Detailed Routing by Sparse Grid Graph and Minimum-Area-Captured Path Search

Gengjie Chen
The Chinese University of Hong Kong
gjchen@cse.cuhk.edu.hk

Chak-Wa Pui
The Chinese University of Hong Kong
cwpui@cse.cuhk.edu.hk

Haocheng Li
The Chinese University of Hong Kong
hcli@cse.cuhk.edu.hk

Jingsong Chen
The Chinese University of Hong Kong
jschen@cse.cuhk.edu.hk

Bentian Jiang
The Chinese University of Hong Kong
btjiang@cse.cuhk.edu.hk

Evangeline F. Y. Young
The Chinese University of Hong Kong
fyyoung@cse.cuhk.edu.hk

ABSTRACT

Different from global routing, detailed routing takes care of many detailed design rules and is performed on a significantly larger routing grid graph. In advanced technology nodes, it becomes the most complicated and time-consuming stage. We propose Dr. CU, an efficient and effective detailed router, to tackle the challenges. To handle a 3D detailed routing grid graph of enormous size, a set of two-level sparse data structures is designed for runtime and memory efficiency. For handling the minimum-area constraint, an optimal correct-by-construction path search algorithm is proposed. Besides, an efficient bulk synchronous parallel scheme is adopted to further reduce the runtime usage. Compared with the first place of ISPD 2018 Contest, our router improves the routing quality by up to 65% and on average 39%, according to the contest metric. At the same time, it achieves 80–93% memory reduction, and 2.5–15× speed-up.

1 INTRODUCTION

Because of its enormous computational complexity, VLSI routing is usually performed in two stages, global and detailed. In the *global routing* stage, the routing space is split into an array of regular cells, where a coarse-grained routing solution is generated. It optimizes wire length, via count, routability, timing and other metrics with a global view. *Detailed routing*, on the other hand, realizes the global routing solution by considering exact metal shapes and positions. It takes care of many complicated detailed design rules (e.g., parallel-run spacing, end-of-line spacing, cut spacing, minimum area, etc). Its solution quality directly influences various eventual design metrics such as timing, signal integrity, chip yield [1]. Meanwhile, its solution space, a 3D grid graph, is significantly larger than that of global routing. In advanced technology nodes, detailed routing becomes the most complicated and time-consuming stage [2].

During the past decade, many approaches were proposed to complete fast and high-quality global routing with sustaining progress (e.g., FGR [3], FastRoute [4], BoxRouter [5], Ancher [6], GRIP [7], BonnRoute [8], NCTU-GR [9]). However, there is insufficient effort for exploring efficient and effective detailed routers in academia. RegularRoute [1] encourages regular routing patterns and exploits a maximum independent set formulation for better design rule satisfaction.

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287678

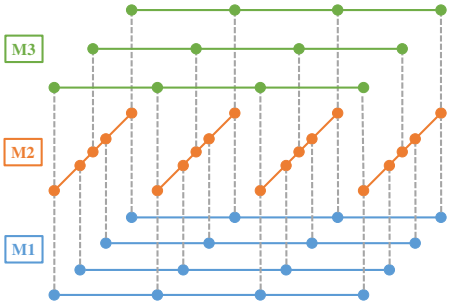


Figure 1: An example 3D detailed routing grid graph. In this example, preferred directions of metal-1 (M1) and M3 layers are both horizontal, while that of M2 is vertical.

MANA [10] considers end-of-line spacing and minimum length of a wire segment in maze routing. The work in [11] presents the data structures and algorithms for detailed routing used in BonnRoute. Besides, several specific issues in detailed routing have been discussed. For example, methods for the pin access optimization are proposed in [12–14]. For others, the impact of various manufacturing technologies have been dealt with, including triple patterning [15–17], self-aligned doubling patterning [18] and directed self-assembly [19].

As the feature size scales down, not only the problem size but also the complexity of design rules of detailed routing becomes increasingly challenging. Moreover, many detailed routers heavily relies on post processing for fixing design rule violations. Design rule dimensions, however, do not scale well with feature miniaturization (e.g., feature size decreases much faster than minimum area values) and require relatively more spaces for fixing. In this way, a post processing step fails more frequently [11]. Therefore, we proposes Dr. CU, a detailed routing framework that is superiorly scalable in runtime as well as memory usage and provides more correct-by-construction design rule satisfaction. Our contributions can be summarized as follows.

- We designed a set of two-level sparse data structures for a 3D detailed routing grid graph of enormous size.
- We proposed an optimal correct-by-construction path search that captures the minimum-area constraint.
- We also proposed an efficient bulk synchronous parallel scheme to further reduce the turn-around time of the detailed routing process.

2 PRELIMINARIES

Before illustrating the details of our data structures and algorithms, the problem formulation of detailed routing is introduced in this section.

2.1 Routing Space

VLSI Routing is on a stack of *metal layers*. A *wire* segment on a layer runs either horizontally or vertically. Each layer has a *preferred direction* for routing, which benefits manufacturability [14], routability and design rule checking [1]. The preferred directions of adjacent layers are perpendicular to each other in common design practice. Besides, regularly-spaced *tracks*, where the majority of wires are routed on, can be predefined according to the wire width and parallel-run spacing constraint. In this work, wrong-way and off-track wires are considered only for short connections (especially to pins). Wires on neighboring layers can be electrically connected by *vias*. The tracks on all metal layers define a 3D grid graph for detailed routing, as Fig. 1 shows. In this grid graph, an *edge* represents either a via or a wire segment.

Over the chip, there are some routing *obstacles* that vias and wire segments should avoid to prevent short and spacing violations. In detailed routing, the relatively small obstacles within standard cells (e.g., pins and intra-cell wires) should also be handled.

