Parallelizing Post-Placement Timing Optimization

Jiyoun Kim[1], Marios C. Papaefthymiou[1], and Jose L. Neves[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI 48109
{jiyoun, marios}@eecs.umich.edu

[2]IBM Server Group
2455 South Road
Poughkeepsie, NY 12533
jneves@us.ibm.com

Abstract

This paper presents an efficient modeling scheme and a partitioning heuristic for parallelizing VLSI post-placement timing optimization. Encoding the paths with timing violations into a task graph, our novel modeling scheme provides an efficient representation of the timing and spatial relations among timing optimization tasks. Our new partitioning algorithm then assigns the task graph into multiple sessions of parallel processes, so that interprocessor communication is completely eliminated during each session. This partitioning scheme is especially useful for parallelizing processes with heavily connected tasks and, therefore, high communication requirements. For circuits with 20–130 thousand cells, the partitioning heuristic achieves speedups in excess of 5$\times$ without degrading solution quality by dynamically utilizing 1–8 processors.

Keywords: Timing Optimization, Parallel VLSI Design, Physical Design, Partitioning

1 Introduction

In an automated VLSI design flow, post-placement timing optimization is the step where paths with excessive delays are optimized to meet given timing specifications. Since the operating speed of a circuit is determined by its largest path delay, timing optimization is an important phase that guarantees the performance of the resulting VLSI system. As chip density and design complexity rapidly grow, this optimization step is becoming increasingly time consuming. In today's VLSI designs, it is common to have chips with several million gates, requiring tens or hundreds of hours per post-placement timing optimization run. Since timing optimization may be performed numerous times until design specifications are met, reducing its runtime without degrading solution quality may significantly impact the design development cycle and yield shorter time-to-market.

Partitioning can speedup a process by enabling parallel processing. To that end, partitioners find balanced partitions of tasks and assign each partition to one of multiple processors. There have been investigations of parallel databases for CAD [9] and parallel algorithms for various CAD tasks, such as placement [2, 7, 12], routing [6, 8] and logic simulation [4, 10]. In post-placement timing optimization, however, tasks are heavily connected and less amenable to efficient parallelization, due to the excessive need for communication and/or synchronization.

This paper introduces a new multi-session partitioning technique for parallel post-placement timing optimization that eliminates interprocessor communication during task computation. At each session, the partitioning algorithm determines path subsets that need not exchange any timing or geometric information to achieve optimal timing optimization results. Consequently, throughout each session, each processor proceeds uninterrupted with the optimization of its assigned partition, without incurring any latency or synchronization overhead due to communication events. All information necessary for subsequent sessions is updated only at the end of each session. Thus, the proposed partitioning scheme provides for structured and highly synchronized parallel processing with minimum communication overhead.

Sun and Sechen have presented a parallel placement scheme without processor interaction during each iteration [12]. This scheme does not remove conflicts during an iteration, however, and thus may not be suited for post-placement procedures. Conflicts (i.e.,overlaps) in post-placement optimizations are more difficult to resolve than conflicts during placement without degrading design quality. The motivation behind our work is to completely remove communication and conflicts during computation by introducing multiple sessions of distribute-compute-merge steps. Since simultaneously processed tasks have no conflicts in their resources, there is no competition that would cause quality degradation for one or more task optimization

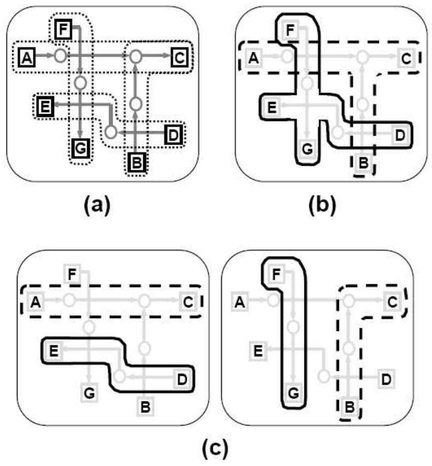


Figure 1. Task partitioning example. (a) Given four paths. (b) Conventional partitioning. (c) Multi-session (2-session) partitioning.

results. To our knowledge, this is the first work addressing the partitioning and parallelizing of the post-placement timing optimization process.

A simple example of timing optimization parallelization is given in Fig. 1. Fig. 1(a) shows a set of four timing paths ($A \rightsquigarrow C, B \rightsquigarrow C, D \rightsquigarrow E, F \rightsquigarrow G$) to be optimized, each of which is associated with physical space specified by dotted boundary. Timing optimization of each path changes the physical space associated with it. For paths that share logic (e.g., $A \rightsquigarrow C$ and $B \rightsquigarrow C$) and/or space (e.g., $A \rightsquigarrow C$ and $F \rightsquigarrow G$), the communication of timing and spatial information among them becomes necessary. When using two processors P1 and P2 for the parallel optimization of the four given paths, it is not possible to find two independent partitions. Conventional partitioners such as METIS [5] will give results akin to this in Fig. 1(b), where paths $A \rightsquigarrow C, B \rightsquigarrow C$ and associated area (dashed line) are assigned to P1, and paths $D \rightsquigarrow E, F \rightsquigarrow G$ and associated area (solid line) are assigned to P2. Since the areas of the two partitions overlap, communication between the two processors is unavoidable.

Fig. 1(c) shows the result of our multi-session partitioning. In the first session, path $A \rightsquigarrow C$ is assigned to P1, and path $D \rightsquigarrow E$ is assigned to P2. After the paths on P1 and P2 are processed independently of each other, at the end of the first session the changes are merged into the main design and propagated to the following session. In the next session, the remaining tasks are independently processed in the same fashion. By guaranteeing that no pair of tasks sharing the same logic or space are processed on two different processors during the same session, the need for interprocessor communication during task computation is eliminated.

This paper first presents a task graph modeling scheme which efficiently represents timing and spatial dependencies among post-placement timing optimization tasks. It then defines the multi-session partitioning problem on the task graph, so that task optimization during each session is guaranteed to proceed uninterrupted. After giving a proof that the multi-session partitioning problem is *NP*-hard, it proceeds to present a greedy, yet efficient heuristic for multi-session partitioning.

In timing optimization experiments with placed VLSI circuits comprising up to 130 thousand cells, our graph modeling and multi-session partitioning heuristic achieves speedups up to 5.14× without degrading solution quality, utilizing 1–8 processors. Speedups are strongly correlated with graph connectivity. Specifically, increased speedups are achieved when the connectivity of the optimization tasks diminishes.

