INTEGRATION, the VLSI journal 47 (2014) 105–114

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Obstacle-avoiding rectilinear Steiner tree construction in sequential and parallel approach

Wing-Kai Chow [a,*], Liang Li [a], Evangeline F.Y. Young [a], Chiu-Wing Sham [b]

[a] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[b] Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Hong Kong

ARTICLE INFO

Article history:
Received 10 July 2012
Received in revised form
19 March 2013
Accepted 9 August 2013
Available online 19 August 2013

Keywords:
Obstacle-avoiding maze routing
GPU
Parallel computing

ABSTRACT

The Rectilinear Steiner Minimum Tree (RSMT) problem is a fundamental one in VLSI physical design. In this paper, we present a maze routing based heuristics to solve the obstacle-avoiding RSMT (OARSMT) problem. Our approach can handle multi-pin nets in good quality and reasonable running time. We also present an implementation of the heuristics in parallel approach with the aid of graphic processing units (GPU). The parallel algorithm is implemented by using CUDA and has been tested on a NVIDIA graphic card. Our experimental results show that our parallel algorithm has promising speedups over our sequential approach. This work demonstrates that we can apply a parallel algorithm to solve the OARSMT problem with the aid of GPU.

1. Introduction

Rectilinear Steiner Minimum Tree (RSMT) problem is one of the fundamental problems in VLSI physical design. In floorplanning and placement stages, RSMT can be applied to the construction of routing topology. The routing topology helps us to estimate the wire-length, congestion and timing. This routing topology can also be used as the initial net topology in detailed and global routing stages. The RSMT problem has been shown to be NP-complete [1] and has attracted a lot of attentions in research. The original problem assumes that there is no obstacle in routing regions. However, modern VLSI designs often contain many obstacles like macro cells, IP blocks, and pre-routed nets. Therefore, the obstacle-avoiding RSMT (OARSMT) is a more practical problem and it becomes a popular research topic in recent years. Since the RSMT problem is NP-complete, the existence of obstacles even increases the problem complexity. Modern ICs can be composed of several billion transistors. The increasing complexity of the VLSI design has generated some interests in parallel algorithm approaches. However, most of the algorithms used in electronic design automation (EDA) tools are sequential. The data and control dependency cannot be resolved efficiently. In recent years, the use of general purpose computing on graphic processing units (GPU) has become popular due to its low-cost and high-performance. In order to deal with the complexity issue of OARSMT, the application of GPU is a potential solution.

There were a lot of research works focusing on the RSMT and OARSMT problems. Clarkson et al. [2] considered only 2-pin nets and presented an algorithm with complexity $O(n(\lg n)^2)$ to compute a rectilinear shortest path between two pins avoiding polygonal obstacles, where n is the number of pins and obstacle boundaries. Wu et al. [3] introduced a sparse connection graph, called track graph, to reduce the search space for computing the shortest path between two points avoiding obstacles. Ganley and Cohoon [4] proposed an algorithm to construct an optimal 3-terminal or 4-terminal OARSMT. The heuristics, called G3S, G4S and B3S, were developed respectively for the cases with less than 20 terminals. Zheng et al. [5] proposed an algorithm to find obstacle-avoiding shortest paths using an implicit connection graph approach. Yang et al. [6] presented a complicated 4-step heuristics to solve the OARSMT problem. Their approach works well when the terminal number is less than seven and the obstacles are convex. Hu et al. [7] developed an efficient hierarchical heuristics, called FORst. The OARSMT is constructed using a connection graph based approach. Their method can tackle large scale problems efficiently. Based on an ant colony optimization, Hu et al. [8] proposed another non-deterministic local search heuristics, called An-OARSMan. It can handle small-scale OARSMT problems with complex obstacles of both concave and convex shapes. Although An-OARSMan is more flexible in handling complex obstacles, it takes extremely long run-time for large scale designs. Later, CDCTree is proposed by Shi et al. [9]. It is based on a current-driven circuit model and this method can achieve shorter wire-length than An-OARSMan.

* Corresponding author. Tel.: +852 95572856.
E-mail addresses: wkchow@cse.cuhk.edu.hk (W.-K. Chow), lli@cse.cuhk.edu.hk (L. Li), fyyoung@cse.cuhk.edu.hk (E.F.Y. Young), encwsham@poly.edu.hk (C.-W. Sham).

0167-9260/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.vlsi.2013.08.001

Shen et al. [10] proposed a connection graph based approach to solve the OARSMT problem. Feng et al. [11] proposed a method to construct obstacle-avoiding Steiner tree in an arbitrary λ−geometry by Delaunay triangulation. Its running time complexity is just $O(n \lg n)$, where n is the total number of terminals and obstacles. Wu et al. [12] presented that an approach first finds a minimum spanning tree of all the terminals by applying a partitioning method. The segments intersecting with the obstacles are removed, forming a set of sub-trees. An ant colony optimization based approach is then used to connect the sub-trees back into a single tree which is rectilinearized at the end to give an OARSMT. Hentschke et al. [13] presented AMAZE, a fast maze routing based algorithm to build Steiner trees. However, the algorithm adopts traditional ways of selecting just one path from multiple paths with a biasing technique when dealing with multiple paths. Lin et al. [14] extended the connection graph based approach in [10] by identifying many "essential" edges which can lead to more desirable solutions in the construction of the obstacle-avoiding spanning graph. They have also improved the OARSMT transformation procedure in [10] significantly. They developed an effective refinement scheme for the U-shaped connections in an OARSMT to further reduce the total wire-length. Long et al. [15] proposed an efficient four-step algorithm to construct an OARSMT. They presented a fast algorithm for the minimum terminal spanning. Li and Young [16] proposed a maze-routing based method on extended Hanan grid, and the method achieves the highest solution quality among the others' works. Liu et al. [17] proposed an efficient path-based framework for the OARSMT construction problem, and it brings about an $O(n \log n)$−time algorithm. Later, Ajwani et al. [18] proposed a top down partitioning approach with use of OASG, and constructs OARSMT using obstacle-aware version of FLUTE. Most recently, Huang and Young [19] proposed an exact algorithm for optimal OARSMT construction and this algorithm can handle complex rectilinear obstacles.

It is commonly believed that the approaches based on maze routing are suitable for small scale problems only. The major drawback is the lack of a multi-terminal variant to handle multi-pin nets. Besides, its time complexity and memory usage can grow very large when the routing area expands. In this paper, we show that maze routing based approaches can also handle large scale OARSMT problems effectively. In our algorithm, in order to handle multi-pin nets, multiple candidates of the shortest path between the pins are kept until all the pins are reached. A graph that is composed of all candidates of the shortest path is formed and the MST of the graph is constructed to create an OARSMT. A post-processing step is then performed to further reduce the total wire-length. A maze router is implemented efficiently by using a heap data structure and propagating on the simplified Hanan grid. This simplified Hanan grid is formed by the original Hanan grid with all the intersection points lying inside an obstacle deleted. In addition, we facilitate the obstacle-avoiding escape graph into a regular array-based data structure. A parallel approach is proposed with applying the multi-pin maze routing algorithm which can be efficiently executed on GPU. Although applying GPU in various computing applications has shown exciting speedup, the use of GPU in EDA is not as popular as the other fields. They only included VLSI design verification [20], floorplanning [21], and power grid analysis [22]. With the GPU-friendly data structure and maze routing algorithm, we propose a GPU based algorithm, which can achieve a significant speedup on our CPU maze routing based OARSMT algorithm. Our work has also proven that we can solve the OARSMT problem with parallel computing by applying the GPU-based algorithm appropriately.

