POLAR 3.0: An Ultrafast Global Placement Engine

Tao Lin
Iowa State University
tlin@iastate.edu

Chris Chu
Iowa State University
cnchu@iastate.edu

Gang Wu
Iowa State University
gangwu@iastate.edu

***Abstract*—Placement is one of the most important problems in electronic design automation. Although it has been investigated for several decades, a more efficient core engine is critically needed for the following reasons: (1) design scale becomes huge; (2) placement is typically run again and again to explore the design space at early design stages (e.g., physical synthesis); (3) placement core engine is called many times to iteratively optimize other objectives (e.g., timing and routability). In this paper, we propose a new ultrafast global placement engine called POLAR 3.0, which explores parallelism in state-of-the-art quadratic placer. POLAR 3.0 can make full use of multi-core system and it delivers 7-30× speedup over state-of-the-art academic placers by using a 8-core CPU, while the solution quality is competitive.**

I. Introduction

Placement is considered to be one of the most important problems in electronic design automation (EDA). Although it has been extensively studied for decades, it is still a very challenging problem and more efficient placers are critically needed. Firstly, the scale of placement is increasing continuously to tens of millions cells nowadays. Secondly, placement is used in early design stages (e.g., physical synthesis) to guide the design process, and it is typically run many times to explore the design space. Last but not least, multiple objectives, such as wirelength, timing and routability, should be optimized simultaneously, and the typical approach is to transform the problem into a sequence of wirelength-driven placement problems. Therefore, [1] indicates that placement is still a hot topic and [2] emphasizes the importance of developing a high performance placement core engine, which minimizes wirelength.

Problem scale has significant impact on the evolution of global placement core engine. In the early age, simulated annealing based placers (e.g., Timberwolf [3]) perform very well for small design. Then industry switches to min-cut based placement techniques (e.g., Capo [4] and Dragon [5]) for medium design. When design scale arrives at hundreds of thousands cells or even several millions, analytical placers [6–16] are considered the only effective method . However, when we are talking about huge design which might have tens of millions cells, the existing placers are still not fast enough considering modern design flow is iterative and placement should be performed many rounds.

To catch up with continuously increasing design scale, multi-threading has been widely used in EDA industry. However, it is challenging to achieve high parallelism for placement problem. For quadratic placer, such as SimPL [12] and POLAR [13], wirelength is minimized by quadratic programming (QP), which is solved as a sparse symmetric positive definite linear system by a preconditioned conjugate gradient (PCG) method. Wirelength minimization by QP dominates the total runtime of global placement stage. Since the x- and y-directed wirelengths are independent of each other, the two directions of wirelength optimization can be easily parallelized with two threads, one thread for each direction. However, PCG is known to be hard to parallelize [17] due to limited memory bandwidth and data dependency. Speedup does not scale well with more CPU cores.

For nonlinear placers [6, 8, 10, 15, 16], nonlinear programming consumes most of CPU runtime. In nonlinear placer, all the constraints are transformed into penalty functions. As a result, the cost function is not decomposable into two independent functions as in quadratic placers. Therefore, it is even more difficult to parallelize nonlinear placers.

In this paper, we systematically study the performance bottleneck of parallelism in quadratic placer. We propose an ultrafast global placement engine called POLAR 3.0. To achieve high scalability that previous works have not reached, the global placement iterations are divided into a series of *frames*, in which partitioning is applied based on cells' locations and then placement of each partition is performed simultaneously. To reduce loss of solution quality, *frames* are configured to allow varied partitioning, in order to prevent cells from being restricted in the same partition. Experimental results show that POLAR 3.0 can make full use of multi-core system and it delivers 7-30× speedup over state-of-the-art academic placers with competitive solution quality by using a 8-core CPU. The main contributions of this paper are concluded as follows.

- We systematically study the performance of state-of-the-art quadratic placers and point out that parallelizing global placement with high scalability is a very challenging rather than simple problem.
- We propose a new global placer to make full use of multi-core system. It is almost one order of magnitude faster than state-of-the-art academic placers, with competitive solution quality.

The rest of this paper is organized as follows. Section 2 is a preliminary to global placement. In Section 3, we point out the inherent challenges of parallelizing global placement with high scalability. In Section 4, we illustrate the ultrafast global placement engine POLAR 3.0. Experimental results are presented in Section 5. Finally, in the Section 6, we make conclusions.

II. Preliminary

In placement problem, a circuit can be represented by a hypergraph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is

the set of cells and $E = \{e_1, e_2, \ldots, e_{|E|}\}$ is the set of nets. Global placement tries to determine physical positions of cells without violating density constraints. We denote the x-coordinates of cells by a vector $\mathbf{x} = (x_1, x_2, \cdots, x_{|V|})$, and the y-coordinates by $\mathbf{y} = (y_1, y_2, \cdots, y_{|V|})$. The objective is the half perimeter wirelength (HPWL), which is measured by Formula (1).

$$\text{HPWL}(\mathbf{x}, \mathbf{y}) = \Sigma_{e \in E}[\max_{i \in e} x_i - \min_{i \in e} x_i + \max_{i \in e} y_i - \min_{i \in e} y_i] \quad (1)$$

A. Quadratic placement

In quadratic placer, a multi-pin net can be decomposed into a set of two-pin nets by bound2boubd (B2B) net model [9]. The Manhattan distance of two connected pins is approximated by their squared Euclidean distance, so the cost function ϕ of global placement can be defined in Formula (2).

$$\phi = \frac{1}{2}\mathbf{x}^T Q_x \mathbf{x} + \mathbf{c}_\mathbf{x}^T \mathbf{x} + \frac{1}{2}\mathbf{y}^T Q_y \mathbf{y} + \mathbf{c}_\mathbf{y}^T \mathbf{y} + const \quad (2)$$

where the connection matrices Q_x and Q_y are both sparse symmetric positive definite. Minimizing ϕ is equal to solving the linear systems (3) and (4).

$$Q_x \mathbf{x} = -\mathbf{c}_\mathbf{x} \quad (3)$$
$$Q_y \mathbf{y} = -\mathbf{c}_\mathbf{y} \quad (4)$$

To reduce cell overlapping, [18] proposed a simple way to add spreading force by pseudo net connecting cell's original position to its anchor (i.e., desirable location). Then the linear systems are updated and solved again.