The remainder of this paper is organized into six sections. In Section 2, we review the timing analysis and optimization problem. Our efficient graph modeling for timing optimization is introduced in Section 3. Section 4 gives a formal definition of the multi-session partitioning problem and a proof that it is *NP*-complete. Our efficient heuristic for the multi-session partitioning problem is given in Section 5. Heuristic evaluation and speedups from parallelizing post-placement timing optimization of several VLSI designs are presented in Section 6. A discussion of ongoing work is presented in Section 7.

2 Review of Timing Optimization

In this section, we give a brief review of basic concepts in the timing analysis and optimization procedure for VLSI design. Relying on these concepts, we describe the timing optimization graph modeling and multi-session partitioning problem in Sections 3 and 4.

In automated VLSI design flows, engineers first generate a design using a behavioral *hardware description language* that specifies the functional behavior of the system. Subsequently, a *logic synthesis* step automatically generates a gate-level netlist by taking pre-designed gates from a *gate library*. The netlist is then sent to a *placement* tool, which arranges design components in a physical space. In the subsequent *routing* step, the interconnects among circuit components are routed. The layout of a placed and routed design is finally generated and used in manufacturing the chip.

Our work focuses on the timing optimization process that takes place after placement. Exact delays can be calculated only after circuit components have been physically placed in 2-dimensional space, since physical locations of components must be known for derivation of interconnect delays. Post-placement timing analysis and optimization steps use accurate delay information to improve the speed of a given placed circuit, which would be very close to the speed of final chip products.

First, timing analysis determines if a circuit meets all

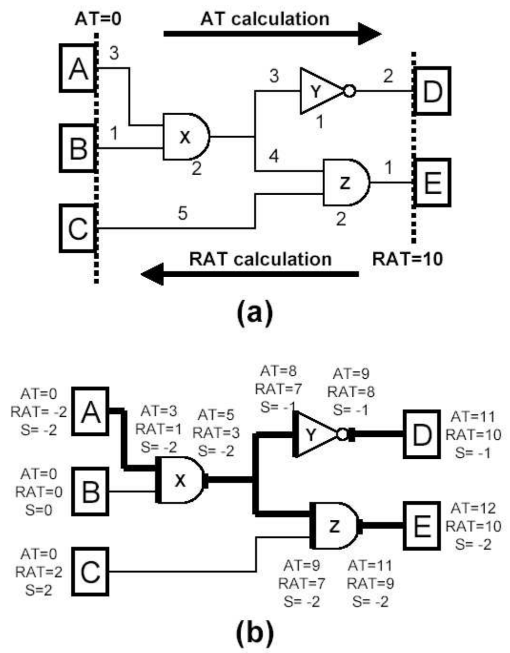


Figure 2. (a) An example circuit with delays and (b) its calculated AT, RAT and slack values

timing goals. Based on the delay information of cells and nets, timing analysis calculates the delays of all timing paths. A timing path is an ordered sequence, through gates and interconnects, of timing points between two *significant timing points (STPs)*. At each STP, such as the input/output pins of flip-flops and the primary inputs/outputs of the design, a timing goal is defined by asserting timing information. If a timing path fails to meet the timing assertion at its STPs, the path is called *critical* and becomes the subject of subsequent timing optimization.

All critical paths are identified during the timing analysis process. At each timing point, a timing analysis tool calculates *Arrival Time (AT)*, the absolute time at which all signals actually arrive, and *Required Arrival Time (RAT)*, the absolute time at which the signals are required to arrive. The *slack S* of the timing point is then obtained as $S = RAT - AT$. The slack indicates how far the timing point in the path is from its goal. If $S = 0$, the point has reached its goal, if $S > 0$, the point is beyond the goal, and if $S < 0$, the point has yet to meet the goal.

Fig. 2 gives an example of timing analysis on a simple circuit with timing goal $T = 10$. Fig. 2(a) shows STPs $\{A, B, C, D, E\}$ with a given timing assertion ($AT = 0$ for launching STPs $\{A, B, C\}$, $RAT = 10$ for receiving STPs $\{D, E\}$) and intermediate gates $\{X, Y, Z\}$. Delays for gates and nets are denoted by integers next to corresponding elements. ATs are calculated starting from launching STPs to receiving STPs along the direction of signal flow. RATs are calculated in the opposite direction. Fig. 2(b) provides

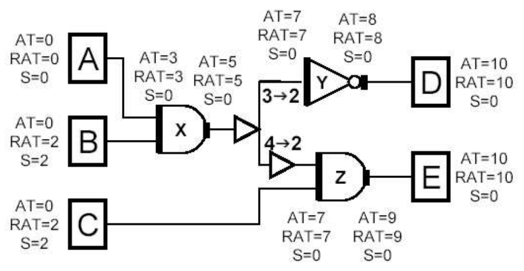


Figure 3. Timing optimization for circuit in Fig. 2.

AT/RAT/slack values at every timing point. When multiple signals meet at a timing point, the latest AT and the earliest RAT become the AT and RAT of the timing point. For example, at the input of gate X, incoming signals from A and B have $AT = 3$ and $AT = 1$, respectively. Therefore, $AT = \max\{3, 1\} = 3$. Note that all timing points on paths $\{A, X, Y, D\}$ and $\{A, X, Z, E\}$ have negative slack values. These two paths are critical.

After timing analysis identifies all critical paths, post-placement timing optimization commences to improve negative slacks of those paths/STPs. Timing optimization tools apply a variety of circuit transformation techniques such as gate sizing, buffer insertion, and gate relocation [1, 11]. Gate sizing powers up a slow gate by replacing it with a larger gate in order to reduce gate delay. To optimize interconnect delays, buffers may be inserted on slow nets to cut down delays which grow quadratically with wire length. If the cells connected to a slow net are placed too far to meet the timing requirement, the optimization tool may move the cells closer to shrink the span of the net. Typically, timing optimization routines have superlinear runtime complexity, resulting in significant runtimes. Consequently, optimizing the timing of a design with several million gates may take hundreds of hours.