In this paper, the problem formulation is shown in Section 2. Our proposed methods including both sequential and parallel approaches are discussed in Section 3. The experimental results are shown and discussed in Section 4.

2. Problem formulation

The problem of the obstacle-avoiding rectilinear Steiner minimum tree (OARSMT) can be formulated with a number of non-overlapping rectilinear blockages and a set of pins. Let $B = \{b_1, b_2, \ldots, b_k\}$ be a set of non-overlapping rectilinear blockages in a 2-dimensional space R. Let $P = \{p_1, p_2, \ldots, p_m\}$ be a set of pins for a m-pin net in R such that no pins $p_i \in P$ lie inside an obstacle $b_j \in B$. In this OARSMT problem, we construct a rectilinear Steiner minimum tree connecting all the pins in P to achieve minimum total length (measured by Manhanttan distance), while avoiding intersection with any blockages in R.

In the OARSMT problem, a pin cannot lie inside an obstacle but it can be at the corner or on the boundary of an obstacle. An obstacle cannot overlap with another obstacle, but it can point-touch another obstacle at a corner or line-touch at a boundary. The edge of the OARSMT cannot intersect with a blockage, but it can run along the borders of the obstacles.

3. Methodology

In this work, we propose a method with using the shortest path region (SPR), which is first defined in Liu et al.'s paper [23]. For two pins p and q on a plane, the corresponding shortest path region (SPR) is defined as an union of all candidates of the shortest path between p and q. We further extend the definition of the shortest path region to the union of the candidates of the shortest path between two rectilinear regions. A vertex, an edge, and a path on a plane are also considered as regions with zero area. For two non-overlapping rectilinear regions P and Q on a plane, the corresponding shortest path region (SPR), denoted as $SPR(P, Q)$, is the union of all possible shortest paths connecting P and Q. Some examples of SPR are shown in Fig. 1. In Fig. 1(a), the shaded rectangular region is $SPR(p_1, p_2)$. In Fig. 1(b), the horizontal line is the SPR between the edge e_1 and pin p_5. In Fig. 1(c), the shaded region is the SPR between edges e_2 and e_3. In Fig. 1(d), the SPR between edges e_4 and e_5 is separated into two parts due to the obstacle.

With the information of the SPR, our sequential and parallel based approaches are both developed based on maze routing method over the escape graph proposed by Ganley and Cohoon [4].

3.1. Escape graph construction

Traditional escape graph is constructed as follow: first, all the pins and the corners of the obstacles project line segments at four orthogonal directions. The projecting lines are blocked by any obstacle. The nodes are formed by all the intersecting points of the lines in an escape graph. The lines among the nodes are edges. To reduce the number of nodes in the escape graph, we only project line segments from pins first. Hence, only obstacles which block any line segment should have line segments projecting from their corners. Those line segments can be blocked by another obstacles and this may cause more line segments. The resulting escape graph ignores the obstacles which has no effect on the optimality of solution.

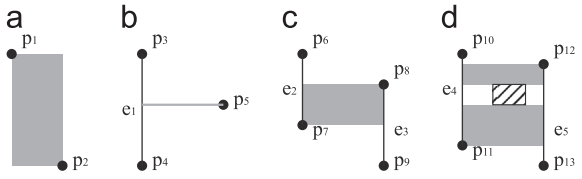


Fig. 1. Examples of shortest path region (SPR).

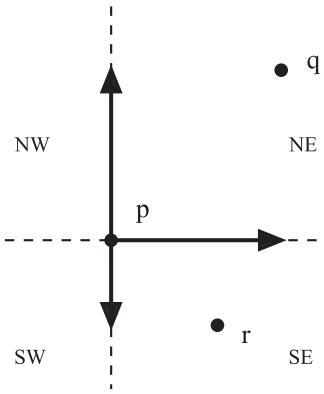


Fig. 2. An example of line projection.

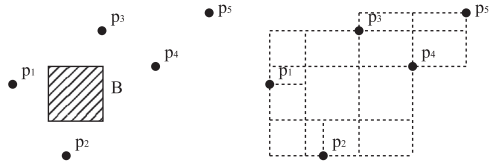


Fig. 3. A rectilinear Steiner tree problem and its corresponding escape graph.

We further reduce the number of nodes in the escape graph by limiting the length of projected lines. Nodes in an escape graph represent all possible locations of the Steiner points and the wire turning points. The nodes must also be related to the sources of the two intersecting line segments. In our OARSMT algorithms, we always find the closest pins or SPRs to connect with. Therefore, an intersection point which is distant from the source is useless. In our escape graph construction algorithm, we divide the routing region in four quadrants for each source of projected lines. The projected line segments are limited to minimum lengths. This guarantees that the intersections can be formed with the projected lines from the closest pins in the two neighboring quadrants. For example, in Fig. 2, p (x_p, y_p) is the source of line segments. q (x_q, y_q) and r (x_r, y_r) are the closest pins from p in north-east quadrant (NE) and south-east quadrant (SE), while there is no pin in the other two quadrants.The line segments projected from p should at least form intersections with line segments projected from q and r. Suppose these four line segments from p at direction north, south, east, west are L_n, L_s, L_e, L_w. L_n should form intersection points with line segments from the closest pins in NE and NW. Therefore, the length of L_n should be at least $|y_q - y_p|$ because p is the closest pin in NE and there is no pin in NW. In order to guarantee that the intersection between p and q can be formed, the length of the line segment from q at west direction should also be at least $|x_q - x_p|$. The lengths of other line segments are determined in similar way. Fig. 3 shows a set of pins, obstacles and their corresponding escape graph constructed with our algorithm. The pseudo-code of the escape graph construction is shown in Algorithm 1.

Algorithm 1. Our escape graph construction algorithm.

Let $P = \{p_1, p_2, \ldots, p_m\}$ be set of all pins
Let $B = \{b_1, b_2, \ldots, b_k\}$ be set of all blockages
Let $N = \{\}$ be set of all line segments' source point
Let $L = \{\}$ be set of all line segments
for each $p \in P$ **do**
  Let line segments l_{pd} is projected from p at direction $d \in \{north, south, east, west\}$
  Initialize $l_{pd} = 0$
  Add l_{pd} into L
  Add p into N
end for
while N is not empty **do**
  Remove n from N
  for each quadrant $R \in NE, NW, SE, SW$ **do**
    Find closest pins q in R from n
    Let (x_n, y_n) be position of point n
    Let (x_q, y_q) be position of pin q
    Let l_{nh} be the horizontal line segment touching R
    Let l_{nv} be the vertical line segment touching R
    Let l_{qh} be the horizontal line segment from q at opposite direction of l_{nh}
    Let l_{qv} be the vertical line segment from q at opposite direction of l_{nv}
    $l_{nv} = max(|y_n - y_q|, l_{nv})$
    $l_{nh} = max(|x_n - x_q|, l_{nh})$
    $l_{qv} = max(|y_n - y_q|, l_{qv})$
    $l_{qh} = max(|x_n - x_q|, l_{qh})$
  end for
  for each line segment l from n **do**
    if l is blocked by obstacle $b \in B$ **then**
      Add all corners of b into N
      Create line segments projected from the corners and initialize them as zero at direction $d \in \{north, south, east, west\}$
      Add line segments into L
    end if
  end for
end while
Let $G = (V, E)$ be the escape graph
Add intersecting points among L into V
Add edges connecting nodes $v_i, v_j \in V$ into E if there is any line segment connecting v_i, v_j

3.2. Sequential approach in maze routing

In sequential approach, we pick an arbitrary pin p' from our m-pin net $P = \{p_1, p_2, \ldots, p_m\}$ to start with. Propagation is done with our heap-based maze routing algorithm to find the second pin p'' which has the shortest obstacle-avoiding path to p'. Multiple shortest paths between p' and p'' are recorded, which enclose the $SPR(p', p'')$. This can be propagated from all recorded pins and paths to find the next pins. A set of paths connecting the pin and the previously recorded paths are recorded. Propagation is done iteratively until all pins are reached. The collection of all recorded paths forms a graph G which connects all pins P. A minimum spanning tree of G is constructed to form the topology of OARSMT. The pseudo-code of a sequential approach is shown in Algorithm 2.

