two things happen. First, the two groves merge into one. Second, the paths from the two shrub roots to the contact node become "hedges" that connect existing hedges within the two participating groves. Within each grove, all member shrubs are connected, and different shrubs have at most one distinguished path (hedge) between them. That path, the hedge, constitutes an internal Steiner tree connecting all the terminals (the roots of shrubs) within the grove. When all terminals eventually belong to a single grove, the final hedge constitutes the desired Steiner tree. An important feature of hedge growth is that part (or all) of the paths from the respective terminals may already be hedges. This means that a new shrub may attach itself to an internal path of a Steiner tree and not directly to a terminal.

A. An example

The entire process is illustrated in Fig. 3. The original problem instance is shown in Fig. 3(a). The shaded vertices (1, 5, and 11) are terminal vertices, and Shrubbery starts by growing shrubs rooted at each of these three vertices.

Fig. 3(b) shows the growth of the shrubs at a maximum distance of five from each terminal. (Numbers in parentheses indicate the shortest distance of the vertex from the root of the shrub.) Notice that at a distance of five, each shrub has grown to include three vertices. Any edge connecting two vertices belonging to the same grove is ignored (Fig. 3(c)). At a distance of seven (Fig. 3(d)), only the shrub rooted at vertex 5 is able to grow.

Fig. 3(e) shows what happens when two shrubs meet. At a distance of eight, the shrub rooted at vertex 5 is able to reach vertex 6, which is part of another shrub - and another grove. The two shrubs become part of the same grove. Within the grove, the connecting path (bold line) forms a single hedge. This hedge is now guaranteed to be part of the "backbone" of the final Steiner tree. However, it still has the potential to expand and grow in parallel with the remaining shrub rooted at vertex 11.

The final Steiner tree is eventually formed when the remaining structures are allowed to expand to a distance of nine (Fig. 3(f)). At this distance, several things occur. First, shrubbery attempts to grow an edge from vertex 2 to vertex 3. However, this edge is discarded as vertex 2 and vertex 3 are now part of the same grove. This action prevents a cycle from forming. Next, shrubbery grows an edge from vertex 6 (which is part of the hedge) to vertex 8. Then the shrub rooted at vertex 11 reaches vertex 8. The result is that the hedge and shrub meet at vertex 8 and are "fused" together to form a final hedge (bold line) and a single grove. In general, when a hedge and shrub (or two hedges) are fused together, only those vertices and edges that do not already belong to any structure will be added as new hedge elements. Each fusion will start from the encountering vertex and move towards the terminal nodes of the structures involved along the two shortest paths. The expansion in each direction will stop at the first vertex contained in the hedge, or else at the terminal vertices.

Returning to our example, we note that all terminal vertices (1, 5, and 11) are now contained in one hedge, and all shrub growth ceases. As a final step, all non-terminal vertices not on a hedge will be pruned from the hedge, which leaves a Steiner tree.

B. Shrubbery algorithm

The shrubbery algorithm is presented in Algorithm 1. The algorithm begins with a search for the edge e_{ij} whose weight, w_{ij}, extends the shrub S_a to which x_i belongs by the smallest distance from the root, r_a (line 2). Once this edge has been determined, the original shrub (S_a) expands to include both the new edge and the vertex if the vertex that edge e_{ij} meets (x_j) does not already belong to a shrub. Moreover, the unique path from the root of the shrub (r_a) to vertex x_j is recorded (line 4), and vertex x_j is made an official member of the shrub S_a (line 5). Observe (line 6) that the distance from root r_a to vertex v_j is simply the total distance from the root of the shrub to vertex x_i (d_{ai}) plus the weight (w_{ij}) of the new edge.

Algorithm 1 Shrubbery algorithm.

1: **repeat**
2: Let e_{ij} satisfy $\forall\ T$, $\min_{i,j}(d_{ai}+w_{ij} : x_i \in S_a, x_j \notin S_a)$
3: **if** $x_j \notin S_b$, $\forall\ b$ in T add x_j and e_{ij} to S_a **then**
4: $P_{aj} = P_{ai} \parallel P_{ij}$
5: $S(x_j) = a$
6: $d_{aj} = d_{ai} + w_{ij}$
7: **else if** $x_j \in S_b$ **then**
8: **if** $S_a \in G_a$ and $S_b \in G_a$ **then**
9: mark e_{ij} ineligible (to prevent cycle)
10: **else**
11: $G_a^n = G_a^n \cup G_v^n$
12: $G_a^e = G_a^e \cup G_v^e \cup \{e_{ij}\}$
13: $H_a^n = H_a^n \cup H_b^n \cup N(P_{ai}) \cup N(P_{bj})$
14: $H_a^e = H_a^e \cup H_b^e \cup E(P_{ai}) \cup E(P_{bj}) \cup \{e_{ij}\}$
15: discard G_b and H_b
16: **until** G_a contains all m terminals