Fig. 3 shows the optimization result when two buffers are inserted on the net connecting $\{X, Y, Z\}$, to decrease $X \rightarrow Y$ and $X \rightarrow Z$ interconnect delays. The $AT/RAT/S$ values on the paths passing through that net change, and slacks on the two critical paths become non-negative. Since all paths have delays no greater than 10, the timing optimization stops here. It can be observed that optimization techniques applied to a cell or a net change not only the timing information of the optimized element, but also the timing information of all paths passing through that element.

The spatial configuration of the circuit is also affected by timing optimization. Fig. 4 illustrates how the buffer insertion in Fig. 3 may change the original placed circuit. In addition to gates X, Y and Z, the circuit area in Fig. 4(a) includes other placed cells which did not participate in the optimization. When two buffers are inserted, the desired locations do not have enough space, so nearby cells need

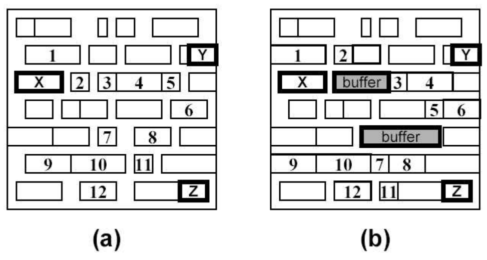


Figure 4. Physical view of the circuit in Fig. 3. (a) before insertion and (b) after insertion.

to be moved. Cells which have been shifted or moved to a different row are labeled with integer numbers. As can be seen in this example, during timing optimization, the spatial configuration is affected not only for the optimized cell or inserted buffer but also for some of the nearby cells.

The timing/spatial relations among paths and their associated area during post-placement timing optimization must be considered when partitioning the optimization tasks to parallelize the process. Following are definitions of terms used in this paper to describe these relations.

Definition 2.1 *Two (sets of) timing paths are* ***timing dependent*** *if timing optimization of one of these (sets of) paths can change the timing information of the other path (set), due to their sharing of logic.*

Definition 2.2 *Two (sets of) timing paths are* ***spatially dependent*** *if optimization of one of these (sets of) paths affects the spatial configuration of the other path (set).*

Each path is associated with its own elements and nearby cells, whose spatial information can be altered when timing optimization is applied to the path. If two paths are in the same physical vicinity so that their associated spaces are overlapping, timing optimization of one path can cause changes in the spatial information of the other path.

Note that timing dependence always implies spatial dependence, due to the shared logic. The implication is not true in the opposite direction, however.

Definition 2.3 *A set of timing paths is* ***self-contained*** *for the timing optimization of a timing point if the set includes all the paths necessary to improve the slack of the timing point.*

The slack of a timing point is the worst slack among all paths passing through that timing point. Therefore, to fix the negative slack of a timing point, all the critical paths passing through that timing point must be optimized to have positive slacks. The set of these paths is then self-contained for the optimization of that timing point. In the example of Fig. 2 and Fig. 3, to improve the slack of STP A, all the critical paths contributing to it need to be optimized. Therefore, the set of paths $\{\{A,X,Y,D\},\{A,X,Z,E\}\}$ is self-contained for the timing optimization of STP A.

3 Task Graph Model

In this section, we introduce a task graph for encoding partitioning constraints related to a given timing optimization problem. We also describe an efficient algorithm for generating it. This graph is subsequently used in Section 5 to derive an efficient task partitioning for timing optimization.

The task graph is constructed from a given placed circuit represented by a *placed circuit graph* $H(C,N)$, where each vertex $c \in C$ corresponds to a circuit cell, and each hyperedge $n \in N$ corresponds to a net connecting circuit cells. Since H is placed, the cells and nets are associated with physical space on the chip. Thus, for every cell $a \in C$ or net $a \in N$, the *bounding box* $BB(a)$ is defined as a minimum-size rectangle containing the element a with vertical/horizontal coordinates.

For any given placed circuit, we introduce a two-dimensional grid structure designed to provide geometric guidelines for timing optimization. When a cell is sized or inserted into a grid location, only the elements in that grid are allowed to be moved around within that grid boundary. The rippling of geometric shifts is thus restricted, and local changes do not globally affect chip area. Each grid rectangle contains a roughly equal number of cells, typically up to a few hundred cells.

The grid structure is also used to represent the physical space occupied by circuit elements. Every cell $a \in C$ or net $a \in N$ is associated with a two-dimensional boolean map $B_a(,)$ derived from the bounding box of a. The boolean map $B_a(i,j)$ is set to '1' when the (i,j)th circuit grid rectangle is partially or entirely covered by $BB(a)$, or set to '0' otherwise. This boolean map indicates the boundary within which the geometric information of circuit components can be changed when a is optimized for timing.

During timing analysis, timing violating cells and nets with negative slacks are marked and later extracted to construct a *timing violation graph* $H_{TV}(C_{TV},N_{TV})$, where $C_{TV} \subset C$ and $N_{TV} \subset N$ are timing violating cells and nets, respectively. Each timing violating element (whether a cell or a net) $a \in H_{TV}(C_{TV},N_{TV})$ is associated with timing analysis information $AT(q), RAT(q), S(q)$, providing AT, RAT, and slack values for each input/output pin q of a. The same boolean grid map information $B_a()$ derived in $H(C,N)$ is again associated with $a \in H_{TV}$.

The timing violation graph H_{TV} embeds all critical paths and all timing and spatial information needed for timing optimization. From H_{TV}, the undirected task graph $G(V,E)$ is constructed so that the task vertices and task edges represent critical path optimization tasks and the interactions between them, respectively. Efficient task graph representation should result in minimum interactions among task partitions when graph partitioning is applied to the task graph.

3.1 Task vertex generation

Each *task vertex* of the task graph G corresponds to a critical launching STP and all critical paths driven from that STP. Since a vertex is always assigned to a single partition, all timing paths contributing to that STP stay in the same partition, and the timing optimization process can improve the negative slack of that STP without importing any timing information from other partitions. Therefore, the partition is self-contained for optimizing slacks of launching STPs that are included in it.

Fig. 5(a) gives an example of task vertex generation. Launching STPs $\{A,B\}$ and receiving STPs $\{C,D,E\}$ are drawn as squares, and intermediate gates $\{1,2,..,6\}$ are drawn as circles. Fig. 5(b) and 5(c) show results from two possible task vertex generation methods. The *receiving-STP method* clusters all paths driving a common STP, so that the generated vertex is self-contained for the receiving STP. Similarly, the *launching-STP method* clusters all paths driven from a common STP, and the generated vertex is self-contained for the launching STP. Other clustering methods are possible; for example, generating a vertex for each critical path, or clustering all timing dependent paths into one vertex. Those methods are not considered, because they generate too many or too few vertices to be handled by a partitioning tool.