Algorithm 2. Our sequential approach.

Let V be set of all grid points in escape graph
$P = \{p_1, p_2, \ldots, p_m\} \in V$
$S = \{p_1\}$
$Paths = \{\}$
while there exist pins $p \in P$ which is not connected **do**
  Propagate from S until reaching another pin p_i
  Trace set of paths which enclose $SPR(S, p_i)$ and add to $Paths$
  Add all grid points $v_j \in V$ enclosed in $SPR(S, p_i)$ to S
end while
Find a MST T of connected graph formed by $Paths$

11: Remove edges from T until no non-pin node has degree of 1
12: Post-process to reduce wire length

3.2.1. Sequential propagation and backtracking

Our sequential approach of propagation and backtracking is different from the traditional maze routing based methods in handling multi-pin nets. Instead of connecting the pins sequentially and deciding paths once some unconnected pins are reached, we keep a set of the possible shortest paths. We defer our path selection until all pins are reached. The final route is constructed globally by finding an MST on all recorded paths. In order to route fast, the propagation is performed on the escape graph. A heap data structure is used such that the current shortest path is always expanded first.

A min-heap H is maintained in the propagation process. H stores 2-tuples (s,d) where s is a reference to a node in the escape graph and d is the length of the shortest path that can reach node s from some connected nodes. The heap H is sorted according to the values of d. Every node s in the escape graph also stores a value v_s which is the shortest distance found from any connected point, is initialized as ∞ before propagation. The value v_s is zero if the node s is connected and it is ∞ if it is not reached before.

Before the propagation, the node of an arbitrary pin p' in the escape graph stores $v_{p'}$ which is zero. A 2-tuple value $(p',0)$ is created and it is added into the min-heap H. Fig. 4(a1) shows the initial values stored in each node, where only the node at p_1 has a value of zero. During the propagation, H pops up the node s_i with minimum path length value d_i. For each neighboring nodes s_j of s_i, we assume that the distance between s_i and s_j is $dist(s_i,s_j)$. If the stored v_{s_j} value in the escape graph is found to be ∞ or larger than $d_i+dist(s_i,s_j)$, the value of v_{s_j} is updated to $d_i+dist(s_i,s_j)$. The 2-tuple value $(s_j, d_i+dist(s_i,s_j))$ is pushed into H. In this way, the path length value is propagated in the escape graph until one of the pins is reached. Fig. 4(b1) shows the values stored in each node when p_2 is reached with value 7.

When a pin p_j is reached in propagation, backtracking is performed to find a set of rectangular paths connecting to p_j with the same shortest path length. The set of paths should contain the whole shortest path region (SPR) between p_j and some previously connected nodes. It means that any point on the border of a SPR should be covered by at least one path. The Fig. 4(b2) shows the SPR between p_1 and p_2 and the two bold paths are the required paths which must be obtained during backtracking. It is hard to identify where the SPR border is. The reason is that a SPR could be a zero-area region, or it is composed of several discontinuous regions. Our backtracking strategy tries to move with turning as little as possible, unless meeting an obstacle in the tracking direction, or seeing an obstacle corner on one side of the path. With this strategy, we find multiple paths of the same length, and the paths will at least contain the whole SPR. All the points on these new paths found are set to zero for the next round of propagation. Fig. 4(c1) shows the result of the second round of propagation as an example. Instead of starting the maze routing once again from scratch, we can continue with the propagation from the last round after inserting some new elements into the heap H. The heap data structure allows us to update the distances of the affected points rapidly without repeated works. The propagation and backtracking are repeated until the all the pins are reached. The Fig. 4(d1)–(e2) shows the remaining steps. We can obtain the set of path in Fig. 4(f) after the last backtracking step.

After finding a set of all paths, we can look at the ending points of each path. Since they are either a pin or a Steiner point, we can identify the positions of all the Steiner points immediately after this step. Then, we need to find the path lengths between the set of pins and Steiner points if they are connected. These distances can be computed very efficiently by tracing each path once. When we are tracing a path p, we can identify the positions of the Steiner points or the pin points, and we can tell the distances between them easily. A Steiner tree T will be constructed based on these distances and paths to connect all the points. It may happen that some paths are connected to some Steiner points only in T. We can remove these edges by removing recursively those leaf nodes

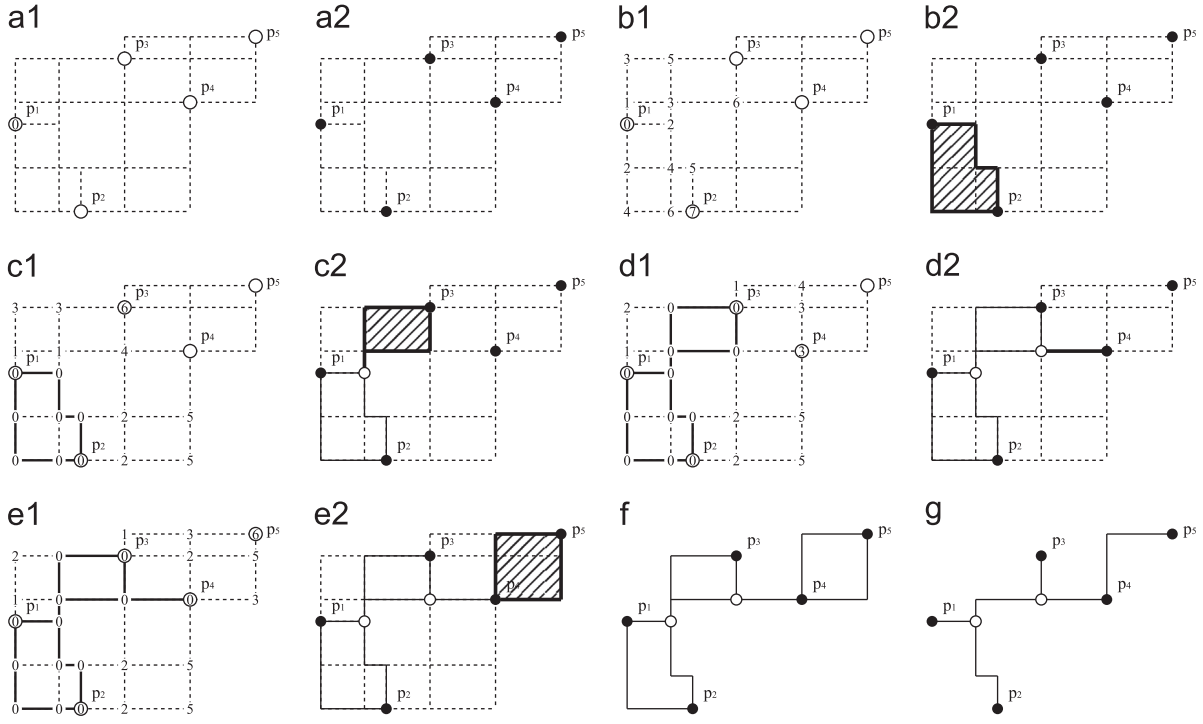


Fig. 4. Sequential approach of rectilinear Steiner tree construction. (a1), (b1), (c1), (d1), (e1) show the values stored in each node, and (a2), (b2), (c2), (d2), (e2) show their corresponding SPRs found by using the node values and the edges formed in each step.

which are Steiner points. After this, the tree gives a reasonable route connecting all the pins and avoiding the obstacles. An example of this MST construction step is shown in Fig. 4(g).