If, however, the met vertex x_j is found to be part of another shrub S_b (line 7), two possibilities exist. Either x_i and x_j belong to shrubs in the same grove, or they belong to shrubs in different groves. The former case implies that both shrubs (S_a and S_b) are already connected by a hedge and, therefore, the edge in question (e_{ij}) is not eligible for further consideration (lines 8 and 9). The latter case implies that the edge (e_{ij}) has caused two groves (G_a and G_b) to meet for the first time. The effect is that both groves will merge to form a single grove (lines 11 and 12), and their respective hedges will be connected by a new hedge segment. The new hedge will form to include the path containing e_{ij}, extended to existing hedges in each grove. The new hedge will consist of the union of all the nodes (line 13) and edges (line 14) involved. Once the new hedge (H_a) and grove (G_a) are formed, the old hedge (H_b) and grove (G_b) can be discarded (line 15), since their members will have been absorbed into H_a and G_a.

In practice, steps 1 through 15 of the algorithm are repeated until all m terminal vertices are contained in a single grove (G_a). The hedge within this grove represents the final Steiner tree.

C. Implementation and time complexity

To achieve an efficient implementation for Shrubbery, all edges and vertices are stored in separate Fibonacci heaps. We also make use of the union-find data structure when determining whether a newly encountered vertex (during shrub growth) belongs to a different grove or is simply (another) part of the existing grove. In Algorithm 1, the operations in lines 3-9 and 15 take O(1) time. Therefore, the time complexity of the algorithm is dominated by the time spent performing the operations in lines 1, 2, and 10-14. Since the time for finding an element with the smallest key value from a heap is $O(\log n)$, a single execution of line 2 takes $O(\log|V|)$ time. Using union-find to perform the operations in lines 11-14, the resulting operations each take $O(\log|V|)$

time. The most expensive step in the algorithm is the first (line 1), which requires at most $|E|$ iterations. As each iteration of the do-until loop takes $O(\log|V|)$ time, we can see that Shrubbery has a time complexity of $O(|E|\log|V|)$. This is the same worst-case run-time complexity as in SPH [7], which is the lowest for the SPG.

VI. Insert

When a Steiner tree is constructed using a heuristic algorithm (like Shrubbery), there is no guarantee that the total cost of the tree is minimal. Consequently, local search can often be used to improve the quality of the tree. We employ a local improvement algorithm called Insert. The basic idea is to grow a new path from each vertex in the current tree. Paths are grown in a fashion similar to that for shrubs, with the closest vertex not already part of the existing tree being added to the expanding path at each step. If the new path meets the tree again, it will form a cycle. The cost of the new path can then be compared with the cost of the other path segments that constitute the remainder of the cycle. If the new path is of lower cost than some existing path segment, it can be inserted into the tree, replacing an existing path segment with greater cost.

A key path has intermediate vertices that are Steiner vertices with degree two, and the vertices at its ends are either terminal vertices or key vertices. In general, it is possible for a cycle to consist of many key paths (and one or two segments of key paths), in which case the one that provides the maximum improvement will be selected for replacement. Similarly, it is possible for the local neighbourhood surrounding each vertex to contain multiple cycles. However, only the cycle that provides the greatest cost improvement is selected.

The Insert algorithm employs a first-improving strategy in which path replacements are made as soon as an improving path is found. Growth of an individual path from a root will stop when a cycle is formed or a limit is exceeded. The limit is the total weight (w_{max}) of the key path having maximum total weight in the current tree. At each step, the neighbourhood search function computes the cost (w') of the expanding path from the newly added node to the root. If $w' \geq w_{max}$, this new path could not possibly yield any cost improvement, because its cost exceeds or equals that of the maximum unit that could be replaced in the current tree.