Assuming that a global routing result is already well optimized for certain metrics (e.g., timing, routability, power), a detailed router needs to honor the global routing result as much as possible. The optimized metrics are thus kept with detailed design rules handled. In this paper, the 3D global routing result is referred as *routing guide*, and out-of-guide routing (either wire or via) is penalized.

2.2 Design Rules

The most fundamental and representative design rules handled by detailed routing are as follows [2]. (1) Short: a via metal or wire metal cannot overlap with another metal object like via metal, wire metal, blockage, or pin, except when the two metal objects belong to the same net. (2) Spacing: the spacing between two metal objects should be large enough. (There are several types of spacing requirement, including end-of-line spacing, parallel-run spacing and cut spacing. Refer to [2] for detailed definitions.) (3) Minimum area: the area of a metal polygon is required to be above a threshold.

2.3 Problem Formulation

The detailed routing problem can be formally defined as follows. Given a placed netlist, routing guides, routing tracks and design rules, route all the nets to minimize a weighted sum of (i) total wire length, (ii) total via count, (iii) non-preferred usage (including out-of-guide and off-track wires/vias, and wrong-way wires), and (iv) design rule violations (including short, spacing and minimum-area violations). Note that design rule violations are highly discouraged and suffer much more significant penalty than others.

3 TWO-LEVEL SPARSE DATA STRUCTURES

The grid graph for detailed routing is similar to that for global routing in structure, but is significantly more fine-grained and thus has much larger scale. To support the detailed routing algorithms with both economic memory usage and efficient query, we design a set of two-level data structures for the routing grid graph.

There are a global grid graph and local ones, as Fig. 2 shows. The *global grid graph* data structure stores the graph implicitly without instantiating all vertices. Here, the information of routed edges are stored sparsely by balanced binary search trees (BSTs) and intervals. The *local grid graph*, a local cache of the global one, is created for routing a net. It is a sparse subgraph of the full-chip 3D grid graph on the routing region of a net, where edge costs are readily available for conducting maze routing.

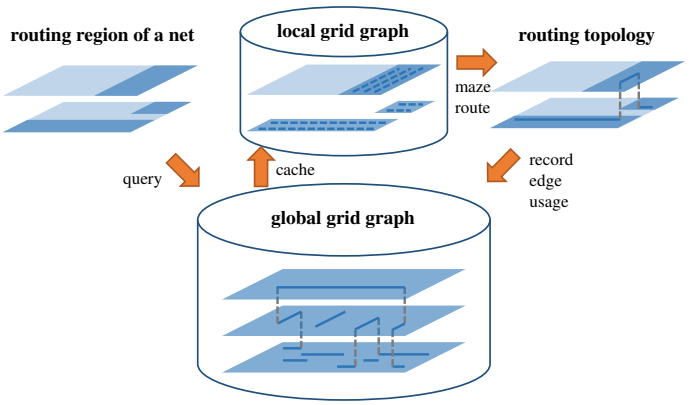


Figure 2: An overview of the two-level grid graph data structures.

3.1 Sparse Global Grid Graph

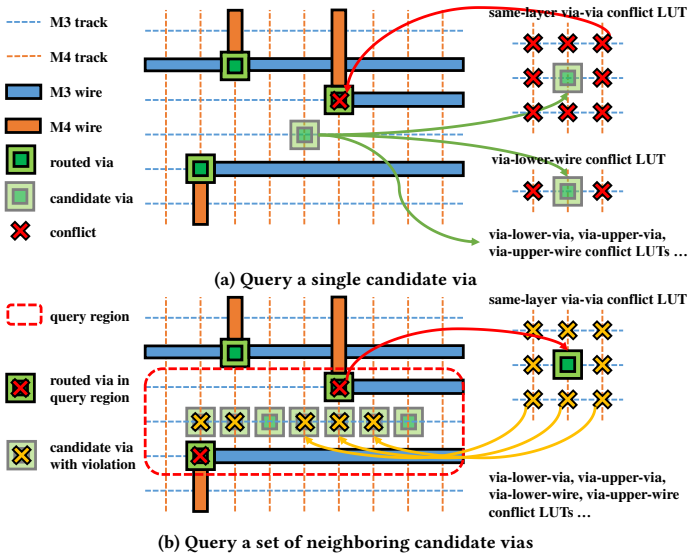
Edges of routed nets are called *routed edges*. Note that the an edge can be either a via or a wire segment. The global grid graph stores routed edges in the sparse data structure based on BSTs and intervals.

3.1.1 Vertex Index vs. Location. In the full-chip grid graph, a vertex is uniquely defined by a 3D index, which is a tuple of layer index, track index (in the non-preferred direction), and relative index along the track (in the preferred direction). The store and query of vias and wires in the global grid graph are done by index instead of by location, because the structural information is clearer with indexes. For example, given a vertex with its 3D index, its neighbors can be obtained immediately.

3.1.2 BST and Interval Based Storage. It is very expensive to use a full-chip 3D direct-address table for storing routed edges. First, its size will be unaffordable (1G vertices for just 10 metal layers and 10k tracks on each layer) [8]. Besides the time-consuming memory allocation and initialization, some queries are also inefficient if using this data structure. For example, to record, query or remove the usage of a wire segment (e.g., spanning 1000 vertices), we need to change or check all the 1000 vertices on it.