Since quadratic placement formulation was first proposed, there are many improvements in academic quadratic placers. The most recent progress is a spreading approach called rough (look-ahead) legalization [12]. Many placers [12, 13, 19–21] based on rough legalization produce competitive results on the placement contests [22–26].

III. Challenge of Parallelization

Parallelizing state-of-the-art quadratic global placers, such as SimPL [12], to achieve high scalability is a very challenging rather than simple problem. The common global placement framework with rough legalization [12] is presented in Fig. 1. The three most time consuming components are respectively linear system generation by B2B model, solving linear system by PCG and rough legalization.

Firstly, linear system generation is difficult to parallelize. Without loss of generality, let us look into how to generate linear system (3) for x-direction. Compressed row storage (CRS) [17] is the most efficient format to store the non-zero elements of sparse matrix in our application. Since Q_x is symmetric, we only need to store its lower triangular part. For any non-zero element $Q_x[i][j]$ $(i \geq y)$, multiple nets may contribute to it. For example, suppose there are two-pin nets N_1 and N_2 connecting movable cells C_1 and C_2, N_1 and N_2 contribute a and b to the non-zero element $Q_x[2][1]$ respectively, then $Q_x[2][1] = a + b$. Therefore, nets have to be scanned one by one to generate all the non-zero elements [1]. Since linear systems (3) and (4) are independent of each other, two threads can are used for generating them, one is for (3) and the other is for (4). However, this is all where we can apply parallelism. The experiments show that SimPL [27] can only achieve 1.86× speedup in this step by using 8 threads. In practice, the speed cannot be more than 2× speedup.

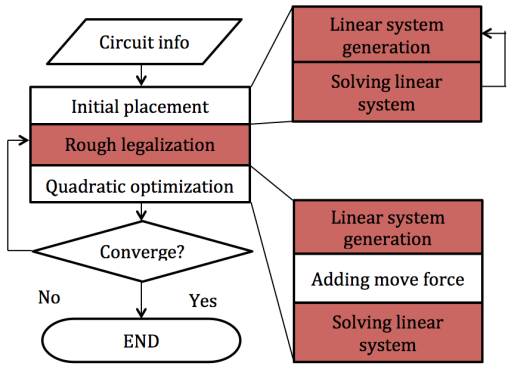


Fig. 1. The global placement framework with rough legalization. The three most time consuming components are highlighted.

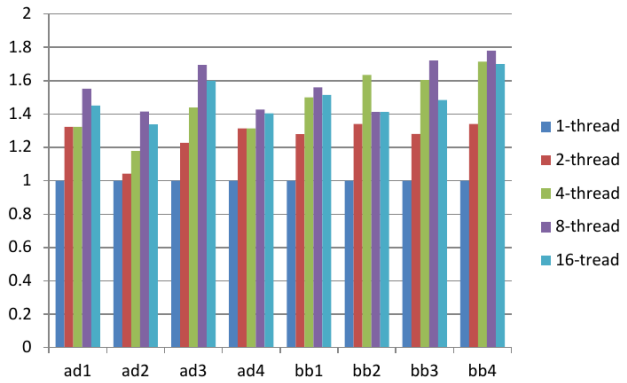


Fig. 2. Parallelizing Jacobi PCG solver in Intel MKL library in quadratic placement application. The y-axis is speedup.

Secondly, rough legalization is intrinsically sequential, since density hotspots are dealt with one by one. For each hotspot, it should get through two steps: expansion region search and cell spreading in expansion region. The runtime bottleneck is the second step, and the runtime of the first step can be ignored compared with that of the second step. Although parallelism is applied in the second step, [27] shows that SimPL only achieves 1.62× speedup in rough legalization by using eight threads. For POLAR [13], parallelizing the second step is even harder since it has already been highly optimized for runtime by lazy update technique. Table 1 gives the average runtime of rough legalization per global placement iteration for SimPL and POLAR.

Thirdly, PCG is widely used in science and engineering and there is still not effective algorithm to parallelize it so far [17]. To verify this, the performance of cutting edge PCG solver in

[1]Another method is to add lock for each non-zero element. Although nets can be scanned parallelly, the overhead of adding/releasing locks would overwhelm that benifet of parallelism and make the program extremely slow.

TABLE I. AVERAGE RUNTIME (S) OF ROUGH LEGALIZATION PER GLOBAL PLACEMENT ITERATION. THE EXPERIMENT WAS PERFORMED ON THE SAME MACHINE OVER ISPD2005 BENCHMARKS [22].

benchmark	SimPL	POLAR
adaptec1	0.52	0.12
adaptec2	0.61	0.16
adaptec3	1.87	0.31
adaptec4	1.57	0.36
bigblue1	1.79	0.21
bigblue2	1.65	0.28
bigblue3	3.79	1.01
bigblue4	9.87	1.72
Norm.	1.00	0.21

Intel MKL library [28] was measured by using a Intel Xeon E5-2640 v3 CPU, which has 8 cores. We used B2B net model to generate linear systems over ISPD2005 benchmarks [22], and then solved them by using different number of threads. The experimental results are presented in Fig. 2. It shows that the speedup is less than 2× and is increasing extremely slowly as the number of threads is increased. Note that, Running PCG solver with 16 threads is even a little slower than that with 8 threads in the experiment. That is because a core can launch 2 hyper-threads and these two hyper-threads share the same execution resources and are not truly parallel.

To further demonstrate the challenge by experiment, we implemented POLAR [13] and parallelized it in the following way using OpenMP: (1) parallelize every loop that can be parallelized; (2) all the computation that related to x-direction is parallel to that of y-direction. We set the number of threads to 1, 2, 4, 8 and 16 respectively, and ran POLAR over ISPD2005 benchmarks. Fig. 3 shows that we only get less than 1.8× speedup. Note that, launching more threads even slows down the program over some test cases. This is not abnormal for complex applications, for example, the ones whose bottleneck is memory bandwidth.

To sum up, existing methods [12, 13] has not reached to 2× speedup, and how to break through this limit while maintain good solution quality is the major challenge of this work.