Among the two methods considered, the launching STP method is selected in this paper because it always includes every net into a task vertex in its entirety, thus facilitating the application of timing optimization techniques to the entire net. For example, a multi-sink net connecting cells $\{1,2,5\}$ of Fig. 5(a) could not be entirely included either in vertex X (path $\{A,1,2,C\}$) or vertex Z (paths $\{A,1,5,D\}$ and $\{B,3,4,5,D\}$) using the receiving STP scheme (Fig. 5(b)). On the other hand, with the launching STP based scheme, the same net is entirely included in vertex U (paths $\{A,1,2,C\}$ and $\{A,1,5,D\}$) in Fig. 5(c), enabling timing optimization of the net as a whole.

The total number of vertices in the task graph equals the number of launching STPs with negative slack, since each task vertex has exactly one launching STP. For the *i*th critical launching STP_i, the corresponding task vertex $v_i \in V$ can be represented as a subgraph $H_i(C_i,N_i)$ of $H_{TV}(C_{TV},N_{TV})$, where $C_i \subset C_{TV}$ and $N_i \subset N_{TV}$ are sets of timing violating cells and nets reachable from STP_i. All critical paths contributing to STP_i are then included in H_i.

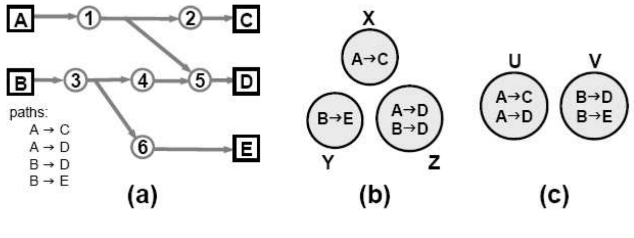


Figure 5. Task vertex generation: (a) Given critical timing paths, (b) receiving-STP method, (c) launching-STP method.

The subgraph H_i can be obtained from H_{TV} for cells and nets that are reachable for each STP_i. To identify H_i for all launching STPs in a single search, the reachable subgraph can be incrementally calculated for every timing violating cell $c \in C_{TV}$ by traversing H_{TV} in reverse topological-sort order. When a cell is visited, subgraphs from sink cells (cells driven by the cell currently visited) are merged to create the subgraph for that cell.

For each task vertex $v_i \in V$, its two-dimensional boolean map B_{v_i} is obtained to indicate the occupancy of v_i on the grid by the bitwise-OR of the boolean maps of all timing violating cells and nets v_i includes. Formally, we define $B_{v_i} = (\bigcup_{c \in C_i} B_c) \cup (\bigcup_{n \in N_i} B_n)$, where $H_i(C_i,N_i)$ is the subgraph for v_i. The boolean maps can be calculated in the same way subgraphs $\{H_i\}$ are obtained in task vertex generation, by also providing a boolean grid map of each reachable subgraph for each timing violating cell.

After $H_i(C_i,N_i)$ is identified, the task vertex v_i is associated with a vertex weight $w(v_i)$ which represents the optimization workload for the tasks in v_i. One way to intuitively assign weights is to count the number of paths requiring optimization in each vertex. However, other metrics can be applied if they estimate the workload of each task.

3.2 Task edge generation

To represent dependencies among partitions, a *task edge* is inserted between any two task vertices that are timing/spatially dependent. Since graph partitioning minimizes edge cutsize, dependency among partitions is minimized when the graph partitioning algorithm is applied to the task graph.

Since timing dependence implies spatial dependence, as mentioned in Section 2, it is sufficient for the task edge generation procedure to only check for spatial dependence. In our modeling scheme, the spatial dependence between two task vertices is determined by computing the bitwise-AND of the boolean maps of the two vertices. Let $u,v \in V$ be two different task vertices. Then a task edge $e(u,v) \in E$ if and only if $B_u \cap B_v \neq \overline{0}$. If the result is not all-zero, then two task vertices need to work in the same grids, corresponding to non-zero elements and are thus spatially dependent.

Fig. 6 gives an illustration of task edge generation. Fig. 6(a) shows five placed STPs $\{A,B,C,D,E\}$ and critical paths $A \rightsquigarrow B$, $A \rightsquigarrow C$, and $D \rightsquigarrow E$ on the circuit grid. The task graph for this circuit is shown in Fig. 6(b). First, vertices X and Y are generated by the launching-STP based clustering scheme described in Subsection 3.1. The boolean maps B_X and B_Y are derived from the union of the boolean maps of

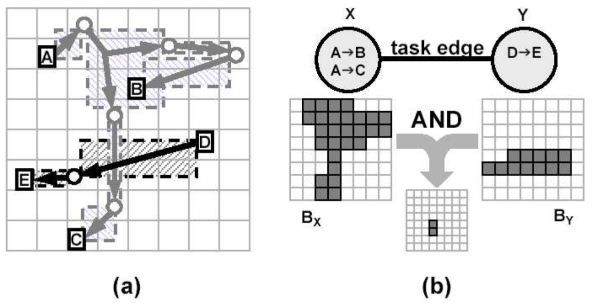


Figure 6. (a) An example of critical paths in a placed circuit and (b) corresponding task vertex/edge generation.

cells and nets included in task vertices X and Y respectively. Note that bounding boxes are used to represent locations of nets. A task edge between vertices X and Y is then inserted since $B_X \cap B_Y \neq \overline{0}$, indicating that the two tasks share the same area.

It can be shown that the task graph is generated in $O(|E_{TV}| + |C_{TV}| \cdot S_{max} + |V| \lg |V| + |V| \cdot OL_{max} \cdot S_{max}))$ time. OL_{max} is the maximum numbers of task vertices overlapping to a task vertex. S_{max} is the number of grids in the largest bounding box among bounding boxes, each of which covers a task vertex. OL_{max} varies from 0 to $|V|$ and is close to $|V|$ for small designs, but not as large in larger designs. S_{max} equals the total number of grids and is thus bounded by $O(n)$ in the worst case. Since placement algorithms tend to bound the physical spans of timing paths, however, S_{max} increases much slower than $O(n)$ in practice.