Instead of a 3D direct-address table, we use a 2D table for the layer dimension and track dimension (i.e., non-preferred direction), and use BST and intervals in the third dimension (i.e., preferred direction) for sparseness. For a track, there are three balanced BSTs, two for storing routed vias and one for storing routed wires. For vias, normal BSTs with indexes in the preferred direction being keys are used. Each via is stored twice, one on the lower track and the other one on the upper track. The duplication benefits the range searches that are needed on both the lower and upper tracks. This will be illustrated in detail later. For wire segments, a BST with nodes representing non-overlapping intervals is employed. In this way, the memory used is linear to the number of wire segments instead of vertices involved.

3.1.3 Conflict with Obstacles. For obstacles (including pins) with irregular shapes, the edges that may cause short/spacing violations with them are marked in advance. Since obstacles cannot be ripped up, the marking is a one-time effort. Via-obstacle violations are more difficult to capture than wire-obstacle violations, because there are several types of vias that can be chosen from. Essentially, all via types need to be attempted. A via location should be penalized if and only if all via types fail to satisfy the spacing requirement with its neighboring obstacles. Note that conflict with pins is net-dependent, because it is fine for a



(a) Query a single candidate via

(b) Query a set of neighboring candidate vias

Figure 3: Query the violations on candidate via edges due to the previously routed edges in the global grid graph.

via or wire to be close to a pin of the same net. Therefore, the marking should be net-dependent.

3.2 Global Grid Graph Query by Look-up Table

When routing a net, the edges that will be considered for using are referred as *candidate edges*. Their costs (possibly penalized by the short/spacing violations) will be queried from the global grid graph before running maze routing.

Different from the conflict with obstacles, the conflict with routed edges will change during the routing process and cannot be marked in advance. Considering various design rules and a significant number of candidate edges, a proper scheme that can efficiently query their costs is in need. We build several via/wire conflict LUTs to achieve that.

3.2.1 Via/Wire Conflict Look-up Table. For routing a net, the metal short with routed edges can be trivially detected as interval overlapping. For the following spacing violation conflicts, their identification is less straight-forward:

- Via-via conflicts: for a specific via, it may conflict not only with vias on the same cut layer (same-layer vias) but also with vias on the adjacent cut layers. The conflict between same-layer vias may be due to spacing rules on either cut layer, metal layers, or both. The conflict between different-layer vias is caused by metal spacing requirement.
- Via-wire conflicts: a via may have spacing violations with wire on the lower and the upper metal layers that it connects.
- Wire-wire conflicts: two wires may be too close to each other at their ends and violate the spacing constraint.

The above violations can be detected during routing. However, these detection operations are extremely frequent and on-the-fly detections are too time-consuming. Since we are working on a relatively regular grid graph, some light-weight LUTs can accelerate the process. Conceptually, *via/wire conflict LUTs* immediately tells what neighboring edges will conflict with a given edge. There are several types of them: when the given edge is a via e_i, a *via-lower-wire conflict LUT* tells what neighboring wire segments in the lower metal layer of e_i cause conflicts with e_i; similarly, given a wire segment e_j, a *wire-upper-via conflict LUT* tells what vias connecting to the layer above e_j may be conflicted with e_j; so on and so forth. Two conflict LUTs are called the *inverse LUT* to each other if the types of the given edge and the neighboring edges are swapped. For example, the inverse of a via-lower-wire LUT is a wire-upper-via LUT.

Regarding the indexing and sizes of conflict LUTs, we explain the via-via one as an example. For two same-layer vias, their distance is unique for specific track index differences in the lower metal layer and the upper metal layer, because of the equal spacing of the tracks. Therefore, only one LUT is needed for each layer. Such an LUT itself is 2D and is indexed by the track index differences. For two different-layer vias, three consecutive metal layers are involved. Using their corresponding vertices on the middle metal layers for indexing, their distance in the non-preferred direction is solely determined by the difference in track indexes. However, in the preferred direction, vertices along a track may have irregular spacing (e.g., M2 in Fig. 1). As a result, a layer needs a series of 2D LUTs, where each LUT serves for vertices with a specific index in the preferred direction.

3.2.2 Single Edge Query. The cost of a candidate edge consists of a unit edge cost and some possible penalty caused by two types of violations. The first type is violations with obstacles, which has been directly marked in the BSTs. The second type is violations with routed edges. The via/wire conflict LUTs tell the neighboring edge positions that will have conflict with the candidate edge. The only thing to do is to check whether the positions are occupied. An example is shown by Fig. 3(a). For the candidate via, a same-layer via-via conflict is detected with the help of the corresponding LUT. Meanwhile, there is no via-lower-wire conflict because no routed wire exists at the two potentially conflicting positions specified by the LUT.

3.2.3 Batch/Long Edge Query. Usually, a set of neighboring edges (either vias or wire segments) along a track are all candidate edges for routing a net. If querying them individually, $O(k \log n)$ time is needed with k being the number of candidate edges and n being the BST size[1]. A range search on BST can improve the efficiency. Given a set of candidate edges along a track and the corresponding LUTs, a *query region* where routed edges may have conflicts with can be identified. By range search on BST according the this query region and referring to the inverse LUTs, the conflicted candidate edges can be found. An example on detecting same-layer via-via conflict is illustrated by Fig. 3(b). First, the query region and two routed vias within it are identified. Starting from the two routed vias, the inverse LUT (the same-layer via-via conflict LUT) finds five conflicted candidate vias.

Suppose the number of routed edges within the query region is m. The range search on BST takes $O(m+\log n)$ time, which can be conducted by finding the first and last tree nodes within the range. Besides, $m = O(k)$. Note that m can be significantly smaller than k because a long routed wire segment is stored as a long interval instead of a bunch of short edges in a BST. Therefore, the time for retrieving the routing cost of the k candidate edges is $O(k) + O(m + \log n) = O(k + \log n)$ instead of $O(k \log n)$. Moreover, the cost of a long wire segment may be queried as a whole, then the time is further improved to $O(m + \log n)$.