3.3. Parallel approach in maze routing

Algorithm 3. Our parallel approach.

1: Let $R=\{p_1, p_2, \ldots, p_m\}$ be all initial SPR
2: Let $T=R$ be set of unconnected group of SPRs
3: **while** T is not empty, i.e., there exists any pair of SPR $r_a, r_b \in R$ are not connected **do**
4: Propagate from all $t_i \in T$ until reaching another disconnected $t_j \in T$
5: **for** each $t_i \in T$ **do**
6: **if** the closest region of t_i is t_j and the closest region of t_j is t_i **then**
7: Trace $SPR(t_i, t_j)$ and add to R
8: Remove t_i and t_j from T
9: Add $SPR(t_i, t_j) \cup t_i \cup t_j$ to T
10: **end if**
11: **end for**
12: **end while**
13: **for** each $r_i \in R$ **do**
14: Trace path P in SPR r_i to connect local source and local target
15: **end for**
16: Remove edges from T until no non-pin node has degree of 1
17: Post-process to reduce wire length

The pseudo-code of our parallel approach is shown in Algorithm 3. We propagate from all pins in the same iteration. For each pin $p_i \in P$, its closest pin, $p_j \in P$, is found. If p_i is also the closest pin to p_j, $SPR(p_i, p_j)$ is recorded and the two pins are called "connected". When a SPR is found, the nodes in the SPR are recorded. The path source and the target of the union of paths, which is called local source and local target, are also recorded. In Fig. 5(a), SPR between p_1 and p_2 and SPR between p_3 and p_4 are found and recorded in the first iteration. After the first iteration, multiple SPRs and some unconnected pins are resulted. In the next iteration, we do the propagation from all SPRs and unconnected pins. All the closest pairs of the propagation sources, which can be SPRs or pins, are found and the corresponding SPRs between them are recorded such as in Fig. 5(b). This process is iteratively done until all pins are connected directly or indirectly via SPRs. Fig. 5(c) shows an example of resulting set of SPRs which connects all pins after two iterations of propagation. Then, for each SPR, we realize a path in SPR connecting the local source and the local target by backtracking process. The path realization step is done in reverse order of SPR creation, i.e., SPRs created in the last iteration realize their paths first. Fig. 5(d) shows the result of the first step of path realization, which realizes path e_1 connecting p_5 , $SPR(p_3, p_4)$ and path e_2 connecting $SPR(p_1, p_2)$, $SPR(p_3, p_4)$.

The white circles in Fig. 5(d) are the points where a realized path connecting the SPRs. These points are the possible locations of Steiner points. The paths in other SPRs must pass through them in order to keep the graph connected. We cannot find a single path that can pass through all the Steiner points sometimes. We find the minimum number of paths that can pass through all of them. In Fig. 5(e), SPR between p_1 and p_2 can be realized with one path, where SPR between p_3 and p_4 are realized with two paths in order to pass through both Steiner points. A connected graph G is formed when all SPRs are realized with paths. Cycles in G are removed and the resulting tree is a valid Steiner tree. Fig. 5(f) shows an example of resulting Steiner tree constructed with our algorithm. The details of our implementation of parallel approach are presented in next sub-section.

3.3.1. Parallel propagation

Our parallel approach is different from the traditional propagation of labels representing the distance from source point. Two 2-tuples $((s_1, d_1), (s_2, d_2))$ on every nodes are used in the escape graph, where d_1 and d_2 are the shortest path length from source s_1 and s_2 respectively. Before propagation, node at pins p_i are initialized as $((p_i, 0), (\varnothing, \infty))$ and all non-pin nodes are initialized as $((\varnothing, \infty), (\varnothing, \infty))$. Fig. 6(a) shows the initial node values before the propagation. During the propagation, each node n_i propagate to each neighboring node n_j with $((s_{1_i}, d_{1_i} + dist(n_i, n_j)), (s_{2_i}, d_{2_i} + dist(n_i, n_j)))$. At node n_j, two 2-tuples are chosen with the smallest d among $\{(s_{1_i}, d_{1_i} + dist(n_i, n_j)), (s_{2_i}, d_{2_i} + dist(n_i, n_j)), (d_{1_j}, s_{1_j}), (d_{2_j}, s_{2_j})\}$, provided that the chosen two 2-tuples should not share the same source s. As the result, every node $n$stores the two closest pins p_i and p_j as well as their shortest path length between n , p_i and between n , p_j. Fig. 6(b) shows the resulting node values after a complete propagation.

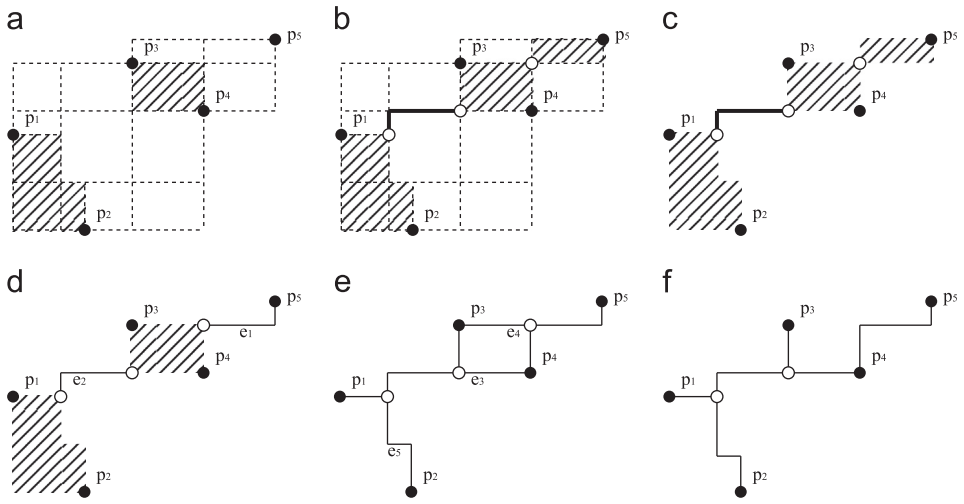


Fig. 5. Parallel approach of rectilinear Steiner tree construction.