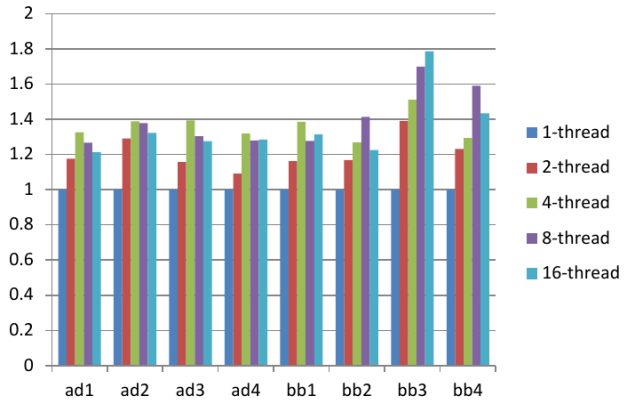


Fig. 3. Run POLAR by multi-threading. The y-axis is speedup.

IV. POLAR 3.0

To resolve the above challenge, we resort to an ancient while powerful strategy–divide and conquer. Divide and conquer used to be applied in global placement, such as partitioning based methods [4, 5, 7, 29–31]. However, since analytical techniques were proposed, partitioning based approaches are considered inferior. In POLAR 3.0, we show that partitioning is not outdated and can be leveraged to speed up analytical placer without sacrificing solution quality significantly.

We chose POLAR as our base engine and built up a new global placer on it. The basic idea is to divide global placement iterations into a series of *frames*, in which partitioning is applied based on cells' locations and placement of each partition can be performed independently by multi-threading.

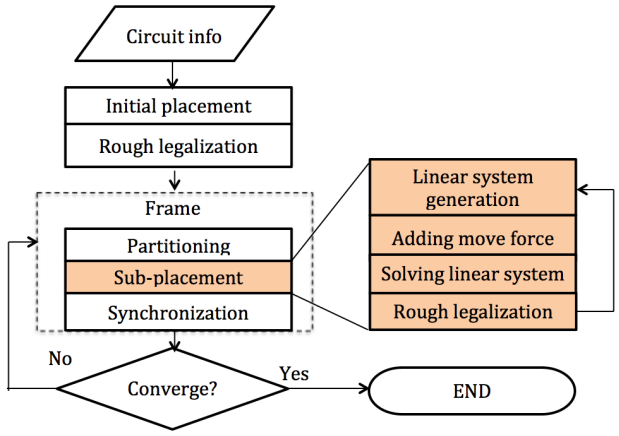


Fig. 4. The global placement framework of POLAR 3.0.

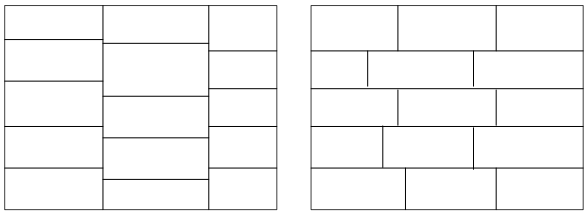
A. Framework

The framework of POLAR 3.0 is presented in Fig. 4. The first stage is initial placement, and then rough legalization. In the next stage, global placement iterations are divided into a series of *frames*. Each *frame* begins with a roughly legalized placement. Partitioning is applied based on cells' locations, and then placement of each partition (also called sub-placement in the rest of paper) is performed independently by multi-threading. Each sub-placement goes through several iterations of four processes: linear system generation, adding move force, solving linear system and rough legalization. Once all the sub-placements finish, POLAR 3.0 enters into synchronization, where the locations of all the cells are updated. POLAR 3.0 continues until the wirelength is not improved.

B. Placement-driven partitioning

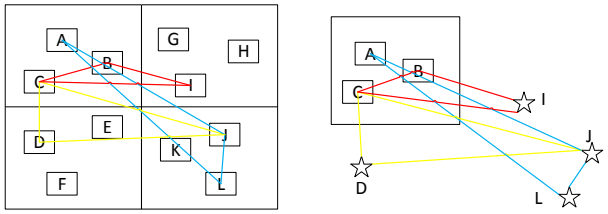
In each *frame*, POLAR 3.0 starts with a roughly legalized placement. Partitioning is performed based on cells' locations. Therefore, we call it placement-driven partitioning. Different from traditional partitioning based approaches, connection information (i.e., netlist) is never used. The goal of placement-driven partitioning is to divide the whole circuit into many small partitions with similar size, while traditional methods, such as hMETIS [32], try to minimize the number of connections among different partitions.

The whole circuit is divided into a set of partitions by horizontal and vertical cut lines. A partitioning scheme is represented by a tuple (xx, yy, dd), where $xx-1$ and $yy-1$ are respectively the number of horizontal and vertical cut lines. If dd is 0, POLAR 3.0 applies horizontal cut first and



(a) Scheme: $(5, 3, 1)$ (b) Scheme: $(5, 3, 0)$

Fig. 5. Two partitioning schemes.



(a) The whole circuit (b) Top-left partition

Fig. 6. Pins are considered fixated at their current locations if they are outside of partition.

then vertical cut. On the contrary, if dd is 1, vertical cut is applied before horizontal cut. Fig. 5 gives two instances of partitioning schemes, vertical cut is applied firstly in Fig. 5(a), while horizontal cut is applied firstly in Fig. 5(b).

Each partition is composed of a region, a set of cells and a set of nets, so it can be considered as a small instance of circuit. For each partition, if a net connects at least one cell inside of it, this net is kept in (belongs to) this partition. Besides, pins are considered fixated at their current locations if they are outside of partition. For example, as shown in Fig. 6, the whole circuit is divided into four partitions. The top-left partition contains three cells A, B and C. There are three nets associated with at least one of A, B and C. For the red net, it connects to cell I, which is outside of top-left partition, so cell I is considered fixated for the top-left partition. By above setting, placements of partitions are independent of each other, and each sub-placement is an placement instance. Note that, once all the sub-placements finish, all the cells' positions are updated before going to next *frame*.