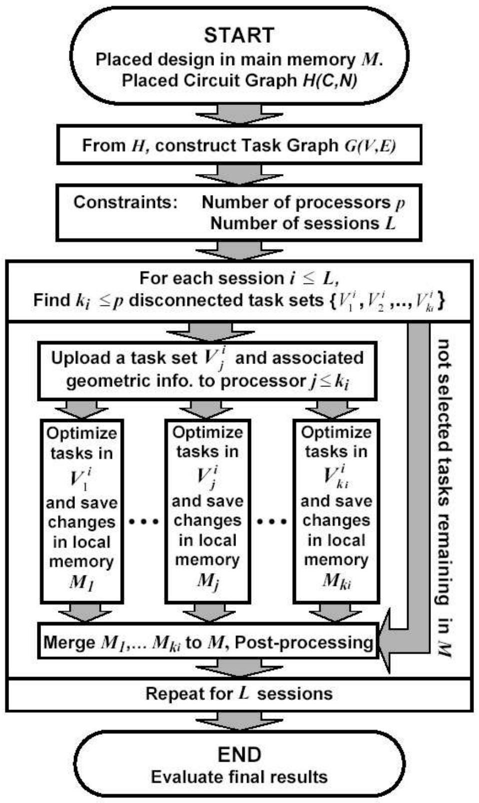
4 Multi-Session Partitioning Problem

This section defines the multi-session partitioning problem and provides a proof for its NP-completeness.

The task graph described in Section 3 guarantees self-containment of a partition. Timing/spatial independence is achieved by finding a partitioning with no task edges between partitions. When conventional graph partitioners partition dense graphs such as the task graphs derived from circuits, partitions typically have a large edge cutsize, resulting in possibly heavy communication during parallel processing.

As mentioned in Section 1, the goal of the proposed partitioning scheme is to eliminate all communication during task computation. To implement parallel processing with this constraint, a process is broken up into multiple sessions over time, with each session performing tasks in parallel.

Fig. 7 shows the flow chart for parallel timing optimization with the proposed multi-session partitioning scheme. The task set of the optimization problem is partitioned into multiple sessions. Within each session, the task subset is partitioned again into multiple processors, so that timing/spatial dependencies among partitions within each session are eliminated. All dependencies among tasks exist only among partitions in different sessions.



START
Placed design in main memory M.
Placed Circuit Graph $H(C,N)$
From H, construct Task Graph $G(V,E)$
Constraints: Number of processors p
Number of sessions L
For each session $i \le L$,
Find $k_i \le p$ disconnected task sets $\{V_1^i, V_2^i, .., V_{k_i}^i\}$
Upload a task set V_j^i and associated geometric info. to processor $j \le k_i$
Optimize tasks in V_1^i and save changes in local memory M_1
Optimize tasks in V_j^i and save changes in local memory M_j
Optimize tasks in $V_{k_i}^i$ and save changes in local memory M_{ki}
not selected tasks remaining in M
Merge M_1,... M_{ki} to M, Post-processing
Repeat for L sessions
END
Evaluate final results

Figure 7. Parallel timing optimization flow chart.

In session i, disjoint task subsets $\{V_1^i, V_2^i, .., V_{k_i}^i\}$ and the physical chip space is distributed over local memories $\{M_1, ..., M_{k_i}\}$, so that for the jth processor $(1 \le j \le k_i)$, M_j keeps only the grid area in H associated to its task set V_j^i. The timing and spatial information in each local memory is then changed independently during the optimization process by the corresponding local processor. Since there is no spatial dependency among $\{V_j^i\}$, i.e., the boolean grid maps associated with these task sets do not share any grid square in $H(C,N)$, there are no spatial conflicts among local memories.

At the end of each session, the grid areas assigned to $\{M_j\}$ are merged into the main memory M. The timing/spatial updates made in local processors should be propagated to upcoming sessions. Spatial dependency update of the entire design is naturally completed when the grid areas are merged together, yet timing update needs to be done separately in a post-processing step, by running timing analysis on the timing violating graph.

The multi-session partitioning problem can be viewed as a kind of scheduling problem. Given a task graph, a typical scheduling scheme would assign each task to a processor so that no dependent tasks are processed at the same time. For example, whenever a processor becomes available, a real-time scheduler would find a task independent of the other tasks being processed in other processors and assign it to the available processor. Compared to such scheduling schemes that assign each individual task, multi-session partitioning clusters many dependent tasks, which share physical chip space and thus memory resources, processing them together in one processor. This scheme saves the effort of memory distribution/merging actions, since the information in the memory associated with many tasks in a cluster is sent to and retrieved from the corresponding processor only at the beginning/end of corresponding session.

The Multi-Session Partitioning problem is formally stated as follows:

Problem Multi-Session Partitioning (MSP): Given task graph $G(V,E)$ with vertex weights $w(v) : V \rightarrow Z^+$ for each $v \in V$, processor number p, and maximum number of sessions L, find disjoint sets $\{V_1^1, V_2^1, ..., V_p^1\}$, $\{V_1^2, V_2^2, ..., V_p^2\}$, ..., $\{V_1^L, V_2^L, ..., V_p^L\}$ such that:

1. $\bigcup_{i=1}^{L} \bigcup_{j=1}^{p} V_j^i = V$ and $V_{j_1}^{i_1} \cap V_{j_2}^{i_2} = \emptyset$ if $(i_1, j_1) \neq (i_2, j_2)$.
2. $\forall i$, $u \in V_{j_1}^i$ and $v \in V_{j_2}^i$ implies $(u,v) \notin E$ if $j_1 \neq j_2$.
3. Minimize $T = \sum_{i=1}^{L} \max_j(W_j^i)$, where $W_j^i = \sum_{v \in V_j^i} w(v)$.

The first condition indicates $\{V_j^i\}$ is a set partition of V, where i is the session index, and j is the processor index. Some V_j^i may be empty, and thus the total effective session number may be smaller than L, and the effective processor number for each session may be smaller than p. The second condition disallows communication between processors at each session.

The third condition states the objective of the problem, which is to minimize the total parallel processing time T. At each session, the processing time is dominated by the largest workload among processors, so total processing time T is the sum of $\max_j(W_j^i)$, where W_j^i is the processing time for task subset V_j^i, and $\max_j(W_j^i)$ is the processing time needed for the ith session. Minimizing T also implicitly balances the partitions in the same session, since imbalance within each session tends to increase T.