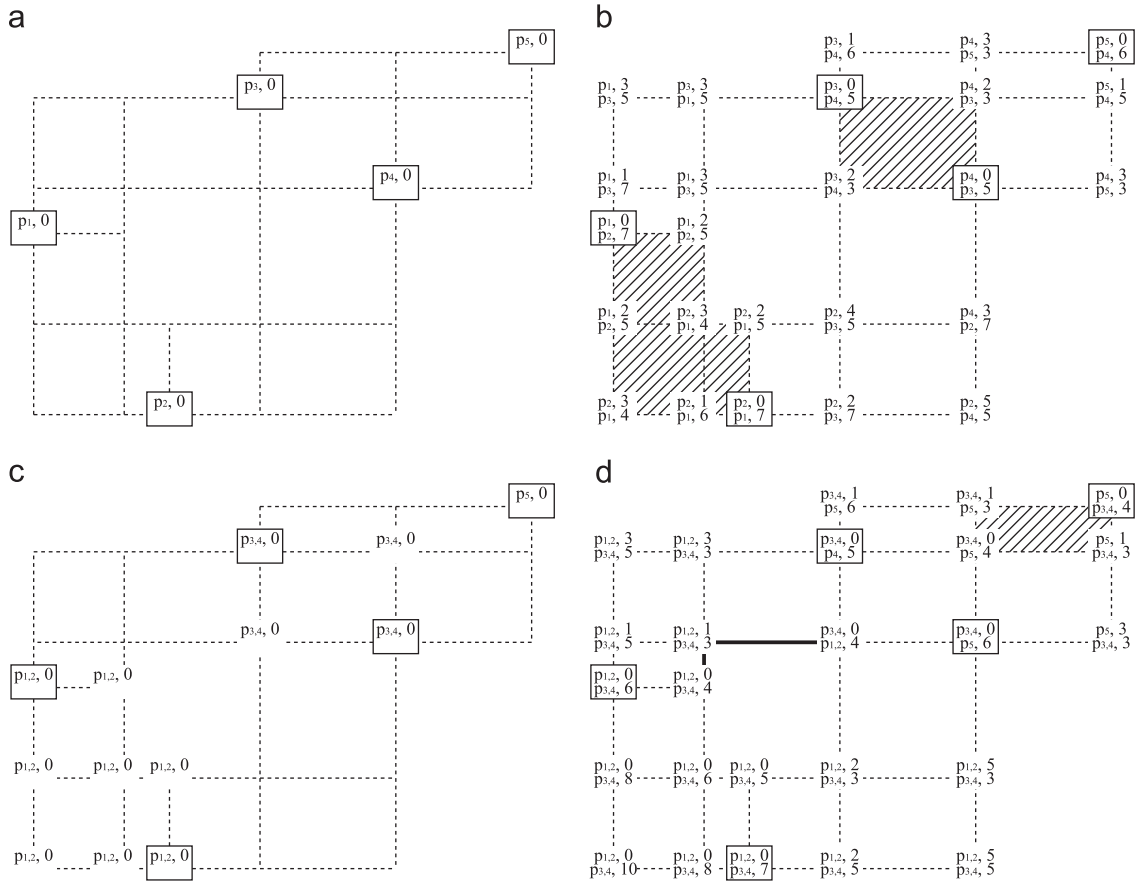


Fig. 6. Propagation of values in our parallel maze routing method. (a) The initial node values at the beginning of the first propagation. (b) The node values after completing the first propagation and the SPRs formed by the values. (c) The initial node values at the beginning of the second propagation. (d) The node values after completing the second propagation and the SPRs formed by the values.

After the propagation, the two values at each node represent the two closest sources and their distances from the node. At each source node s_i, d_1 must be 0 and d_2 is the shortest path length to the closest another source s_j, i.e., $((s_i,0),(s_j,d_j))$. If there exists another node with values $((s_j,0),(s_i,d_i))$ where $d_i{=}d_j$, the two sources s_i and s_j can be connected with at least one path. Multiple connectable source pairs can be found in each iteration of propagation-backtracking process. All possible shortest paths between each source pair s_i and s_j form a SPR, where every node in the SPR have value $((s_i,d_1),(s_j,d_2))$ such that $d_1+d_2=d_i=d_j$. It can be observed in Fig. 6(b) and (d) that the two shaded region satisfy the above conditions. Our SPR recording process does not need any path-tracing from one end to another. For creating SPR between sources s_i and s_j, we simply mark every node with the value $(s_i,d_1),(s_j,d_2))$ where $d_1+d_2=d_i$. The marking process for each node is data-independent and it is done in parallel with the number of parallel threads equal to the total number of nodes. Then, we set every node at pins and SPRs with $((s_{ij},0),(\varnothing,\infty))$ as shown in Fig. 6(c) and start another round of propagation. The propagation is done iteratively until all pins are connected.

Updating of each node depends on data at neighbor nodes at all four directions. If we update all nodes in parallel with GPU, communication among threads is needed after every update. Since inter-thread communication is expensive in GPU, it should be minimized. We partition the escape graph into two sets of paths: horizontal paths set and vertical paths set, as in Fig. 7. Every node in the horizontal set maps to its corresponding node in the vertical set. During the propagation, each parallel thread first handles propagation of one horizontal path. Since all the paths are not connected, the propagations are data-independent among threads

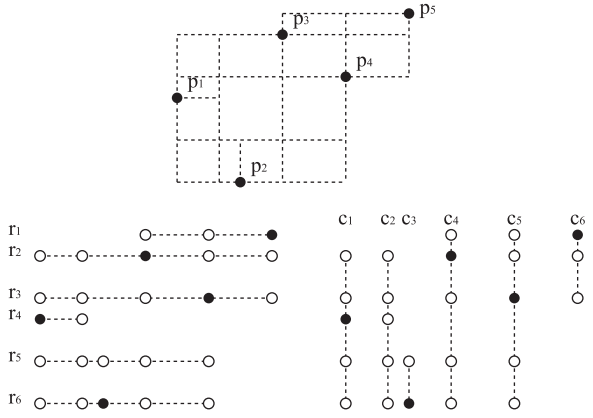


Fig. 7. Partitioning of escape graph for data-independent propagation.

and no inter-thread communication is required. Once all horizontal propagations are completed, the results in nodes are mapped to the nodes in vertical set. Propagation is done in the vertical paths in parallel and the results are mapped back to the horizontal set again. The propagation is finished when there is no more update in values in either horizontal or vertical propagation. The result is then mapped back to the original escape graph. The partitioning of 2D escape graph into set of 1D paths allows us to allocate GPU memory in the way that favor memory coalescing. All paths in GPU memory can be coalesced despite its orientation in escape graph. It is an important technique to ensure data are moved efficiently among global memory and registers or shared memory.

3.3.2. Parallel local path tracing

After the parallel propagation and backtracking process, all pins are connected to each other pins directly or indirectly with one or more SPRs. The relationship of SPR can be represented with a directed acyclic graph G. An example is given in Fig. 8 which corresponds to the case we used in previous sub-section. Nodes in G represent the SPRs or pins. Each SPR node has two outgoing edges and pin nodes have no outgoing edges. The directed edges from u to v_1 and u to v_2 represent the SPR u connects v_1 and v_2, where v_1 and v_2 can be another SPR or pin. When a path is realized in SPR u, the two end points of the path s_{v_1} and s_{v_2} may locate at the border of v_1 and v_2, which are the potential location of Steiner point. When we trace paths in v_1 and v_2, the path in v_1 should pass through s_{v_1}. In addition, the path in v_2 should pass through s_{v_2} in order to keep all the paths being connected and form a tree structure. In some cases, more than one path is required to connect all the potential Steiner points and a cycle is formed in such case. To remove the cycle, we remove the longest non-overlapping path segment of $n-1$ sub-paths when n paths are realized in the SPR.

For each SPR, we use dynamic programming approach to find the best shortest path such that it contains the most number of potential Steiner point. In the sub-graph covered by the SPR, we reuse the values which define SPR in previous step. The values provide information of the distance of the node to source nodes and target nodes. Thus, such information can provide all possible path directions at each node. We define another value sp at each node and it is initialize to 0. sp represents the maximum number of potential Steiner points which the sub-path from source to the node can pass through. At the source node, the sp value is propagated to all the neighboring nodes. If the neighboring node is a potential Steiner point, sp is increased by 1 before it is written into the neighboring node. Moreover, if the propagating value is larger than the value in a node, the larger value is also written. At the target node, we can obtain the best path by back-tracing from target to source following the neighboring node with the largest sp. The whole process of path tracing in a single SPR is shown in algorithm and illustrated in Fig. 5(d–f).