To maximize parallelism, each partition is expected to have a similar number of cells. Otherwise, the placement of the partition which has the most number of cells would become the runtime bottleneck in all the sub-placements. Algorithm 1 gives the method of placement-driven partitioning. The placement region is split into a set of uniformed bins by a $m \times n$ grids. The number of cells in each bin is calculated, and then a lookup table can be built to quickly return the number of cells in any rectangular region which is composed of bins. When a rectangular region (e.g., the whole placement) is partitioned, the locations of cut lines can be easily computed by Algorithm 2[2].

[2]Without loss of generality, Algorithm 2 presents how to compute the locations of horizontal cuts. For vertical cuts, the method is similar.

Algorithm 1 Placement-driven partitioning

Require: Placement is split into a set of uniformed bins.
Require: A roughly legalized placement and a partitioning scheme (xx, yy, dd).
Require: Horizontal (or vertical) cut line should be exactly one of the lines that are used to split placement into bins.
Ensure: Each partition has similar number of cells.
1: Build up a lookup table to quickly return the number of cells in any rectangular region;
2: Apply horizontal (or vertical) cut first if $dd = 0$ or 1;
3: **for** any rectangular region produced in the last step **do**
4: Apply vertical (or horizontal) cut if $dd = 0$ or 1;
5: **end for**

Algorithm 2 Find the locations of cut lines

Require: A rectangular region represented by (l_x, l_y, r_x, r_y) in bin coordinate system, where (l_x, l_y) and (r_x, r_y) are respectively the coordinates of its left-bottom and right-up bins.
Require: : The number of horizontal cut lines, denoted by k.
Ensure: The locations of horizontal cut lines, stored in an array $p[1..k]$.
1: Initialize an array $n[l_y..r_y] = 0$;
2: **for** $i = l_y$ to r_y **do**
3: $n[i]$ = the number of cells in rectangular region (l_x, l_y, r_x, i);
4: **end for**
5: $m = \frac{n[r_y]}{k+1}$;
6: **for** $i = 1$ to k **do**
7: find the index l and u, that satisfy $n[l] \leq m*i \leq n[u]$;
8: **if** $m*i - n[l] < n[u] - m*i$ **then**
9: $p[i] = l$;
10: **else**
11: $p[i] = u$;
12: **end if**
13: **end for**

C. Varied partitioning scheme

If placement-driven partitioning is only applied once and then sub-placements are performed until the end of global placement stage, it would result in significant quality loss because each cell is restricted in the same partition and cannot be migrated from one partition to another, which leads to limitation of solution space. To resolve this issue, we introduce the concept of varied partitioning scheme.

A *frame* is composed of a partitioning scheme and several global placement iterations, so it can be represented as a tuple (xx, yy, dd, n), while (xx, yy, dd) is the scheme of placement-driven partitioning and n is the number of global placement iterations applied in the *frame*. To prevent cells from being restricted in the same partition, the *frame* configuration is varying from one frame to next. For example, if *frame* $(4, 4, 0, 5)$ is used first and then $(1, 1, 0, 1)$, it means that we apply a 4×4 placement-driven partitioning to get 16 partitions and perform 16 sub-placements in the next 5 global placement iterations, and then one flat global placement iteration is performed to allow cells to move outside of their partitions, in order to get better locations.

The design of *frame* configuration is important for runtime performance and solution quality. For example, If all the *frames* are configured to $(1, 1, 0, 1)$, then POLAR 3.0 is degenerated to POLAR [13] and cannot leverage multi-core system. However, if all the *frames* are configured to $(5, 5, 0, 1)$, then placement quality is significantly suffered. Therefore, the design of *frame* configuration is a trade-off between runtime performance and placement quality. In Section 5, we will further discuss about how to choose proper configuration of *frames* in the experiments.

D. Support partitioning efficiently

To make sub-placements run independently, a straight forward idea is to reconstruct the data structures (such as nets and cells) for each partition. It is the initial implementation of POLAR 3.0. However, this idea has several performance issues.

Firstly, memory usage is increased as the number of partitions is increased. For one thing, the memory to store a cell is doubled. One copy is for the whole circuit, and the other is for the partition to which this cell belongs. For another thing, which is even worse, a net (contains a set of pins) which crosses over k partitions has k copies, since each partition has one copy. Overall all, the whole memory usage is increased by several times. Huge memory footprint not only limits the scalability, but also slows down the program (e.g., results in higher cache miss rate). Besides, we observed that reconstructing netlist for each partition is relatively time consuming (about 20% of total runtime in the initial implementation).

To address the above issues, we redesigned the architecture of POLAR [13] in order to support partitioning efficiently. The key idea is to let partition share memory from the whole circuit. Therefore, there is only one copy for each cell/net/pin in the memory. By this new architecture, the runtime of partitioning is reduced to less than 1% of total runtime in the final implementation of POLAR 3.0, and can support partitioning as many as possible.

Fig. 7 presents the memory footprints of the whole circuit and the four partitions for the instance in Fig. 6. In the whole circuit, as shown in the top table of Fig. 7, we have a cell list, net list and pin list. Cells, nets and pins are stored in different kind of structures maintaining their own variety of information (e.g., width, height and location for cell, weight for net, offset (or cell id that it may have) for pin). Each net has a pair of values to store the start and end index of its pins in the pin list. In the partition, for each cell (or net), which belongs to this partition, only its id rather than its structure is stored. A hash table is used to map cell id in the whole circuit to a new id in the partition. This hash table has two functions: (1) check whether a cell belongs to this partition; (2) used in linear system generation, the new id of cell in its partition is exactly the same as its row index in the matrix Q_x (Q_y). Besides, the boundary of placement region should be stored for the whole circuit and partitions.

Scanning netlist on circuit (or partition) is the most fundamental operation in placement algorithm. It is the basic to implement a placer and widely used in many places, such as generating linear system and calculating total wirelength. Algorithm 3 shows how to scan netlist on partition. The hash table is used to check the condition in line 6.