In the batch query along a track, routed vias to both lower and upper layers should be considered. As mentioned in Section 3.1.2, a via is stored twice on both its lower and upper tracks. In this way, efficient BST range search along either track is enabled.

[1] To be more rigorous, since multiple BSTs (for vias or wires, for different layers) may all need to be queried, n represents the largest size of all BSTs.

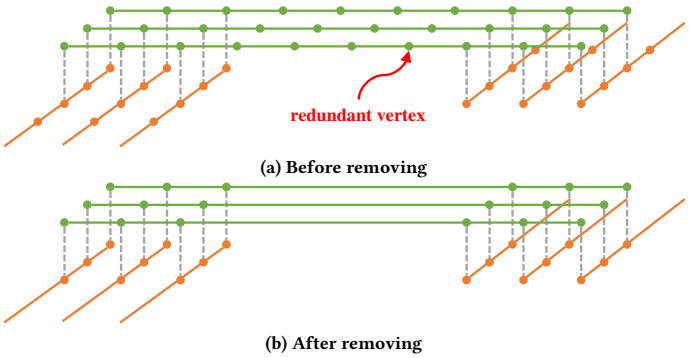


Figure 4: Long edges by removing redundant vertices.

3.3 Sparse Local Grid Graph

The local grid graph of a net is the subgraph of the full-chip 3D grid graph within its *routing region* (the routing guide with possibly minor expansion). In terms of data structures, it caches the graph structure and all edge costs of the subgraph by direct-address tables, supporting the maze routing.

Its sparsity is in two aspects. First, only the routing region is considered, which is substantially smaller than the full-chip region. Second, many vertices become redundant in this subgraph and are removed.

3.3.1 Routing Region. When routing a net, only the region around its routing guide is considered due to two reasons. First, detailed routing should honor global routing solution, i.e., routing guides, because many objectives (e.g., timing, routability) are optimized in global routing. For some local congestions, global routing may not be able to model and resolve, so minor out-of-guide routes may be necessary. However, such disturbance should be minimized. Second, maze routing on the full-chip 3D grid graph will suffer from prohibitive runtime due to its enormous scale. In our implementation, the routing region of a net is expanded by a small margin from its routing guide. All out-of-guide edges are penalized. For difficult-to-route nets, the expansion margin may be increased.

3.3.2 Long Edge. Conceptually, the local grid graph is simply a subgraph induced by vertices within the routing region. However, many vertices in the subgraph have only two neighbors remained and become redundant, as Fig. 4(a) shows. In this snippet of the subgraph, many vertices originally have neighbors on adjacent layers that are out of the routing region now. They have thus only two neighbors left on the track. In this way, as long as such a vertex does not belong to a pin, it can be safely removed with the two connected edges merging into one. This compressing step cuts down the problem size without affecting the final results. Both memory usage and runtime can be reduced.

3.3.3 Explicit Storage. In the global grid graph, vertices are implied by 3D indexes but are not instantiated. To support efficient vertex-wire operation in maze routing (e.g., recording the prefix and cost, propagating to neighbors), the local graph instantiates all its vertices and edges. To be more specific, vertices are assigned with continuous indexes staring from zero, and adjacency lists are also created. In this way, any vertex/edge information can be efficiently stored and retrieved by direct-address tables (instead of hash tables or BSTs).

4 ROUTING ALGORITHM

In routing (especially detailed routing), sequential maze routing is widely adopted due to its scalability (compared with concurrent methods like [7, 20]) and flexibility (for capturing various objectives and

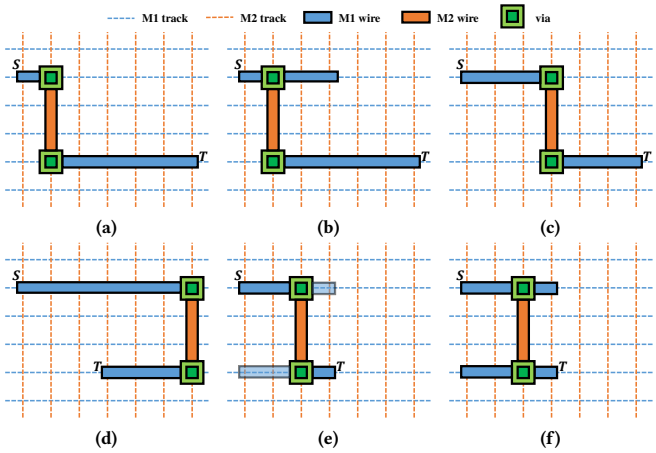


Figure 5: Capture minimum area cost in path search. Suppose the minimum area implies a length of three pitches. A path from source S to sink T is needed. (a) A normal path search without considering minimum-area violation. (b) Post fixing by extending wire. (c) Forcing the minimum length of wire segment in path search. (d) Detour due to the forcing. (e) & (f) Path search with wire extension considered.

violations). Recall from Fig. 2 that our local grid graph is sparse because of the routing guide and long edges, which enhances the efficiency of our maze routing. We follow the convention of sequential maze routing. Essentially, nets are routed one after another, where previously routed nets are treated as blockages. After routing all nets with possible violations, several rounds of rip-up and reroute (RRR) help to clean them up.

4.1 Edge Cost in Local Grid Graph

The cost $w(e)$ of each edge e in the local grid graph $G(V, E, w)$ is a weighted sum of the (i) basic wire cost (by length), (ii) basic via cost (by count), (iii) out-of-guide penalty, and (iv) short/spacing violation penalty. In this way, a path search (like Dijkstra's algorithm [21]) running on the grid graph will optimize these objectives automatically. The basic via/wire cost together with the short/spacing violation penalties are queried from the sparse global grid graph in batch.