In the decision version of Problem MSP, a positive integer $K \leq |V|$ is also given, and Condition 3 is changed as follows:

3'. $T = \sum_{i=1}^{L} \max_j(W_j^i) \leq K$, where $W_j^i = \sum_{v \in V_j^i} w(v)$.

In this case, the problem would ask if there are disjoint sets $\{V_j^i\}$ satisfying conditions 1, 2, and 3'.

In the next theorem, we briefly show that Problem MSP$_d$, the decision version of Problem MSP, is *NP*-complete.

Theorem 4.1 *For multiple processors ($p \geq 2$) and multiple sessions ($L \geq 2$), Problem MSP$_d$ is* NP*-complete.*

Proof. Can be shown by transforming the BALANCED COMPLETE BIPARTITE SUBGRAPH (BCBS) problem to Problem MSP$_d$ for the case $p = 2, L = 2, w(v) = 1$. Let us call the latter problem MSP$_d(2,2)$. Given bipartite graph $G'(V',E')$ and positive integer $K' \leq |V'|$, Problem BCBS asks for two disjoint subsets $V_1, V_2 \subseteq V'$ such that $|V_1| = |V_2| = K'$ and such that $u \in V_1, v \in V_2$ implies that $(u,v) \in E'$. BCBS is known to be *NP*-complete [3]. From $G'(V',E')$ and K', we can construct $G(V,E)$ and K for MSP$_d(2,2)$ as follows: $V = V' \cup \{w\}$, $E = E_1 \cup E_2$, where $E_1 = \{(u,v) | \ u \in V, v \in V, and \ (u,v) \notin E'\}$ and $E_2 = \{(w,v) | \ \forall v \in V\}$, and $K = |V'| + 1 - K'$. Then BCBS has a solution if and only if MSP$_d(2,2)$ has a solution. It is straightforward to verify that MSP$_d(2,2)$ is in P, and that the transformation can be done in polynomial time. Therefore, MSP$_d(2,2)$ is *NP*-complete, and thus Problem MSP$_d$ is *NP*-complete for $p \geq 2$, $L \geq 2$, and non-uniform $w(v)$. $\square$

5 Partitioning Heuristic

In this section, we describe Algorithm MSPart, our efficient and effective heuristic for Problem MSP. Given task graph $G(V,E)$, available number of processors p, and maximum number of sessions L, the proposed heuristic computes a sequence of no more than L sessions. For each session, it generates at most p balanced disjoint subsets which are not connected to each other by any edges.

The main function of MSPart is a subroutine for finding disjoint subsets for each session. This subroutine can be described with the help of Fig. 8, where the black region represents vertices already included into each set, and the gray region represents vertices adjacent to the vertices of the black region. To find k disjoint subsets $\{V_1^i, V_2^i, ..., V_k^i\}$ for session i, k disconnected vertices are chosen as seeds (Fig. 8(a)). Each subset is then greedily grown from each seed by including a vertex at a time, while maintaining disconnectivity among subsets. The seed and vertex selection tries to keep the black balance among $\{V_j^i\}$, and/or gray balance among $\{U_j^i\}$, where V_j^i is jth subset with selected vertices and U_j^i is the set of vertices only adjacent to $\exists v \in V_j^i$. Vertices in U_j^i are candidates for vertex selection for V_j^i. Maintaining balance for each of both $\{V_j^i\}$ and $\{U_j^i\}$ heuristically increases the number of vertices selected for

Figure 8. Finding subsets for a session.

MSPart(G, p, L)

```
 1: initialize i=1, inputG=G, max_procnum=p;
 2: repeat                 // for each session i,
 3:  [procnum,subsets]=FindSubsets(inputG,max_procnum);

     // parallel timing optimization
 4:  parallel_timingOpt(procnum, subsets);
 5:  merge_optResult(procnum,subsets,inputG);
 6:  timingAnalysis(inputG);

 7:  remnantG=GetRemnantGraph(inputG,procnum,subsets);
 8:  inputG=remnantG, i++; // for the next session
 9:  if i==L            // if next session is the last,
10:   then max_procnum=1;  // use single processor
11: until ( isempty(inputG)==true )
```

FindSubsets(*inputG*, *max_procnum*)

```
 1: initialize best_procnum, best_subsets;
 2: for procnum->max_procnum to 1
      // find disjoint subsets for given procnum
 3:   seeds=SelectSeeds(inputG, procnum);
 4:   subsets=GrowSubsets(inputG, seeds);
      // check if the result is good for parallelization
 5:   if goodForParallel(subsets)==true
 6:     then break;
 7: return procnum, subsets;
```

Figure 9. Algorithm MSPart.

this session. If the balance of $\{V_j^i\}$ is more critical than the balance of $\{U_j^i\}$, a vertex that gives the smallest deviation $\min(\max_j(w(V_j^i)) - \min_j(w(V_j^i)))$ is selected, where $w(V)$ is the total weight in the vertex set V. If gray balance is more critical, a vertex causing $\min(\max_j(w(U_j^i)) - \min_j(w(U_j^i)))$ is selected.

An intermediate state of the subset-generating subroutine is shown in Fig. 8(b). Fig. 8(c) shows the result of the subset search in this session. Three black regions become the disjoint subsets $\{V_1^1, V_2^1, V_3^1\}$ for session 1, and the graph under the gray region, called the *remnant graph* of this session, is sent to the next session. In the next session, the remnant graph becomes the new input task graph and the disconnected subset search is performed on this graph. A session ends when the predefined maximum session number L is reached or when the remnant graph is empty or not large enough to warrant parallelization. In these cases, the last session will find an appropriate number –typically one– of subsets to include all vertices of the remnant graph.

At each session, a lower bound on the subset size assigned to each processor is enforced, because each processor incurs a fixed overhead for task distribution in the beginning of the session and task merging at the end of the session. This constraint enforces the use of fewer processors for some sessions, rather than all the available processors.

Fig. 9 shows the pseudocode for the heuristic MSPart. At each session with an input graph `inputG` and number of processors `max_procnum`, first subroutine `FindSubsets()` finds the proper processor number and corresponding disconnected subsets in line 3. These disconnected subsets are then distributed over the processors and optimized in parallel in line 4. After all processors complete their tasks, the results are merged back into the input graph for the session (line 5), and a timing analysis is performed to update timing information for the next session (line 6). This procedure is iterated until there are no vertices left to be partitioned. When the heuristic reaches the Lth session, the input graph is assigned to a single processor, the remnant graph becomes empty, and the routine terminates.