Algorithm 3 Scan netlist on a partition.

```
1:  for any net id belongs to the partition do
2:      fetch the corresponding net structure;
3:      get the start and end index of its pins in the pin list,
    denoted by s and e;
4:      for i = s; i ≤ e; i = i + 1 do
5:          fetch the corresponding pin structure;
6:          if this pin belongs to some cell in the partition then
7:              //it is a movable pin;
8:          else
9:              //it is a fixed pin;
10:         end if
11:     end for
12: end for
```

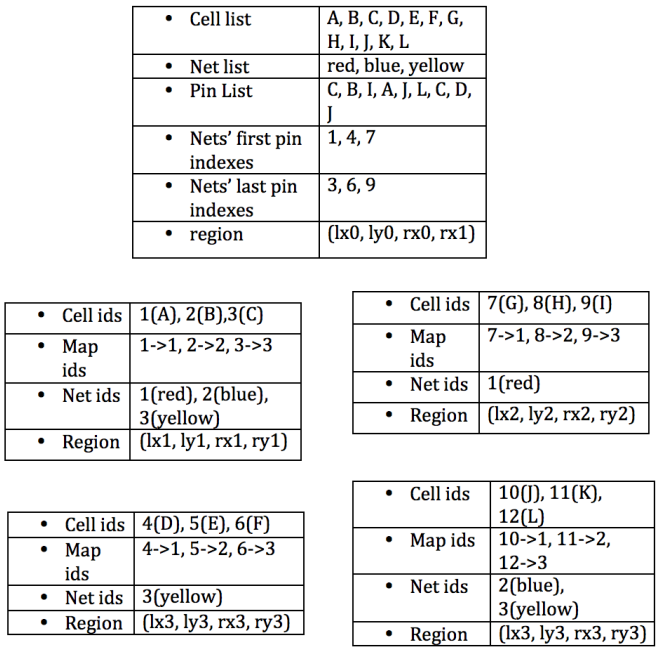


Fig. 7. Memory footprints for supporting partitioning efficiently. The placement instance is shown in Fig. 6.

V. Experimental Results

POLAR 3.0 is implemented in C++, and OpenMP is used to support parallelism. To verify the efficiency of POLAR 3.0, we obtained binaries of some state-of-the-art academic placers, such as FastPlace [11], NTUplace3 [16], ComPLx [19] and ePlace [10], and ran them over the same benchmarks on the same platform. Besides, all the detailed placement are performed by the detailed placer inside of NTUplace3.

The machine that was used in the experiments is a x86_64 Linux server, which has a Intel(R) Xeon(R) E5-2640 v3 CPU (with 8 cores, 2.6GHz frequency and 20M cache) and 132 GB RAM.

The benchmarks were downloaded from the official websites of some contests, including ISPD2005 [22], ISPD2006

[23], ISPD2011 [24], DAC2012 [25][3]. For those benchmarks from routability-driven placement contests [24, 25], routing information is ignored since all the above placers are wirelength-driven. The benchmarks from recent ISPD detailed routing-driven placement contests [33, 34] are not used in the experiments, because (1) all the above placers do not have an interface to read lef/def format files; (2) those large benchmarks are generated based on previous contests [24–26]. The target density is set to 1 for all the benchmarks.

In the experiments, we use the same *frame* configuration for all the benchmarks. As mentioned in Section 4.3, the design of *frame* configuration is to trade-off runtime and solution quality. If many partitions are used in the previous *frame*, then there are fewer partitions in the next frame to make up quality loss. Besides, flat placement is tried to avoid as much as possible, since it cannot make full use of multi-core system. We tried several *frame* configurations and picked a pretty good one for experiments. In this configuration, every four *frames* $(3, 5, 0, 5)$, $(1, 1, 0, 1)$, $(5, 3, 1, 5)$ and $(1, 1, 0, 1)$ are defined as a group and applied repeatedly. Each sub-placement would use at most two threads, for example, when generating (or solving) linear systems (3) and (4) simultaneously. Therefore, POLAR 3.0 launches at most 30 threads. Note that this configuration of *frames* does not necessarily give the best average results for all the benchmarks. Besides, the CPU used in the experiments only has 8 cores. Theoretically speaking, POLAR 3.0 could get more speedup by using a CPU which has more cores[4].

The experimental results are presented in Table 2. We reported the wirelength and runtime of both global and detailed placement for each placer. "G-WL" and "G-RT" represent the wirelength and runtime of global placement, while "D-WL" and "D-RT" represent the wirelength and runtime of detailed placement. FastPlace crashes on some benchmarks, which are annotated "crash". ePlace does not converge on some benchmarks, which are annotated "fail".

On average, compared with ePlace, POLAR 3.0 is about 4% worst on wirelength, while 31.8× faster. Compared with FastPlace, NTUplace3 and ComPLx, it is at least not worse or even better on solution quality, while significantly faster. The average wirelength ratio among FastPlace, NTUplace3, ComPLx and POLAR 3.0 is 1.07:1.04:1.00:1.00, and runtime ratio is 6.88:32.9:7.55:1.00.

For all these placers, the runtime of detailed placement is roughly comparable. To previous academic placers (not only FastPlace, NTUplace3, ComPLx and ePlace), global placement stage is more time consuming than detailed placement stage. Besides, global placement stage has more impact on solution quality compared with detailed placement stage. Therefore, much less works pay attention to detailed placement. However, for POLAR 3.0, on the contrary, detailed placement stage uses much more runtime than global placement stage. From runtime perspective, we hope that POLAR 3.0 would inspire more research on detailed placement. Besides, we believe that detailed placement stage is relatively easy to parallelize compared with global placement stage and will parallelize detailed placement stage in our future work.

[3]The benchmarks of ICCAD2012 contest [26] are the same as those of ISPD2011 and DAC2012 contest.

[4]We currently do not have 16-core or 32-core CPU to verify this, but will do it later.

A. POLAR 3.0 Runtime analysis

We analyzed the runtime of POLAR 3.0 by using different number of threads. The method is to change the number of partitions in *frame*. As shown in Table 3, for example, to measure the runtime of POLAR 3.0 using 8 threads, we can repeatedly apply *frame* group $(2, 2, 0, 5), (1, 1, 0, 1), (2, 2, 1, 5)$ and $(1, 1, 0, 1)$. POLAR 3.0 launches maximally 8 threads, since each partition would use 2 threads for generating (or solving) linear systems (3) and (4) simultaneously. The experimental results are shown in Table 4, where "WL" is the wirelength of detailed placement and "G-RT" is the CPU runtime of POLAR 3.0 in global placement stage. To measure the runtime of 1-thread and 2-thread, placement-driven partitioning is not applied. The difference between 1-thread and 2-thread is that generating (or solving) linear systems (3) and (4) are parallel (one thread for each direction) in 2-thread. Besides, we also added a group of *frames* called "speedy". We can use speedy to gain more speedup at the cost of sacrificing solution quality.