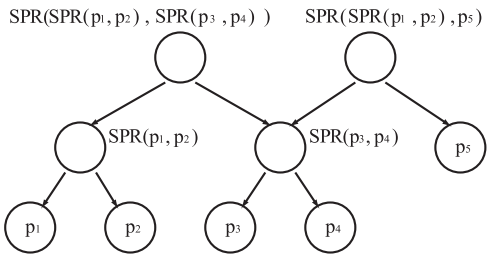


Fig. 8. Connection relationship of pins and SPRs.

3.4. Local refinement scheme

With the results generated by either our sequential or parallel approach, we perform a post-processing step to further reduce the total wire length. In this post-processing step, we first identify all the pin points, Steiner points, and turning points from the paths we computed. For each point, we will first check whether it forms one of the topologies shown in Fig. 9.

If such topologies exist, we will try to form a cycle by adding one straight edge (dotted line) between the two parallel paths. The longest path on this cycle will then be deleted to reduce the total wire length. We will consider all the possible cases of having pins or Steiner points lying on this cycle. This local refinement scheme is more general than the U-shape pattern refinement in [14].

3.5. Time complexity of parallel propagation

The algorithm exploits parallelism mainly in parallel propagation. Parallel propagation do multiple propagations in one direction at a time. A complete Hanan graph contains at most $(m+2k)^2$ nodes, where m is number of pins and k is number of blockages. In a multi-threaded single-direction propagation, there are at least $m+2k$ independent threads and each thread do $m+2k-1$ times of node-to-node propagation. When there is no obstacle, one iteration of propagation in all four directions reaches every node in the Hanan graph. When there is obstacles, propagation in single direction can be blocked. Sometimes, it requires a detour path to reach another SPR, and the formation of detour path requires multiple iterations of propagation. If the blockages are not overlapping, a blockage can block a propagation from the same source at most once. The worst case required is $k-1$ propagations to reach every grid point. Therefore, the complexity of a complete propagation is in $O((m+2k-1)(k-1))$, which is about $O(mk+k^2)$. However, since each source of propagation just have to reach its nearest SPR in each of the four quadrants, only the blockages locating between each pair of nearest SPRs can block the propagation. The experiments show that the worst benchmark just requires 8 iterations of propagation to reach all grid point in the routing graph. That benchmark is an extreme case of having a lot of slender blockages but very few pins. Column B in Table 1 shows maximum number of iterations required to do a complete propagation. We can see that the actual number of iterations is far less than $O(mk+k^2)$.

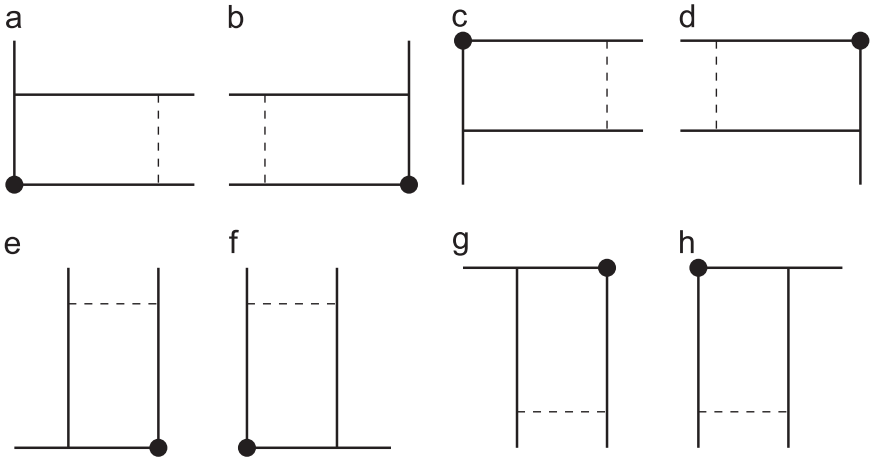


Fig. 9. Local refinement scheme. Dotted edges are the paths we consider adding to form a cycle.

Table 1
Number of iterations in parallel approach. m is number of pins, k is number of blockages, and $m+2k$ is maximum width and height of Hanan graph with m pins and k blockages. Column A is the number of complete propagation and traceback done to connect all pins. Column B is the maximum number of iterations in a single propagation to reach all grid points.

Benchmarks	m	k	$m+2k$	A	B
ind1	10	32	74	7	4
ind2	10	43	96	7	5
ind3	10	50	110	5	4
ind4	25	79	183	14	4
ind5	33	71	175	13	4
rc01	10	10	30	5	4
rc02	30	10	50	11	5
rc03	50	10	70	13	5
rc04	70	10	90	11	3
rc05	100	10	120	21	3
rc06	100	500	1100	21	6
rc07	200	500	2200	21	4
rc08	200	800	3400	22	5
rc09	200	1000	4200	22	5
rc10	500	100	900	22	4
rc11	1000	100	1400	17	4
rc12	1000	10,000	41,000	17	5
rt1	10	500	2010	4	8
rt2	50	500	2050	11	6
rt3	100	500	2100	21	5
rt4	100	1000	4100	21	7
rt5	200	2000	8200	21	7
RL01	5000	5000	25,000	18	4
RL02	10,000	500	12,000	17	3
RL03	10,000	100	10,400	16	4
RL04	10,000	10	10,040	16	3
RL05	10,000	0	10,000	17	4

Table 2
Comparison on wire length (in nm), where A columns are wire length, B columns are comparison of wire length to optimum solution from [19].

Bench-marks	[19]	[17]		[18]		[23]		Ours (sequential)		Ours (parallel)	
	A	A	B (%)	A	B (%)	A	B (%)	A	B (%)	A	B (%)
ind1	604	626	3.64	604	0.00	604	0.00	619	2.48	609	0.83
ind2	9500	9700	2.11	9500	0.00	9600	1.05	9500	0.00	9500	0.00
ind3	600	600	0.00	600	0.00	600	0.00	600	0.00	600	0.00
ind4	1086	1095	0.83	1129	3.96	1092	0.55	1096	0.92	1092	0.55
ind5	1341	1364	1.72	1364	1.72	1374	2.46	1360	1.42	1345	0.30
rc01	25,980	26,740	2.93	25,980	0.00	26,040	0.23	25,980	0.00	25,980	0.00
rc02	41,350	42,070	1.74	42,110	1.84	41,570	0.53	42,010	1.60	41,740	0.94
rc03	54,160	54,550	0.72	56,030	3.45	54,620	0.85	54,390	0.42	55,500	2.47
rc04	59,070	59,390	0.54	59,720	1.10	59,860	1.34	59,740	1.13	60,120	1.78
rc05	74,070	75,430	1.84	75,000	1.26	74,770	0.95	74,650	0.78	75,390	1.78
rc06	79,714	81,903	2.75	81,229	1.90	81,854	2.68	81,607	2.37	81,340	2.04
rc07	108,740	111,752	2.77	110,764	1.86	111,211	2.27	111,542	2.58	110,952	2.03
rc08	112,564	118,349	5.14	116,047	3.09	116,132	3.17	115,931	2.99	115,663	2.75
rc09	111,005	114,928	3.53	115,593	4.13	113,559	2.30	113,460	2.21	114,275	2.95
rc10	164,150	167,540	2.07	168,280	2.52	167,460	2.02	167,620	2.11	167,830	2.24
rc11	230,837	234,097	1.41	234,416	1.55	236,018	2.24	235,283	1.93	235,866	2.18
rc12	N.A.	780,528	N.A.	756,998	N.A.	762,435	N.A.	761,606	N.A.	762,089	N.A.
rt1	2146	2259	5.27	2191	2.10	2193	2.19	2231	3.96	2192	2.14
rt2	45,852	48,684	6.18	48,156	5.02	47,488	3.57	47,297	3.15	47,690	4.01
rt3	7964	8347	4.81	8282	3.99	8231	3.35	8187	2.80	8278	3.94
rt4	N.A.	10,221	N.A.	10,330	N.A.	9893	N.A.	9914	N.A.	10,073	N.A.
rt5	N.A.	53,745	N.A.	54,598	N.A.	52,509	N.A.	52,473	N.A.	52,616	N.A.
RL01	N.A.	N.A.	N.A.	483,027	N.A.	N.A.	N.A.	481,813	N.A.	482,455	N.A.
RL02	N.A.	N.A.	N.A.	637,753	N.A.	N.A.	N.A.	638,439	N.A.	640,866	N.A.
RL03	N.A.	N.A.	N.A.	640,902	N.A.	N.A.	N.A.	642,380	N.A.	644,646	N.A.
RL04	N.A.	N.A.	N.A.	697,125	N.A.	N.A.	N.A.	699,502	N.A.	701,098	N.A.
RL05	N.A.	N.A.	N.A.	728,438	N.A.	N.A.	N.A.	730,857	N.A.	731,301	N.A.