Algorithm MSPart is greedy because at each session, it tries to find the largest possible disconnected subsets for the session without any direct consideration of subsequent sessions. Furthermore, subsets grow by greedily finding and including a vertex which most balances gray and black regions of subsets.

Algorithm `MSPart()` makes at most L calls to `FindSubsets()`. It can be shown that `FindSubsets()` runs in $O(p^2|E_i| + p|V_{sel,i}| \cdot |V_i|)$, where $G_i(V_i, E_i) =$ `inputG` and $|V_{sel,i}|$ is the number of selected vertices for ith session. Since $\sum_i |V_{sel,i}| = |V|, |V_i| \leq |V|, |E_i| \leq |E|$, overall execution takes $O(p^2L|E| + p|V|^2))$ steps. In practice, the runtime is substantially shorter than this analysis suggests, since the number of `for` iterations for large $|V_i|$in `FindSubsets()` is much smaller than p. Moreover, $|V_i|$ and $|E_i|$ decrease fast in early sessions, while the complexity analysis uses just the overestimated upper bound $|V|$ and $|E|$, respectively.

6 Experimental Results

To validate our partitioning heuristic and overall parallel timing optimization strategy, we developed a prototype path-based timing optimization code. Given tasks, the code iteratively finds the most critical path and optimizes the path delay by inserting buffers on slow nets on the path, until all paths have non-negative slacks. Buffer insertion affects both timing and spatial information. It is regarded as one of the most effective timing optimization techniques [1]. Therefore, it is most suitable for the purposes of evaluating our parallelization of post-placement timing optimization.

In our experiments we used a suite of test circuits that were placed and routed to derive wire delays. Static timing analysis was then run to obtain timing violating cells and nets. The task graphs were generated using the procedure described in Section 3, where for each vertex v, the workload $w(v)$ was set equal to the linear combination of the slack amount of the launching STP to improve and the number of critical paths that correspond to that vertex.

Some important characteristics of the test circuits and their corresponding task graphs are given in Table 1. Designs are sorted by size. To indicate task graph density, the average vertex degree and connectivity metrics are given. Connectivity indicates what portion of the graph is con-

Design	Placeable Gates	Task Graph				Modeling Time
		Task Vertices	Task Edges	Average Degree	Connect-ivity	(sec)
1	24K	559	121677	435	78%	3.04
2	67K	2265	642824	568	25%	4.46
3	85K	3463	3981240	2299	66%	41.56
4	100K	3479	1236764	711	20%	10.82
5	134K	4742	3207015	676	14%	16.57

Table 1. Test circuits.

nected to a vertex and is calculated by dividing the average degree by the total vertex count. This metric is equal to the ratio $|E|/(|V|(|V|-1)/2)$, i.e., the edge count over the maximum possible edge count. As can be seen in Table 1, the task graphs derived from our placed circuits are very dense with average degree ranging from 435 to 2285, and connectivity ranging from 14% to 78%. It can be also observed that the runtime for task graph generation increases with design size. In Design 3, edge generation takes a particularly long time since the tasks overlap particularly heavily for its size (connectivity of 66%), although connectivity tends to decrease as design size grows.

Each task graph was multi-session partitioned using our heuristic MSPart, and timing optimization was applied to each partition. While an implementation using OpenMP is under construction, the parallel processing in each session was emulated as follows: for each session, after the disconnected subsets for the session were found, we performed timing optimization for each task subset separately and recorded the runtimes of the individual runs. The duration of that session was set to the longest run over its partitions. The optimization results for subsets were then merged and post-processed, before proceeding to the next session.

Results from the application of parallel timing optimization to our test circuits using Algorithm MSPart for multi-session partitioning are given in Table 2. For each design, the first row (Proc.#) gives the number of processors used in each session. In general the number of processors decreases over time, since the size of the input graph for each session monotonically decreases. The second row (Part. Time) gives the runtime for computing disconnected partitions for the session. The third row (Session Time) shows the duration of each session when parallelized. The runtime spent for the merging step is given in the fourth row (Merge Time). The last row (Serial Time) provides the runtime spent to do the tasks in each session serially, i.e., with a single processor. The last entries for rows 2 through 5 in each design give the total runtime for partitioning overhead, parallel processing, merging overhead, and serial processing, respectively. The partitioning time for Design 3 is two orders of magnitude longer than those for other designs. The reason is that the partitioning procedure could not find disconnected subsets larger than the specified size with the

Session		1	2	3	4	5	6	7	8	9	10	11	12	13	14	Total
Ckt. 1	Proc. #	2	1													
	Part.Time	2.3	1.2													3.5
	Session Time	523	1128													1651
	Merge Time	3	0													3
	Serial Time	1044	1128													2172
Ckt. 2	Proc. #	5	4	5	5	4	2	1								
	Part.Time	3.3	1.3	0.3	0.2	0.9	0.1	0.1								6.2
	Session Time	430	329	187	168	170	113	159								1566
	Merge Time	4	3	2	2	1	1	0								13
	Serial Time	2060	1306	886	597	669	222	159								5899
Ckt. 3	Proc. #	2	2	2	2	2	1									
	Part.Time	1050	322	261	156	160	125									2073
	Session Time	1939	710	581	516	366	3998									8110
	Merge Time	54	20	20	14	18	0									126
	Serial Time	3842	1217	1083	1009	674	3998									11823
Ckt. 4	Proc. #	6	6	6	6	6	3	4	3	2	3	4	3	2	1	
	Part.Time	1.7	6.3	1.2	0.8	0.7	5.5	0.6	1.8	0.3	0.1	0.1	0.1	0.1	0.1	19.1
	Session Time	245	283	134	136	38	205	104	143	191	81	129	218	165	575	2647
	Merge Time	11	8	6	5	5	4	4	3	2	2	3	2	2	0	57
	Serial Time	1346	1431	695	634	99	551	246	419	346	213	235	336	324	575	7450
Ckt. 5	Proc. #	8	8	8	8	6	6	5	4	5	5	5	2	1		
	Part.Time	3.2	2.1	1.6	0.7	0.6	0.4	1.3	1.9	2.4	2.8	3.4	2.8	3.4		26.8
	Session Time	437	338	180	246	120	210	29	107	86	57	70	51	254		2185
	Merge Time	11	7	5	7	2	2	2	2	1	2	1	2	0		44
	Serial Time	3345	2437	1309	1691	656	880	99	350	217	144	165	56	254		11603

Table 2. Partitioning/parallelization results.

regular subroutine, triggering a different (much slower) routine to search for the subsets satisfying the specification.