TABLE III. FRAME CONFIGURATION TO MEASURE RUNTIME OF POLAR 3.0 BY USING DIFFERENT THREADS.

# of threads	Configuration of *frame* group			
1-thread	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)
2-thread	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)	(1,1,0,1)
4-thread	(2,1,0,5)	(1,1,0,1)	(1,2,1,5)	(1,1,0,1)
8-thread	(2,2,0,5)	(1,1,0,1)	(2,2,1,5)	(1,1,0,1)
16-thread	(2,4,0,5)	(1,1,0,1)	(4,2,1,5)	(1,1,0,1)
speedy	(3,5,0,5)	(5,3,1,5)	(3,5,0,5)	(5,3,1,5)
default	(3,5,0,5)	(1,1,0,1)	(5,3,1,5)	(1,1,0,1)

By using 2/4/8/16/30 threads, POLAR 3.0 can achieve similar solution quality compared with POLAR, while gain 1.43/1.59/2.5/4.0× runtime speedup. Note that compared with 2-thread, 4-thread only gains little speedup. The main reason is that the number of nets kept in each partition is very close to that of whole circuit, so the runtime of generating (or solving) linear systems is also very closed to that of the whole placement.

VI. CONCLUSIONS

In this paper, we systematically study the problem of parallelizing state-of-the-art quadratic placer. The main challenge is that the major runtime components, such as solving quadratic problem, are difficult to parallel. Experiments show that existing method can only achieve less than 1.8× speedup by using 16 threads provided by a modern 8-core CPU.

To resolve this challenging problem, we built up a new global placer (POLAR 3.0) based on POLAR to fully leverage multi-core system. In POLAR 3.0, we propose placement-driven partitioning and verify partitioning scheme to trade-off runtime and solution quality. We demonstrate that by reasonable designing of *frames*, POLAR 3.0 could achieve competitive solution quality while reducing runtime significantly.

Since POLAR 3.0 runs much faster than existing detailed placement tools, detailed placement stage becomes the runtime bottleneck. We will try to parallelize existing detailed placement methods in our future work.

TABLE II. COMPARISON WITH THE STATE-OF-THE-ART ACADEMIC PLACERS. RUNTIME IS MEASURED IN SECONDS.

benchmark	test case	FastPlace				NTUplace3				ComPLx				ePlace				POLAR 3.0			
		G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT	G-WL	G-RT	D-WL	D-RT
ISPD05	adaptec1	80.6	73	79.2	44	81.5	217	80.3	33	80.2	92	78.1	40	73.1	190	75.6	82	78.6	12	78.9	34
	adaptec2	94.7	107	93.6	64	91.4	245	90.2	49	90.7	103	90.0	53	83.5	250	84.9	46	86.5	15	86.8	48
	adaptec3	214.5	209	215.5	112	239.8	570	233.8	105	206.0	261	206.2	96	193.9	906	196.5	82	206.1	26	207.2	92
	adaptec4	201.9	208	198.7	131	222.5	689	215.0	134	186.5	221	184.3	113	178.1	956	179.0	94	187.0	28	188.6	95
	bigblue1	100.7	103	97.6	70	96.2	432	98.7	49	96.2	175	94.3	51	89.6	298	91.0	41	95.0	16	95.2	48
	bigblue2	160.4	205	155.7	177	162.9	998	158.3	163	147.7	242	145.3	159	139.8	504	142.0	137	144.1	29	145.3	142
	bigblue3	371.8	487	373.4	329	351.3	1100	346.3	208	326.4	486	334.7	271	299.5	1432	308.2	207	324.2	64	329.6	340
	bigblue4	855.2	1024	840.6	663	852.2	3233	829.1	539	796.1	1379	788.9	542	738.6	3682	752.8	416	805.7	160	808.0	492
ISPD06	adaptec5	329.0	473	366.6	405	345.8	1238	344.6	177	324.0	428	322.4	211	294.8	1074	301.1	171	327.1	57	329.6	198
	newblue2	193.4	183	203.4	119	188.4	627	191.7	75	185.9	264	189.6	98	171.5	361	184.3	91	181.4	25	189.4	102
	newblue3	298.8	185	293.1	177	284.4	505	275.9	175	265.9	216	261.4	153	259.4	585	262.6	134	266.7	20	264.8	162
	newblue4	254.7	270	250.7	171	252.2	1036	247.6	149	237.1	328	232.9	138	216.1	882	222.9	114	238.2	38	239.0	127
	newblue5	424.7	616	472.4	481	429.1	2080	426.9	299	411.6	707	406.2	331	372.0	1578	383.1	257	410.4	88	415.6	289
	newblue6	516.1	595	506.3	376	505.0	1936	498.2	330	477.0	736	470.8	323	433.0	2136	443.1	255	478.1	88	479.6	291
	newblue7	1086.4	1063	1072	958	1114.4	3886	1100.2	525	998.1	1907	989.8	776	937.8	2612	956.9	648	984.6	187	990.8	683
ISPD11	superblue1	292.4	280	287.1	328	271.9	1620	267.9	180	259.5	261	256.5	248	269	6589	250.3	178	253.2	48	253.5	221
	superblue2	crash	crash	crash	crash	615.0	2778	609.7	211	605.9	323	608.2	382	fail	fail	fail	fail	592.8	52	592.5	273
	superblue4	crash	crash	crash	crash	217.8	757	216.5	121	214.4	187	212.3	152	fail	fail	fail	fail	209.0	26	209.3	132
	superblue5	358.6	242	355.1	237	352.9	1391	345.1	181	349.2	229	337.7	219	fail	fail	fail	fail	338.8	37	336.5	184
	superblue10	crash	crash	crash	crash	557.7	1356	551.2	215	538.3	377	535.6	246	520.9	1644	527.8	204	531.6	54	534.8	212
	superblue12	277.1	543	271.4	601	241.6	5804	238.7	333	242.9	642	237.8	479	206.5	3158	212.1	291	240.5	98	238.7	370
	superblue15	313.5	264	310.1	240	295.8	2124	297.3	155	297.2	385	295.0	191	280.9	892	283.9	175	292.2	51	294.8	188
	superblue18	157.0	190	152.2	169	139.1	1414	139.9	114	141.5	169	137.7	147	137.9	641	136.9	118	137.4	27	137.3	128
DAC12	superblue3	322.4	324	317.5	351	309.3	1493	302.9	219	314.1	267	304.8	282	fail	fail	fail	fail	299.0	45	300.1	228
	superblue6	crash	crash	crash	crash	318.6	1819	319.8	209	319.5	358	318.5	291	fail	fail	fail	fail	313.4	53	315.4	247
	superblue7	404.7	541	395.6	422	379.8	3467	377.3	294	388.6	530	385.0	395	366.1	1788	368.6	301	375.7	83	375.7	318
	superblue9	239.2	308	236.1	278	222.0	2285	221.8	204	219.7	305	217.2	233	fail	fail	fail	fail	214.6	46	215.2	219
	superblue11	crash	crash	crash	crash	335.9	1465	335.7	179	334.9	277	337.4	202	774.6	5399	391.5	623	338.5	48	337.4	199
	superblue14	236.4	197	234.7	202	224.2	1310	229.9	136	222.2	197	220.4	165	fail	fail	fail	fail	218.6	28	219.1	156
	superblue16	269.1	208	271.4	223	259.1	1662	262.8	120	253.9	192	256.7	190	fail	fail	fail	fail	260.4	37	258.7	144
	superblue19	154.1	204	155.8	216	143.9	1587	144.8	157	147.1	152	147.8	181	fail	fail	fail	fail	145.7	25	145.3	164
Norm.		1.08	6.88	1.07	1.37	1.05	32.9	1.04	0.95	1.01	7.55	1.00	1.15	1.01	31.8	0.96	1.05	1.00	1.00	1.00	1.00