Some benchmarks are not run on some papers and they are marked with N.A.

The tree structure in Fig. 8 shows how the pins get connected in pairs. The number of level in the tree thus equals to the number of propagation and traceback iterations required to connect all pins. Therefore, m pins require at least $log_2(m)$ iterations and at most $m-1$ iterations. Therefore, the complexity of a complete parallel propagation is in $O((mk+k^2)(m-1))$, which is about $O(m^2k+mk^2)$. However, the experiments show that all benchmarks can be solved in less than 23 iterations of complete propagations. Column A in Table 1 shows total number of complete propagation and traceback.

Table 3
Comparison on runtime (in s) of different works, and speedup of our parallel approach over sequential approach. Some benchmarks are not run on some papers and they are marked with N.A.

Benchmarks	OASG approaches			Hanan graph approaches			Speedup by parallel
	[17]	[18]	[23]	[19]	Ours (sequential)	Ours (parallel)	
ind1	0.001	0.00	0.00	<1	0.01	0.05	0.20
ind2	0.001	0.00	0.00	<1	0.03	0.06	0.50
ind3	0.001	0.00	0.00	<1	0.01	0.05	0.20
ind4	0.002	0.00	0.00	1	0.01	0.09	0.11
ind5	0.002	0.00	0.00	1	0.01	0.08	0.13
rc01	0.001	0.00	0.00	<1	0.04	0.05	0.80
rc02	0.001	0.00	0.00	<1	0.09	0.06	1.50
rc03	0.001	0.00	0.00	<1	0.08	0.07	1.14
rc04	0.001	0.00	0.00	<1	0.09	0.06	1.50
rc05	0.001	0.00	0.00	1	0.07	0.09	0.78
rc06	0.017	0.03	0.02	335	0.38	0.38	1.00
rc07	0.026	0.03	0.03	541	0.75	0.31	2.42
rc08	0.043	0.05	0.04	24,170	1.35	0.46	2.93
rc09	0.049	0.06	0.05	14,174	1.75	0.61	2.87
rc10	0.016	0.02	0.01	176	0.30	0.20	1.50
rc11	0.021	0.03	0.02	706	0.94	0.34	2.76
rc12	0.681	1.19	1.20	N.A.	147.14	21.78	6.76
rt1	0.017	0.00	0.01	25	0.19	0.10	1.90
rt2	0.018	0.02	0.02	31	0.64	0.27	2.37
rt3	0.020	0.03	0.02	840	0.23	0.36	0.64
rt4	0.040	0.06	0.04	34,521	0.43	0.49	0.88
rt5	0.078	0.15	0.12	276,621	3.38	1.02	3.31
RL01	N.A.	1.15	0.63	N.A.	27.39	16.22	1.69
RL02	N.A.	1.18	0.37	N.A.	23.30	18.74	1.24
RL03	N.A.	1.13	0.32	N.A.	21.47	17.82	1.20
RL04	N.A.	1.57	0.29	N.A.	27.72	15.90	1.74
RL05	N.A.	0.12	N.A.	N.A.	31.08	16.74	1.86

4. Experimental results

We implemented our sequential algorithm in the C programming language. All the experiments were done in a computer with a 2.2 GHz Intel Pentium processor and 4GB memory running in the Linux environment. Our parallel algorithm is implemented in C programming language with CUDA library for GPU programming. The experiments were done in a computer with CPU of quad-core 1.6 GHz Intel Pentium processor and 4GB memory running in Linux environment. The GPU device is NVIDIA Tesla C1060 with 240 processing cores, clock rate of 1.3 GHz and 4GB memory. There are 27 benchmark circuits. Five industrial test cases (ind1–ind5) are provided from Synopsys. Twelve test cases are provided in [11] (rc1–rc12). Five randomly generated test cases are obtained from [14], and five large randomly generated test cases are obtained from [16]. We compared our sequential and parallel algorithms with the approaches in [17–19,23].

Table 2 lists the total wire length of the Steiner tree constructed by all six algorithms. The left columns (A) for each algorithm shows the resulting wire-length and the right columns (B) shows the amount of wire-length that is longer than the optimal solution proposed by [19] in percentage. We can see that the qualities of our settings are comparable to other algorithms.

Table 3 shows the runtimes of all six algorithms, as well as speedup of our parallel approach comparing to our sequential approach. [17,18,23] are based on OASG routing graph, which is irregular in structure and thus they are difficult to benefit from parallelism with GPU. Huang and Young [19] and both of our sequential and parallel approaches are based on Hanan graph. From this table, we can see that our parallel method can achieve significant speedup on large test-cases. For the test-cases which use more than 10 s in our sequential approach, we can observe speedups of up to 6.76. It shows that our parallel framework can improve the overall performance in large test cases. For small benchmarks which take less than 1 s to complete, the parallel approach is much less beneficial. The reason is that the performance overhead due to data transfer between the main memory to the GPU memory is inevitable. Notice that there is only 240 processing cores on our GPU. Therefore, when total number of parallel threads over the number of processing cores, they are processed in batch sequentially. In our algorithm, the maximum number of parallel threads equals the width or height of Hanan grid, which is about $m+2k$ where m is number of pins and k is number of blockages. From Table 1, we can see that the value of $m+2k$ is usually much larger than 240. This significantly constraint our experiment runtime. Significant speedup improvements are expected when number of processing cores is increased with future advance of GPU technology.

5. Conclusion

In this paper, we have improved the sequential approach of maze routing based method and proposed a new parallel approach to construct obstacle avoiding Rectilinear Steiner Minimal Trees. Experimental results on several test-cases show the speedups of our parallel approach over the sequential approach. This is the first work to propose a parallel approach to construct OARSMTs. With our framework of simultaneously building paths among multiple pins and SPRs pair, the number of computing iterations in RSMT construction is greatly reduced. In each iteration, our graph partitioning scheme provides a good adaptation of maze routing algorithm into GPU. However, the technology of applying GPU in general purpose computation is still new and there is large rooms for improvement. The performance of a single GPU core is much lower than that of a CPU core. The maximum number of GPU threads is also not enough for our algorithm to handle the large test-cases, which reduce parallelism and brings some runtime penalty. We can foresee that our algorithm can be benefited by future GPU technology advance.