Table 3 summarizes the results from Table 2. Available Proc. is the maximum available number of processors given to the partitioning routine. This number was determined in proportion to design size. Parallel Runtime is the sum of overheads and runtime in parallel processing, i.e., task graph modeling time shown in Table 1, total of partitioning/merging times over sessions, and total of session durations over sessions, all shown in the last column of Table 2. Serial Runtime is the total of Serial Times over all sessions, taken from the last column of Table 2. Speedup is given by (Serial Runtime)/(Parallel Runtime), and Proc. Utilization is the ratio of speedup to the available number of processors.

As shown in Table 3, processor utilization ratios range from 23% to 75%. Design 3 gives the poorest results. This design has particularly heavy connections among tasks, resulting in long task graph generation and partitioning times and the lowest processor utilization. Another test circuit with high task graph connectivity is Design 1, which also yields relatively low speedups. Task graph connectivity may therefore be used to predict the speedup from the parallelization of its timing optimization. Moreover, to improve

Design	Available Proc. #	Parallel Runtime	Serial Runtime	Speedup	Proc. Utilization
1	2	1657	2172	1.31	66%
2	5	1575	5899	3.74	75%
3	5	10309	11823	1.15	23%
4	6	2723	7450	2.74	46%
5	8	2256	11603	5.14	64%

Table 3. Parallelization analysis.

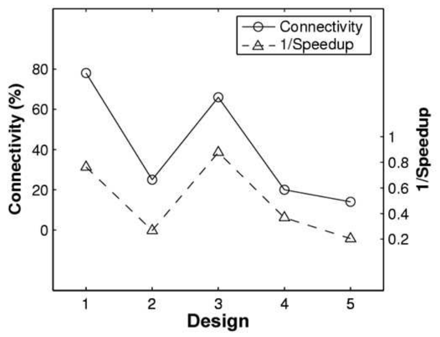


Figure 10. Task graph connectivity and 1/speedup.

processor utilization, the maximum number of available processors should be determined based not only on the size but also on the connectivity of the task graph. For example, if two processors were assigned to Design 3, reflecting its high connectivity, processor utilization would increase from 23% to 58%.

Design 5 is the largest design in our test suite with 130K cells. Total serial runtime for optimization is 11603 seconds. Its task graph has the lowest connectivity of 14%, and the parallelization achieves the highest speedup 5.14× by dynamically assigning 1–8 processors. Overhead due to task graph building, partitioning, and merging is 87 seconds total, which is 0.8% of serial runtime.

For each design in our test suite, Fig. 10 gives the connectivity of its task graph and the value of the metric 1/speedup (which equals the ratio of parallel runtime over serial runtime). Our data show a strong correlation of connectivity with this metric, leading to the conclusion that speedups increase as connectivity decreases. In general, connectivity tends to decrease as circuit size increases, since the physical span of a timing path is limited, and the portion of the circuit that overlaps with the path becomes smaller. Therefore, our multi-session partitioning for parallel timing optimization is expected to scale well, yielding increasingly higher speedups when applied to larger circuits.

7 Conclusion and Future Work

In this paper, we explore the problem of parallel post-placement timing optimization in VLSI design. To our knowledge, this is the first exploration of this topic. We first present a task graph representation for efficient parallelization of the path-based timing optimization process. We then describe a new multi-session partitioning scheme to improve the parallelization of heavily connected tasks by concentrating all necessary communication to take place only between sessions. In experiments with placed designs containing 20–130 thousand cells, our modeling/partitioning technique achieves speedups up to 5.14×, while dynamically utilizing 1–8 processors. The highest speedup is obtained for the design with the lowest connectivity. As connectivity tends to decrease with graph size, we expect our approach to scale well for increasingly larger circuits.

Future work includes the investigation of criteria for improving balance among sessions and metrics for increasing the accuracy of workload estimation for each path optimization task.

References

[1] C. Alpert, C. Chu, G. Gandham, M. Hrkic, J.Hu, C. Kashyap, and S.Quay. Simultaneous driver sizing and buffer insertion using a delay penalty estimation technique. In *Proc ACM Symp. on Physical Design*, pages 104–109, 2002.

[2] J. Chandy and P. Banerjee. A parallel circuit-partitioned algorithm for timing-driven standard cell placement. *Journal of Parallel and Distributed Computing*, 57(1):64–90, Apr. 1999.

[3] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP- Completeness*. Freeman and Co., New York, 1979.

[4] A. Guettaf and P. Bazargan-Sabet. Efficient partitioning method for distributed logic simulation of VLSI circuits. In *Proc IEEE Annual Simulation Symp.*, pages 196–201, 1998.

[5] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. Technical Report #98-019, University of Minnesota, 1998.

[6] S. Khanna, S. Gao, and K. Thulasiraman. Parallel hierarchical global routing for general cell layout. In *Proc IEEE Great Lakes Symp. on VLSI*, pages 212–215, 1995.

[7] F. Khundakjie, P. Madden, N. Abu-Ghazaleh, and M. Yildiz. Parallel standard cell placement on a cluster of workstations. In *Proc IEEE Intl. Conf. Cluster Computing*, pages 85–94, Aug. 2001.

[8] J. Lienig. Parallel genetic algorithm for performance-driven VLSI routing. *IEEE Trans. Evolutionary Computation*, 1(1):29–39, Apr. 1997.

[9] B. Ramkumar and P. Banerjee. ProperCAD: A portable object-oriented parallel environment for VLSI CAD. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 829–42, July 1994.

[10] E. Rudnick and J. Patel. Overcoming the serial logic simulation bottleneck in parallel fault simulation. In *Proc IEEE Intl. Conf. on VLSI Design*, pages 495–501, 1997.

[11] W. Shi and Z. Li. A fast algorithm for optimal buffer insertion. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 24(6):879–891, June 2005.

[12] W.-J. Sun and C. Sechen. A parallel standard cell placement algorithm. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(11):1342–57, Nov. 1997.