TABLE IV. COMPARISON OF POLAR 3.0 WITH DIFFERENT NUMBER OF THREADS. RUNTIME IS MEASURED IN SECONDS.

benchmark	test case	1-thread		2-thread		4-thread		8-thread		16-thread		default		speedy	
		WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT	WL	G-RT
ISPD05	adaptec1	76.9	59	76.9	42	77.7	32	79	20	79.1	14	78.9	12	85.9	9
	adaptec2	86.5	72	86.5	51	87.5	39	86.2	24	86.8	17	86.8	15	91.1	10
	adaptec3	200.1	128	200.1	89	199.3	63	202.7	41	204.8	33	207.2	26	241.5	20
	adaptec4	181.6	129	181.6	88	184.9	64	185.4	40	188.4	30	188.6	28	201.2	18
	bigblue1	95.7	80	95.7	56	92.9	42	93.7	26	95.4	20	95.2	16	106.1	11
	bigblue2	143.3	133	143.3	93	143.9	73	144.9	44	145.6	32	145.3	29	147.5	18
	bigblue3	320.4	311	320.4	220	322.9	171	326.2	112	326.1	73	329.6	64	362.5	42
	bigblue4	787.7	704	787.7	508	795.3	426	800.5	260	811.4	188	808	160	835.4	108
ISPD06	adaptec5	313	258	313	181	320	154	319.6	98	324.6	65	329.6	57	363.2	40
	newblue2	189.2	123	189.2	87	188	65	187.8	37	187.1	29	189.4	25	193.1	16
	newblue3	262.7	93	262.7	63	263.3	51	264.3	32	265.5	25	264.8	20	267.6	15
	newblue4	231.7	184	231.7	132	228.8	104	234	61	233.4	46	239	38	256.9	28
	newblue5	407.7	407	407.7	298	404.5	225	406.9	149	410.6	107	415.6	88	432.6	64
	newblue6	473	399	473	282	469.9	229	471.6	147	477.3	105	479.6	88	498.7	60
	newblue7	986.8	841	986.8	603	982.1	472	978.6	300	980.4	223	990.8	187	1000.9	130
ISPD11	superblue1	251.3	208	251.3	144	249.3	114	251.5	74	251.5	49	253.5	48	258.7	30
	superblue2	585	215	585	149	586.1	119	587.6	77	586.6	56	592.5	52	593.8	36
	superblue4	207.5	114	207.5	79	205.6	58	207.9	42	208.3	31	209.3	26	214.8	17
	superblue5	333.1	160	333	111	334.2	84	333.4	55	335.7	39	336.5	37	340.9	24
	superblue10	534.7	220	534.7	149	532.6	132	533.6	87	534.1	61	534.8	54	540.1	38
	superblue12	225.5	484	225.5	332	229.4	230	232.4	154	235.8	114	238.7	98	258.7	68
	superblue15	294.7	210	294.7	145	290	134	291.1	82	295.8	61	294.8	51	311.7	38
	superblue18	135.9	119	135.9	84	135.2	66	135.1	40	135.4	32	137.3	27	162.3	20
DAC12	superblue3	296.6	207	296.6	145	292.4	124	297.4	77	299.8	57	300.1	45	308.9	32
	superblue6	311	229	311	160	311.1	142	313.2	88	312.2	62	315.4	53	320.6	39
	superblue7	372.3	352	372.3	248	373.4	226	377.8	139	386.3	97	375.7	83	392.2	59
	superblue9	210.6	207	210.6	147	213.9	123	221.1	78	221.6	55	215.2	46	236.6	32
	superblue11	337.9	207	337.9	143	336.4	123	335.5	76	337	50	337.4	48	349.9	33
	superblue14	219.1	135	219.1	94	219.9	83	218.9	49	218.2	35	219.1	28	222.9	21
	superblue16	255.3	164	255.3	118	255.1	96	257.2	60	257.7	43	258.7	37	271.1	26
	superblue19	145.1	120	145.1	83	145.2	63	145.7	43	147.2	31	145.3	25	150.9	17
Norm.		1.000	1.00	1.000	0.70	1.000	0.63	1.006	0.4	1.012	0.25	1.015	0.22	1.069	0.15

ACKNOWLEDGMENTS

This work is partially supported by NSF under grant CCF-1219100 and by Mentor Graphics. Besides, we would like to thank Dr. Ismail Bustany, Dr. Joseph Shinnerl and Ivan Kissiov from Mentor Graphics for the discussion of techniques.