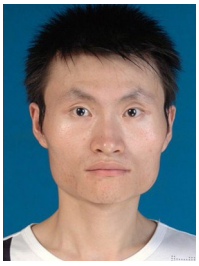
Acknowledgements

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK418611).

References

[1] M. Garey, D. Johnson, The rectilinear Steiner Tree problem is NP-complete, SIAM Journal on Applied Mathematics 32 (June (4)) (1977) 826–834.
[2] K. L. Clarkson, S. Kapoor, P. M. Vaidya, Rectilinear shortest paths through polygonal obstacles in $O(n \log^2 n)$, in: Proceedings ACM Symposium on Computational Geometry, 1987, pp. 251–257.
[3] Y.F. Wu, M.D.F. Schlag, P. Widmayer, C.K. Wong, Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles, IEEE Transaction on Computer-Aided Design 36 (3) (1987) 321–331.
[4] J. Ganley, J.P. Cohoon, Routing a multi-terminal critical net: Steiner Tree construction in the presence of obstacles, in: Proceedings of the International Symposium on Circuits and Systems, 1994, pp. 113–116.
[5] Y.F. Wu, M.D.F. Schlag, P. Widmayer, C.K. Wong, Finding obstacle-avoiding shortest paths using implicit connection graphs, IEEE Transactions on Computer-Aided Design 15 (1) (1996) 103–110.
[6] Y. Yang, Q. Zhu, T. Jing, X. Hong, Y. Wang, Rectilinear Steiner minimal tree among obstacles, in: Proceedings of IEEE ASICCON, 2003, pp. 348–351.
[7] H. Wu, Z. Feng, T. Jing, X. Hong, Y. Yang, G. Yu, X. Hu, G. Yan, FORst: a 3-step heuristic for obstacle-avoiding rectilinear Steiner minimal tree construction, Journal of Information and Computational Science (2004) 107–116.
[8] H. Wu, T. Jing, X. Hong, Z. Feng, X. Hu, G. Yan, An-OARSMan: obstacle-avoiding routing tree construction with good length performance, in: Proceedings of ASP-DAC, 2005, pp. 7–12.
[9] Y. Shi, T. Jing, L. He, Z. Feng, X. Hong, CDCTree: novel obstacle-avoiding routing tree construction based on current driven circuit model, in: Proceedings of ASP-DAC, 2006.
[10] Z. Shen, C. Chu, Y. Li, Efficient rectilinear Steiner tree construction with rectilinear blockages, in: Proceedings of ICCD, 2005, pp. 38–44.
[11] Z. Feng, Y. Hu, T. Jing, X. Hong, X. Hu, G. Yan, An $O(n \log n)$ algorithm for obstacle-avoiding routing tree construction in the λ−geometry plane, in: Proceedings of ISPD, 2006, pp. 48–55.
[12] P.C. Wu, J.R. Gao, T. C. Wang, A fast and stable algorithm for obstacle-avoiding rectilinear Steiner minimal tree construction, in: Proceedings of ASP-DAC, 2007, pp. 262–267.
[13] R. Hentschke, J. Narasimham, M. Johann, R. Reis, Maze routing Steiner trees with effective critical sink optimization, in: Proceedings of ISPD, 2007.
[14] C.W. Lin, S.Y. Chen, C.F. Li, Y.W. Chang, C.L. Yang, Efficient obstacle-avoiding rectilinear Steiner tree construction, in: Proceedings of ISPD, 2007.
[15] J. Long, H. Zhou, S.O. Memik, EBOARST: an efficient edge-based obstacle-avoiding rectilinear Steiner tree construction algorithm, IEEE Transactions on Computer-Aided Design 27 (12) (2008) 2169–2182.
[16] L. Li, F.Y. Young, Obstacle-avoiding rectilinear Steiner tree construction, in: Proceedings of ICCAD, 2008. pp. 523–528.
[17] C.H. Liu, S.Y. Kuo, S.Y. Yuan, Y. H. Chou, An $o(n \log n)$ path-based obstacle-avoiding algorithm for rectilinear Steiner tree construction, in: Proceedings of DAC, 2009, pp. 314–319.
[18] G. Ajwani, C. Chu, W.K. Mak, Foars: flute based obstacle-avoiding rectilinear Steiner tree construction, in: Proceedings of ISPD, 2010, pp. 27–34.
[19] T. Huang, F.Y. Young, An exact algorithm for the construction of rectilinear Steiner minimum trees among complex obstacles, in: Proceedings of DAC, 2011, pp. 164–169.
[20] Y. Deng, Gpu accelerated VLSI design verification, in: Proceedings of International Conference on Computer and Information Technology (CIT), 2010, pp. 1213–1218.
[21] Y. Han, S. Roy, K. Chakraborty, Optimizing simulated annealing on GPU: a case study with ic floorplanning, in: Proceedings of International Symposium on Quality Electronic Design (ISQED), 2011, pp. 1–7.
[22] Z. Feng, P. Li, Multigrid on GPU: tackling power grid analysis on parallel SIMT platforms, in: Proceedings of ICCAD, 2008, pp. 647–654.
[23] C.H. Liu, S.Y. Yuan, S.Y. Kuo, J.H. Weng, Obstacle-avoiding rectilinear Steiner tree construction based on Steiner point selection, in: Proceedings of ICCAD, 2009, pp. 26–32.

Wing-Kai Chow received his B.Sc. degree from The Hong Kong Polytechnic University in 2009. He has worked as a research assistant in the same university in 2010 and as a research assistant in The Chinese University of Hong Kong in 2010–2012. In 2012, he received the M.Sc. degree in Computer Science from The Chinese University of Hong Kong. He is now a Ph.D. student from The Chinese University of Hong Kong. His research interests are design automation of VLSI including placement and routing.



Liang Li received the Bachelor degree from the Nanjing University of Aeronautics and Astronautics, and the M.Phil degree from the Chinese University of Hong Kong. Liang Li is now working in the Oracle Research & Development Center (Shzenzhen) Company. His research interests include VLSI CAD, algorithms and combinatorial optimization. He focusses on routing problems, especially the problem in the presence of obstacles.

Evangeline Young received her B.Sc. degree and M. Phil. degree in Computer Science from The Chinese University of Hong Kong (CUHK). She received her Ph.D. degree from The University of Texas at Austin in 1999. She is currently a professor in the Department of Computer Science and Engineering in CUHK. Her research interests include algorithms and CAD of VLSI circuits. She is now working actively on floorplanning, placement, routing, DFM and algorithmic designs. Dr. Young served on the organization committees of ISPD, ARC and FPT and the program committees of several major conferences including DAC, ICCAD, ISPD, ASP-DAC, DATE and GLSVLSI. She also served on the editorial boards of IEEE TCAD, ACM TODAES and Integration, and the VLSI Journal.

Chiu-Wing Sham received the Bachelor degree in computer engineering, and the M.Phil. degree and the Ph.D. degree from The Chinese University of Hong Kong, Hong Kong, in 2000, 2002, and 2006, respectively. He was a Research Engineer with Synopsys, Shanghai, China, and an Electronic Engineer working on the FPGA applications of motion control system with ASM (HK). He joined the Electronic and Information Engineering Department of The Hong Kong Polytechnic University, as a Lecturer in August 2006. His research interests include design automation of VLSI, design optimization of digital VLSI systems and embedded systems.