Note that it is not determined by a single edge whether the minimum-area rule is violated or not. The minimum-area violation thus cannot be reflected as expensive edges like short/spacing violations and can only be captured by the path search algorithm.

4.2 Minimum-Area-Captured Path Search

For wires with a specific width, a minimum area implies a minimum-length constraint l_{min}. A straight-forward idea for fixing the violation after maze routing is to extend the wire segments that are not long enough. Such a greedy method may suffer from excessive wire length (e.g., Fig. (b) compared with Fig. (c)) and even insufficient spare space for extension. Another method, multi-label path search [11], forces the minimum length for every wire segment without considering the possibility of extension. In this way, significant but unnecessary detour may be paid (Fig. (d)). By capturing the minimum-area violation and its possible fixing during the path search, a better solution can be obtained (Fig. (e)).

We extend the conventional Dijkstra's algorithm [21] to comprehensively handle the minimum-area rule. In Dijkstra's algorithm, the cost/distance of a path can be directly incremented. That is, the cost of

Algorithm 1 Optimal Minimum-Area-Captured Path Search

Require: A local grid graph $G(V, E, w)$, source and sink vertices s and t, minimum length l_{min} of wire segment (implied by the minimum-area constraint).
Ensure: $s - t$ path P.

```
1: Q ← an empty priority queue for storing paths
2: v.costUB ← ∞, ∀v ∈ V
3: Initialize path P' at s (P'.prefix ← null, P'.vertex ← s, P'.costLB ←
   0, P'.costUB ← 0, P'.length ← l_min)
4: Push P' into Q
5: while Q is not empty do
6:     Pop the path P' with smallest P'.costLB from Q
7:     if P'.vertex = t then
8:         return P'
9:     for v ∈ P'.vertex.neighbors do
10:        Relax(P', v)

11: function Relax(P', v)                          ▷ Extend path P' to v
12:    P''.prefix ← P'
13:    P''.vertex ← v
14:    if u.layer ≠ v.layer then
15:        P''.costLB ← P'.costUB + w(P'.vertex, v)
16:        P''.length ← 0
17:    else
18:        P''.costLB ← P'.costLB + w(P'.vertex, v)
19:        P''.length ← P'.length + dist(P'.vertex, v)
20:    P''.costUB ← P''.costLB+
           MinAreaOverhead(P''.length, v.hasSpace)
21:    if P''.costLB < v.costUB then
22:        Push P'' into Q
23:        if P''.costUB < v.costUB then
24:            v.costUB ← P''.costUB
```

a path from vertex v_1 via v_2 to v_3 is simply the sum of the cost of the two partial paths:

$$cost(v_1 \rightsquigarrow v_2 \rightsquigarrow v_3) = cost(v_1 \rightsquigarrow v_2) + cost(v_2 \rightsquigarrow v_3).$$

The challenge for considering the minimum area constraint is an uncertain cost of a partial path, which is unknown until the path turns or stops. At vertex v_2, it is unknown whether a minimum-area overhead (either wire extension or violation penalty) is needed, which depends on the future propagation of the path. However, for a path up to a certain wire segment, bounds on its cost can be calculated as follows.

- Lower bound cost: sum of edge costs plus the minimum-area overhead on all the previous wire segments.
- Upper bound cost: lower bound cost plus the potential minimum-area overhead on the current wire segment.

Our path search is detailed by Algorithm 1. The process is still based on a priority queue Q, but the operation domain is generalized from vertices to paths, because each vertex may have several candidate paths now. The information stored for a partial path P' includes:

- Prefix path $P'.prefix$ and current vertex $P'.vertex$. Note that such incremental storage requires $O(1)$ memory only for each propagated path, instead of $O(|P'|)$.
- The lower bound $P'.costLB$ and upper bound $P'.costUB$ of the path cost.
- Length of the current wire segment $P'.length$. It is needed for calculating the minimum-area overhead.

The information stored at each vertex v is the smallest upper bound cost $v.costUB$ among all the paths reaching it.

In each iteration, the path P' with the smallest lower bound cost in the priority queue Q is popped out (line 6). It will be considered for propagating to the neighbors of $P'.vertex$. For an extended path P'' to a neighbor $v \in P'.vertex.neighbors$, satisfying $P''.costLB < v.costUB$ means that P'' is a potentially optimal path and should be considered for further propagation (line 21). If $P''.costLB \geq v.costUB$, P'' can be pruned. The algorithm stops when a sink vertex is reached (line 7). Note that for a sink vertex, the pin metal is sufficiently large and thus can guarantee that $P'.costLB$ is achievable (i.e., no minimum-area overhead charged).

The overhead due to the minimum-area rule depends on the length of the current wire segment $P''.length$, whether vertex v has sufficient spare space for wire extension ($v.hasSpace$), and the minimum length requirement l_{min} (line 20). To be more specific,

$$\text{MinAreaOverhead}(P''.length, v.hasSpace) = \begin{cases} 0, & \text{if } P''.length \geq l_{min}, \\ w_{wire} \cdot (l_{min} - P''.length), & \text{if } P''.length < l_{min} \text{ and } v.hasSpace, \\ w_{minArea}, & \text{otherwise,} \end{cases}$$

where w_{wire} is the unit-length basic cost for wires, and $w_{minArea}$ is the penalty for each minimum-area violation. Note that the flag $v.hasSpace$ for all the vertices in the local grid graph can be queried from the global grid graph in batch. The flags are then stored explicitly in the direct-address table mentioned in Section 3.3.3.