REFERENCES

[1] C. Alpert, Z. Li, G.-J. Nam, C. N. Sze, N. Viswanathan, and S. I. Ward, "Placement: Hot or not?," ICCAD '12, pp. 283–290, 2012.

[2] I. L. Markov, J. Hu, and M.-C. Kim, "Progress and challenges in VLSI placement research," ICCAD '12, pp. 275–282, 2012.

[3] W.-J. Sun and C. Sechen, "Efficient and effective placement for very large circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 14, no. 3, pp. 349–359, 1995.

[4] J. A. Roy, D. A. Papa, S. N. Adya, H. H. Chan, A. N. Ng, J. F. Lu, and I. L. Markov, "Capo: robust and scalable open-source min-cut floorplacer," in *ISPD*, pp. 224–226, 2005.

[5] M. Wang, X. Yang, and M. Sarrafzadeh, "DRAGON2000: Standart-Cell placement tool for large industry curcuits," in *ICCAD*, pp. 260–263, 2000.

[6] A. B. Kahng, S. Reda, and Q. Wang, "APlace: A general analytic placement framework," ISPD '05, pp. 233–235, 2005.

[7] J. M. Kleinhans, G. Sigl, F. M. Johannes, and K. Antreich, "GORDIAN: VLSI placement by quadratic programming and slicing optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 10, no. 3, pp. 356–365, 1991.

[8] J. Cong, G. Luo, K. Tsota, and B. Xiao, "Optimizing routability in large-scale mixed-size placement," ASP-DAC '13, 2013.

[9] P. Spindler, U. Schlichtmann, and F. M. Johannes, "Kraftwerk2: A fast force-directed quadratic placement approach using an accurate net model," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 8, pp. 1398–1411, 2008.

[10] J. Lu, P. Chen, C.-C. Chang, L. Sha, D. J.-H. Huang, C.-C. Teng, and C.-K. Cheng, "ePlace: Electrostatics based placement using nesterov's method," DAC '14, pp. 121:1–121:6, 2014.

[11] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," ASP-DAC '07, pp. 135–140, 2007.

[12] M.-C. Kim, D.-J. Lee, and I. L. Markov, "SimPL: an effective placement algorithm," ICCAD '10, pp. 649–656, 2010.

[13] T. Lin, C. Chu, J. R. Shinnerl, I. Bustany, and I. Nedelchev, "POLAR: Placement based on novel rough legalization and refinement," ICCAD '13, 2013.

[14] U. Brenner, A. Hermann, N. Hoppmann, and P. Ochsendorf, "BonnPlace: A self-stabilizing placement framework," ISPD '15, pp. 9–16, 2015.

[15] W. Zhu, J. Chen, Z. Peng, and G. Fan, "Nonsmooth optimization method for VLSI global placement," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 34, no. 4, pp. 642–655, 2015.

[16] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 7, pp. 1228–1240, 2008.

[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*. 2003.

[18] B. Hu, Y. Zeng, and M. Marek-Sadowska, "mFAR: Fixed-points-addition-based VLSI placement algorithm," ISPD '05, pp. 239–241, 2005.

[19] M.-C. Kim and I. L. Markov, "ComPLx: A competitive primal-dual lagrange optimization for global placement," DAC '12, pp. 747–752, 2012.

[20] X. He, T. Huang, L. Xiao, H. Tian, G. Cui, and E. F. Y. Young, "Ripple: an effective routability-driven placer by iterative cell movement," ICCAD '11, pp. 74–79, 2011.

[21] T. Lin and C. Chu, "POLAR 2.0: An effective routability-driven placer," DAC '14, pp. 123:1–123:6, 2014.

[22] G.-J. Nam, C. J. Alpert, P. Villarrubia, B. Winter, and M. Yildiz, "The ISPD2005 placement contest and benchmark suite," ISPD '05, pp. 216–220, 2005.

[23] G.-J. Nam, "ISPD 2006 placement contest: Benchmark suite and results," ISPD '06, pp. 167–167, 2006.

[24] N. Viswanathan, C. J. Alpert, C. Sze, Z. Li, G.-J. Nam, and J. A. Roy, "The ISPD-2011 routability-driven placement contest and benchmark suite," ISPD '11, pp. 141–146, 2011.

[25] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei, "The DAC 2012 routability-driven placement contest and benchmark suite," DAC '12, pp. 774–782, 2012.

[26] N. Viswanathan, C. Alpert, C. Sze, Z. Li, and Y. Wei, "ICCAD-2012 CAD contest in design hierarchy aware routability-driven placement and benchmark suite," ICCAD '12, pp. 345–348, 2012.

[27] M.-C. Kim, D. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 50–60, 2012.

[28] Intel, "Intel Math Kernel Library." https://software.intel.com.

[29] M. Can Yildiz and P. H. Madden, "Improved cut sequences for partitioning based placement," DAC '01, pp. 776–779, 2001.

[30] A. Khatkhate, C. Li, A. R. Agnihotri, M. C. Yildiz, S. Ono, C.-K. Koh, and P. H. Madden, "Recursive bisection based mixed block placement," ISPD '04, pp. 84–89, 2004.

[31] Z.-W. Jiang, T.-C. Cheny, T.-C. Hsuy, H.-C. Chenz, and Y.-W. Changyz, "NTUplace2: A hybrid placer using partitioning and analytical techniques," ISPD '06, pp. 215–217, 2006.

[32] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Application in VLSI domain," in *DAC*, pp. 526–529, 1997.

[33] V. Yutsis, I. S. Bustany, D. Chinnery, J. R. Shinnerl, and W.-H. Liu, "ISPD 2014 benchmarks with sub-45nm technology rules for detailed-routing-driven placement," ISPD '14, pp. 161–168, 2014.

[34] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," ISPD '15, pp. 157–164, 2015.