In addition to improving a "seed" Steiner tree, Insert can be used to generate numerous unique solutions by varying the order in which paths are grown from vertices. By exploiting this behaviour, Insert is able to generate a diverse set of high-quality solutions, based on a single "seed" Steiner tree. Since each repetition of Insert is independent, we propose an implementation where several executions of Insert are run in parallel on multiple homogenous processors. Using this multi-processor implementation we achieve near-linear speed-up over the serial execution of the same Insert runs, as shown in Section VII.

TABLE I: Classes of problem instances from SteinLib[20].

Class	Series	Description
Random	b, c, d, e, mc, p6z	Random graphs with random weights
FST	es*fst, tspfst	Rectilinear graphs
VLSI	dmxa, diw, gap, lin, msm, taq	Grid graphs with holes
Incidence	i080, i160, i320, i640	Random graphs with incidence weights
Euclidean	x, p6e	Graphs with Euclidean weights
Hard	sp, puc	Artificial, hard instances

VII. Results

In this section, we report on results obtained by our method with special emphasis placed on run-time, tree cost, and diversity among pool members. In the experiments that follow, diversity among pool members is assessed with the following measure:

$$D = \frac{\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1-(com(E_i,E_j)/max(E_i,E_j))}{(n(n-1))/2} \quad (1)$$

where n is the number of trees in the pool, $com(E_i,E_j)$ is the percentage of edges that two trees (i and j) share in common, and $max(E_i,E_j)$ is the maximum number of edges in either tree. Hence, pool diversity is simply the average (expressed as a percentage) of the dissimilarities of all possible pairs of trees in the pool. The higher the value of D, the more diverse the trees in the pool. Our method was tested on over 800 commonly used problem instances already available at the SteinLib [20] repository. One of the main advantages of using this test suite is that it facilitates comparison with global optimum solutions (or best known solutions). We wish to emphasize that none of the graphs used in our experiments were reduced by means of the reduction tests, as in [21]. Otherwise, the times given in the experiments that follow would be further reduced.

To simplify the analysis, we organized the various problem instances into the six classes listed in Table I. A complete description of each class can be found in [20].

Both Shrubbery and Insert were implemented in C++ using HP-MPI and compiled under Linux with PathScale EKOPath™compiler version 2.2.1. Run-times were obtained on a cluster of systems connected through a Myrinet 2g (gm) interconnect, each node having four 2.2 GHz Intel Opteron processors and 8GB of RAM.

A. Parallel Run-time Performance

For each problem instance, Insert was run 32 times, each time using a different order of path growth, resulting in up to 32 unique solutions. To evaluate the run-time performance of our parallel implementation, the 32 runs of Insert were run on a varying number of processors: 1, 2, 4, 8, and 16. Table II reports the average run-times for each problem class across 1 to 16 processors. Fig. 4 shows the run-times of all

TABLE II: Average Insert run-time (per tree) across varying number of processors.

Set Name	Run-time (s) on N Processors				
	1	2	4	8	16
Random	0.7058	0.3541	0.1626	0.0949	0.0577
Fst	1.4865	0.7628	0.4135	0.2026	0.1069
VLSI	0.2103	0.0925	0.0643	0.0336	0.0174
Incidence	7.5801	3.8263	2.3671	1.0996	0.8524
Euclidean	1.2347	0.6018	0.3040	0.2199	0.1164
Hard	0.2809	0.1369	0.0777	0.0449	0.0259

instances within each problem class averaged. Note that the run-times are normalized by dividing by the serial (single-processor) run-time for each problem class. For all problem classes, a near-linear speed-up trend can be observed with an increase in the number of processors.

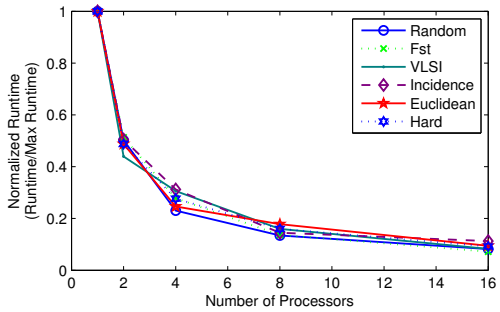

Fig. 4: (Normalized) Average Insert run-times across varying number of processors.

B. Diversity and Quality of Parallel Solutions

In addition to run-time averages, solution costs were recorded for the tree pools generated for each problem instance. The average diversity metric of each problem class was calculated according to (1). Table III shows the relative error between the value of the solution found and the optimal (or best known) solution from SteinLib[20], after the construction phase and improvement/diversification phase. Table III also shows the number of unique trees generated by 32 distinct Insert path growth orderings, along with the average solution set diversity.