Theorem 1 states the optimality of Algorithm 1. The proof is similar to that of the original Dijkstra's Algorithm (see [22]). Details are omitted here due to space limit.

Theorem 1. *For a given local grid graph $G(V, E, w)$, Algorithm 1 gives an optimal $s - t$ path P satisfying the minimum length constraint l_{min}.*

The path search algorithm in MANA [10] also captures the minimum length constraint in a similar manner. The strengths of our approach over MANA are two folds. First, our framework allows minimum-area violations to exist in earlier RRR iterations. The minimum-area penalty serves as Lagrange multiplier [3] and helps to build a smooth RRR optimization process. It avoids satisfying minimum-area constraint at a huge price of sacrificing other metrics (e.g., wire length) in early iterations but still leads to zero minimum-area violation eventually. Second, we query the flag $v.hasSpace$ in batch from our global grid graph, which is more efficient.

For a multiple-pin net, path search starts from a source pin t. When reaching the first other pin, all vertices on the path are regarded as source for searching a next pin, until all pins are reached [23].

4.3 Rip-up and Reroute

One round of sequential maze routing usually cannot generate a violation-free solution for all the nets. Several rounds of rip-up and reroute (RRR) help to iteratively reduce violations. Our RRR strategy is similar to those widely used in global routing (e.g., NCTU-GR [9]) with two differences. First, only nets with violations are ripped up to save runtime, considering that detailed routing is more time-consuming. Second, for ripped-up nets, their routing regions will be slightly expanded for attempting a larger solution space in the next iteration.

5 PARALLELISM

The turn-around time of the detailed routing step can be further shortened by routing different nets in parallel. The challenge here is that the routing regions of different nets may overlap. We design an efficient bulk synchronous parallel scheme [24]. It routes batches of independent nets one after another.

Algorithm 2 Scheduling for Parallel Routing

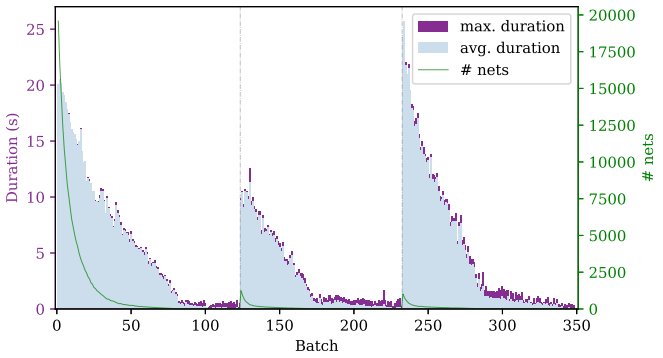
Require: Nets
Ensure: $batchList$
1: $batchList \leftarrow \emptyset$
2: **for** each net n_i **do**
3: **for** each batch b_j in $batchList$ **do**
4: **if** n_i has no conflict with b_j **then** ▷ By R-trees
5: Add n into b_j
6: Break
7: **if** n_i has not been assigned to any batch **then**
8: Append a single-net batch with n_i to $batchList$
9: **for** each batch b_j in $batchList$ **do**
10: Sort nets in b_j by decreasing size of routing region.

For nets in the same batch, their routing regions do not overlap. Here a *safety margin* is also considered, which captures spacing rules and possible wire extension for minimum-area compliance. There are two phases for each batch. The *routing phase* queries nets from the global grid graph, constructs the local grid graphs, and runs maze routing; the *committing phase* records routed edges into the global grid graph (see Fig. 2), which can be regarded as a data synchronization needed by later batches. The parallelism for the independent jobs in either routing or committing phase is trivial: each thread keeps consuming a net from a pool of unprocessed nets until the pool becomes empty. With runtime dominated by the routing phase, the reason for having a separate committing phase is to avoid a heavy usage of mutual exclusion (mutex) [25] among threads. Routed edges in the global grid graph are stored by BSTs. A BST cannot be accessed when it is being modified by another thread, even if the ranges of access and modification do not overlap. One solution is to set up locks. Its drawback is that reading BSTs is significantly more frequent than writing. Note that for a net, reading BSTs is performed on its routing region, while writing is only performed for the solution paths, which comprise just a small part of the whole routing region. By separating the committing phase, the BST reads in the routing phase become lock-free and thus can be performed faster.

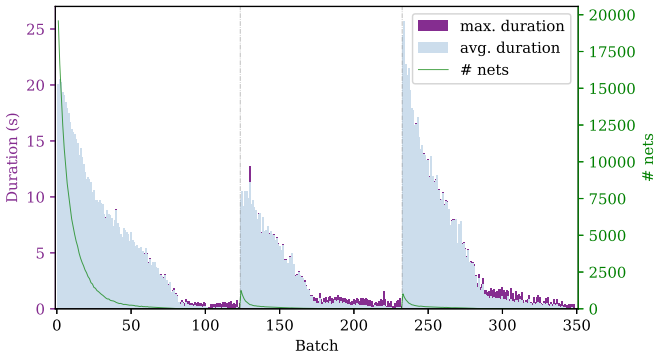
A scheduling of all the batches will be performed in the beginning of a RRR iteration by Algorithm 2. Nets are assigned one after another by trying to join an existing batch (lines 3–6) and thus minimizing the number of batches. R-trees [26] are used to detect the conflict between a net and a candidate batch. For a batch of nets, there are several R-trees storing their rectangular routing regions, one for each layer. In this way, the scheduling is very efficient and empirically only takes 0.6–1.4% of the total running time. Fig. 6(a) shows the runtime profile of all the batches on a test case. Note that in a batch, different threads may finish their last jobs at different time and thus has various durations. The maximum duration of all the threads is the time that a batch needs, while the average duration is the runtime lower bound that can be achieved by an ideal scheduling. Their small difference shown in Fig. 6 justifies the good quality of our scheduling. Moreover, the workload of different threads in a batch can be more balanced by processing larger nets first (lines 9–10). Improvement is further evidenced by the smaller gaps between the maximum and the average durations of the batches in Fig. 6(b).