As shown in Table III, an average diversity of at least 20% was achieved after the diversification phase for all tested problem classes. Also, it can be seen that Insert is able to significantly reduce the cost of the average solution for each pool. For most problem classes, the average cost of pool members is within less than 5% of the best known cost.

C. Run-time and Solution Quality Comparison with SPH [7]

To further evaluate the efficacy of Shrubbery (with and without Insert), we compare the run-time and solution quality obtained by running one serial instance of Shrubbery against the results of our implementation of the *Shortest Path Heuristic* (SPH), developed by Takahashi et. al. [7]. Here

TABLE III: Pool diversity with best-case, worst-case and average Steiner tree solution costs.

	Shrubbery		Insert					
Set Name	[a]T	C_a	T	N	D	C_b	C_w	C_a
Random	0.027	9.4%	0.706	18	26.2%	1.1%	4.3%	2.8%
Fst	0.009	4.2%	1.487	26	20.9%	0.5%	1.5%	1.0%
VLSI	0.023	6.0%	0.210	29	32.2%	0.7%	3.2%	1.9%
Incidence	0.087	45.5%	7.580	27	33.1%	8.9%	17.1%	12.9%
Euclidean	1.420	11.4%	1.235	31	29.3%	2.6%	6.8%	4.6%
Hard	0.029	38.1%	0.281	29	37.1%	11.8%	15.8%	13.8%

[a] Note: T is the average run-time (seconds) per generated tree; N is number of unique trees generated by 32 distinctly ordered Insert runs; C_b, C_a, and C_w are best, average and worst trees variance from optimal cost, respectively; D is diversity among pool members.

only one Steiner tree is being produced by each algorithm. Both the serial Shrubbery/Insert and the SPH algorithms were implemented in C++ and compiled under Linux with the GCC compiler version 2.96. Run-times were obtained on a system with a 2.4 GHz Intel Pentium 4 processor and 512MB of RAM. Table IV shows the average run-time and percentage variance from the optimal solution cost, across all instances in each test data-set. Several VLSI representative data-sets were chosen from SteinLib[20] for comparison.

As Table IV shows, on average for the selected benchmarks, Shrubbery achieves solution quality within 5.99% of the optimal cost, while SPH performs within 3.49% of optimal. However, Shrubbery is about two orders of magnitude faster. Moreover, when Insert is applied after Shrubbery, the run-time still remains approximately an order of magnitude faster than SPH, while Shrubbery with Insert outperforms SPH by about 2% in terms of solution quality.

VIII. Conclusion

Global routing of macro cells remains an important, but time-consuming, step in the VLSI design cycle. In this paper, we have addressed this issue by presenting a new muli-processor algorithm for quickly constructing a diverse pool of Steiner trees suitable for routing multi-terminal nets. The main idea behind the algorithm is its unique two-phase approach to tree construction, with the second phase exploiting parallelism to produce a pool of trees in roughly the same time as generating a single tree.

The performance of the algorithm has been tested on over 800 graphs with up to 38418 vertices, 221445 edges, and 11849 terminals. The experimental results show that, on average, the best tree found in a pool of up to size 32 is within 3% of the global optimum for most problem classes, while the average tree cost is found to be within 5% of the optimum for most problem classes. Although the run-times vary from series to series, they are uniformly low; typically a fraction of a second per tree for Shrubbery, and slightly longer for Insert. In particular, in Section VII-C, a comparison is made between our serial Shrubbery/Insert implementation and SPH [7] in terms of solution quality and run-time. On average, Shrubbery with Insert achieves a cost approximately 2% better than SPH, while running an order of magnitude faster.

TABLE IV: Comparison of run-time and solution quality between Shrubbery and *Shortest Path Heuristic* (SPH) [7].