6 EXPERIMENTAL RESULTS

Our detailed router is implemented in C++ with the boost geometry library [27] for R-tree query. Experiments are performed on a 64-bit Linux workstation with Intel Xeon 3.4 GHz CPU and 32 GB memory. Benchmarks are from ISPD 2016 Initial Detailed Routing Contest [2],



(a) Without balancing



(b) With balancing

Figure 6: Better parallelism by load balancing. The result is on case `ispd18_test9` with 178,857 nets and 349 batches across three RRR iterations.

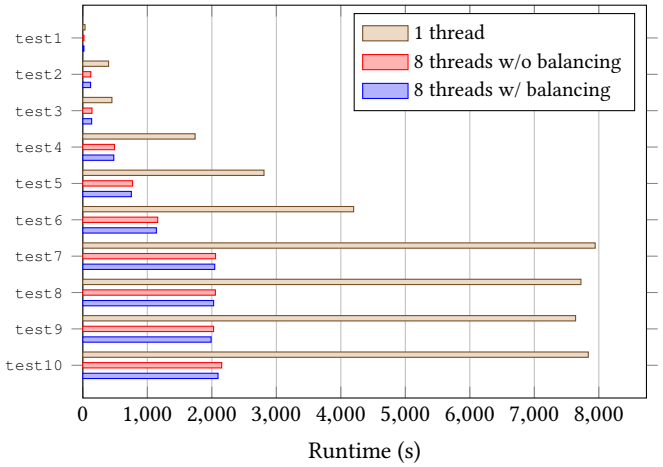


Figure 7: Speed-up by parallelism.

where the largest case `ispd18_test10` has 290,386 standard cells and 182,000 nets. Consistent with the contest, eight threads are used by default. The result reporting is conducted by Cadence Innovus.

The acceleration achieved by our parallelism is shown in Fig. 7. Eight threads give almost four times speed-up, where the load balancing contributes 2.52% improvement on average.

Table 1: Result Comparison with the First Place of ISPD 2018 Contest

		WL	# vias	Non-preferred usage					Design rule violations			Quality score	Mem (GB)	Time (s)
				Out-of-guide		Off-track		Wrong-way	Short	#	# min			
				WL	# vias	WL	# vias	WL	area	spacing	area			
Metric weight		0.5	2	1	1	0.5	1	1	500	500	500	-	-	-
Dr. CU	test1	434914	34443	4352	859	276	0	2363	15	122	0	**362725**	**0.32**	**17**
	test2	7817285	339055	104720	11784	4353	0	22023	1330	1949	0	6366886	**1.15**	**121**
	test3	8707641	331958	176736	10731	4344	0	22187	1982	2419	0	**7430092**	**1.25**	**139**
	test4	26042785	701994	769265	31444	41791	0	89537	26329	11224	0	**34112928**	**2.89**	**494**
	test5	27852167	942588	649224	43071	13390	0	63397	4722	7742	0	**22805761**	**3.87**	**767**
	test6	35813473	1446807	976672	68656	20357	0	95811	12891	11023	0	**33908653**	**5.16**	**1155**
	test7	65360688	2349580	2187794	101866	33105	0	170316	33041	14880	0	**63816462**	**8.86**	**2071**
	test8	65668468	2360231	2288159	102982	33373	0	170583	22353	14384	0	**58501486**	**8.92**	**2060**
	test9	54993356	2358857	1604576	115465	29620	0	168722	17316	14470	0	**50010785**	**8.52**	**2016**
	test10	68282001	2532666	2826908	140343	32865	0	180586	150705	20837	0	**128141527**	**8.98**	**2132**
	Avg. ratio	1.00	1.00	1.00	1.00	1.00	-	1.00	1.00	1.00	-	**1.00**	**1.00**	**1.00**
1st place of ISPD 2018	test1	472032	41641	6246	1385	3528	116	3509	1	107	0	386188	4.64	207
	test2	8150588	409551	71685	13451	20402	1362	18214	95	1158	1	**5636274**	32.55	1514
	test3	9086139	427410	69182	2450	33470	1216	18882	4891	1387	0	8645534	43.40	2019
	test4	27514053	858224	240226	8841	150961	1011	224715	52947	50957	6	67978777	46.52	4706
	test5	29151781	1140804	309785	30902	45523	10656	193203	28199	66250	22	64660336	24.25	1914
	test6	37987679	1775407	467961	42448	153900	17644	281060	30949	100229	12	89025895	28.40	3107
	test7	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail
	test8	69559382	2929578	1006247	82478	375236	22294	455824	76790	161229	48	161426598	41.81	6262
	test9	58803453	2920259	813750	67367	331766	22915	446432	56581	158305	40	144221466	40.16	5128
	test10	72244024	3110163	1414338	81831	625291	27392	476670	120966	177426	33	193867714	45.15	5554
	Avg. ratio	1.06	1.23	0.58	0.73	9.02	-	2.18	2.28	6.10	-	1.97	13.27	6.89

We also compare our results with the first place in ISPD 2018 Contest (TABLE 1)[2]. Regarding the routing quality, our router show significantly better scores in many aspects (including wire length, via count, design rule violations, off-track and wrong-way usage) in most cases. In ISPD 2018 Contest, the total quality score is a weighted sum of different metrics (weights are shown in TABLE 1). According to this metric, our routing quality wins the first place in nine out of ten test cases (35% better on average with their failed case excluded). At the same time, our router takes 80–93% less memory, and has 2.5–15× speed-up.