		SPH		Shrubbery		Shrubbery+Insertion	
Series	**Instances**	**Avg. Time(s)**	**Avg. Gap (%)**	**Avg. Time(s)**	**Avg. Gap (%)**	**Avg. Time(s)**	**Avg. Gap (%)**
DIW	21	3.00	2.53	0.02	4.36	1.09	0.84
DMXA	14	0.20	4.14	0.01	7.26	0.06	2.03
GAP	13	0.59	4.42	0.01	5.75	0.43	1.42
MSM	30	0.38	2.66	0.01	5.49	0.17	1.37
TAQ	14	0.71	2.98	0.01	6.36	0.16	1.54
LIN	37	57.70	4.20	0.12	6.73	6.18	1.48
Average		10.43	3.49	0.03	5.99	1.35	1.45

Our parallel implementation of the diversification phase shows near-linear speed-up in relation to the number of processors. The communication overhead for splitting up the parallel work is insignificant compared to the computational time required for Insert. Thus, our parallel implementation will scale well to many processors and large tree pool sizes. Most importantly, while run-times are kept small, the trees in the generated pool are highly dissimilar. The average dissimilarity among pool members was found to be approximately 30%, thus providing for a variety of possible routing options.

REFERENCES

[1] L.-C. E. Liu and C. Sechen, "Multilayer chip-level global routing using an efficient graph-based Steiner tree heuristic." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 10, pp. 1442–1451, 1999.

[2] N. A. Sherwani, *Algorithms for VLSI Physical Design Automation, 3rd edition*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[3] C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*. Kluwer Academic Publishers, 1988.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, January 1979.

[5] M. P. de Aragão and R. F. F. Werneck, "On the implementation of MST-Based heuristics for the Steiner problem in graphs," in *ALENEX*, ser. Lecture Notes in Computer Science, D. M. Mount and C. Stein, Eds., vol. 2409. Springer, 2002, pp. 1–15.

[6] C. Duin and S. Voß, "Efficient path and vertex exchange in Steiner tree algorithms," *Networks*, vol. 29, no. 2, pp. 89–105, 1997.

[7] H. Takahashi and A. Matsuyama, "An approximate solution for the Steiner problem in graphs," *Math. Japonica*, vol. 24, pp. 573–577, 1980.

[8] J. Plesnik, "A bound for the Steiner tree problem in graphs," *Mathematica Slovaca*, vol. 31, pp. 155–163, 1981.

[9] M. Minoux, "Efficient greedy heuristics for Steiner tree problems using reoptimization and supermodularity," *INFOR*, vol. 28, pp. 221–223, 1990.

[10] V. Rayward-Smith and A. Clare, "On finding Steiner vertices," *Networks*, vol. 16, pp. 283–294, 1986.

[11] M. S. M. Verhoeven and E. Aarts, *Local Search for Steiner Trees in Graphs*, V. Rayward-Smith, I. Osmam, C. Reeves, and G. Smith, Eds. Wiley, 1996.

[12] K. Dowsland, "Hill-climbing simulated annealing and the Steiner problem in graphs," *Engineering Optimization*, vol. 17, pp. 91–107, 1991.

[13] H. Esbensen, "Computing near-optimal solutions to the Steiner problem in a graph using a genetic algorithm," *Networks*, vol. 26, pp. 173–185, 1995.

[14] A. Kapsalis, V. Rayward-Smith, and G. Smith, "Solving the graphical Steiner tree problem using genetic algorithms," *Journal of Operational Research Society*, vol. 44, pp. 397–406, 1993.

[15] M. Bastos and C. Ribeiro, "Reactive tabu search with path-relinking for the Steiner problem in graphs," in *Essays and Surveys in Metaheuristics*, C. Ribeiro and P. Hansen, Eds. Kluwer Academic Publishers, 2001, pp. 39–58.

[16] S. L. Martins, C. C. Ribeiro, and M. C. Souza, "A parallel GRASP for the Steiner problem in graphs," in *Workshop on Parallel Algorithms for Irregularly Structured Problems*, 1998, pp. 285–297.

[17] S. Martins, P. Pardalos, M. Resende, and C. Ribeiro, "Greedy randomized adaptive search procedures for the Steiner problem in graphs," 1999.

[18] S. Voß, "Steiner's problem in graphs: Heuristic methods," *Discrete Applied Mathematics*, vol. 40, no. 1, pp. 45–72, 1992.

[19] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner Tree Problem (Annals of Discrete Mathematics)*. North-Holland, 1992.

[20] T. Koch, A. Martin, and S. Voß, "SteinLib: An updated library on Steiner tree problems in graphs," Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, Berlin, Tech. Rep. ZIB-Report 00-37, 2000.

[21] E. Uchoa, M. P. de Aragão, and C. C. Ribeiro, "Preprocessing Steiner problems from VLSI layout," *Networks*, vol. 40, no. 1, pp. 38–50, 2002.