7 CONCLUSION

In this paper, we propose an efficient and effective detailed router to tackle the challenges in detailed routing. A set of two-level sparse data structures is designed for the routing grid graph of enormous size. An optimal path search algorithm is proposed to handle the minimum-area constraint. Besides, an efficient bulk synchronous parallel scheme is adopted to further reduce the runtime usage. Compared with the first place of ISPD 2018 Contest, our router shows superior routing quality, runtime, and memory usage.

ACKNOWLEDGMENT

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. CUHK14208914).

REFERENCES

[1] Y. Zhang and C. Chu, "RegularRoute: An efficient detailed router applying regular routing patterns," *IEEE TVLSI*, vol. 21, no. 9, pp. 1655–1668, 2013.
[2] S. Mantik, G. Posser, W.-K. Chow, Y. Ding, and W.-H. Liu, "ISPD 2018 initial detailed routing contest and benchmarks," in *Proc. ISPD*, 2018, pp. 140–143.
[3] J. A. Roy and I. L. Markov, "High-performance routing at the nanometer scale," *IEEE TCAD*, vol. 27, no. 6, pp. 1066–1077, 2008.
[4] Y. Xu, Y. Zhang, and C. Chu, "FastRoute 4.0: global router with efficient via minimization," in *Proc. ASPDAC*, 2009, pp. 576–581.
[5] M. Cho, K. Lu, K. Yuan, and D. Z. Pan, "BoxRouter 2.0: A hybrid and robust global router with layer assignment for routability," *ACM TODAES*, vol. 14, no. 2, p. 32, 2009.

[2] The scores of the first place are obtained from the contest organizer. The machine configuration is slightly different. But on a same binary, the results reported by the contest organizer have the same detailed scores as ours except runtime and memory usage. The result reported by them is 13.3% faster and takes 52.2% more memory on average.

[6] M. M. Ozdal and M. D. Wong, "Archer: A history-based global routing algorithm," *IEEE TCAD*, vol. 28, no. 4, pp. 528–540, 2009.
[7] T.-H. Wu, A. Davoodi, and J. T. Linderoth, "GRIP: Global routing via integer programming," *IEEE TCAD*, vol. 30, no. 1, pp. 72–84, 2011.
[8] M. Gester, D. Müller, T. Nieberg, C. Panten, C. Schulte, and J. Vygen, "BonnRoute: Algorithms and data structures for fast and good vlsi routing," *ACM TODAES*, vol. 18, no. 2, p. 32, 2013.
[9] W.-H. Liu, W.-C. Kao, Y.-L. Li, and K.-Y. Chao, "NCTU-GR 2.0: Multithreaded collision-aware global routing with bounded-length maze routing," *IEEE TCAD*, vol. 32, no. 5, pp. 709–722, 2013.
[10] F.-Y. Chang, R.-S. Tsay, W.-K. Mak, and S.-H. Chen, "MANA: A shortest path maze algorithm under separation and minimum length nanometer rules," *IEEE TCAD*, vol. 32, no. 10, pp. 1557–1568, 2013.
[11] M. Ahrens, M. Gester, N. Klewinghaus, D. Müller, S. Peyer, C. Schulte, and G. Tellez, "Detailed routing algorithms for advanced technology nodes," *IEEE TCAD*, vol. 34, no. 4, pp. 563–576, 2015.
[12] M. M. Ozdal, "Detailed-routing algorithms for dense pin clusters in integrated circuits," *IEEE TCAD*, vol. 28, no. 3, pp. 340–349, 2009.
[13] T. Nieberg, "Gridless pin access in detailed routing," in *Proc. DAC*, 2011, pp. 170–175.
[14] X. Xu, Y. Lin, V. Livramento, and D. Z. Pan, "Concurrent pin access optimization for unidirectional routing," in *Proc. DAC*, 2017, p. 20.
[15] Q. Ma, H. Zhang, and M. D. Wong, "Triple patterning aware routing and its comparison with double patterning aware routing in 14nm technology," in *Proc. DAC*, 2012, pp. 591–596.
[16] Y.-H. Lin, B. Yu, D. Z. Pan, and Y.-L. Li, "Triad: A triple patterning lithography aware detailed router," in *Proc. ICCAD*, 2012, pp. 123–129.
[17] Z. Liu, C. Liu, and E. F. Young, "An effective triple patterning aware grid-based detailed routing approach," in *Proc. DATE*, 2015, pp. 1641–1646.
[18] Y. Ding, C. Chu, and W.-K. Mak, "Self-aligned double patterning-aware detailed routing with double via insertion and via manufacturability consideration," *IEEE TCAD*, vol. 37, no. 3, pp. 657–668, 2018.
[19] Y.-H. Su and Y.-W. Chang, "VCR: Simultaneous via-template and cut-template-aware routing for directed self-assembly technology," in *Proc. ICCAD*, 2016, pp. 49:1–49:8.
[20] C. Albrecht, "Global routing by new approximation algorithms for multicommodity flow," *IEEE TCAD*, vol. 20, no. 5, pp. 622–632, 2001.
[21] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
[23] C. J. Alpert, D. P. Mehta, and S. S. Sapatnekar, Eds., *Handbook of algorithms for physical design automation*. CRC press, 2008.
[24] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[25] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.
[26] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. SIGMOD*, 1984, pp. 47–57.
[27] Boost geometry library. [Online]. Available: https://www.boost.org/doc/libs/1_67_0/libs/geometry/doc/html/