Improving Min-cut Placement for VLSI Using Analytical Techniques

Saurabh N. Adya
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
sadya@eecs.umich.edu

Igor L. Markov
University of Michigan
EECS Department
Ann Arbor, MI 48109-2122
imarkov@eecs.umich.edu

Paul G. Villarrubia
IBM, Corporation
11501 Burnet Road
Austin, TX 78758
pgvillar@us.ibm.com

ABSTRACT

Over the past few decades, several powerful placement algorithms have been used successfully in performing VLSI placement. With the increasing complexity of the VLSI chips, there is no clear dominant placement paradigm today. This work attempts to explore hybrid algorithms for large-scale VLSI placement. Our work aims to evaluate existing placement algorithms, estimate the ease of their reuse, and identify their sensitivities and limitations. We study particular aspects of large-scale placement and particular types of netlists on which cause commonly known placement algorithms produce strikingly sub-optimal layouts. Our work points out that significant room for improvement remains in terms of robustness of placement algorithms. Indications are that combining multiple placement algorithms is a worthwhile approach to improve performance on multiple types of netlists.

In practice, it is important to have a robust and efficient linear-system solver for VLSI analytical placement. To this end, we describe the design of a parallel variant for linear systems derived from VLSI analytical placement problems.

1. INTRODUCTION

The performance of algorithms for large-scale ASIC placement has been gradually improving over the last 30 years. Algorithms based on balanced min-cut partitioning have been known at least as early as in the 1970s and reached the state-of-the-art again in the late 1990s after successful implementations of multi-level min-cut partitioners were reported at DAC '97 [4]. Placement algorithms based on analytical techniques (including force-directed) and those based on Simulated Annealing, have been widely studied and deployed since the 1980s. They, too, experienced a renaissance in 1990s. Currently, none of the three types can be called a clear winner and all are used, in one form or another, in commercial tools and academic implementations. A number of authors explored hybrid algorithms, mostly based on partitioning, including min-cut, and analytical techniques [24, 17, 22, 25] as well as partitioning and simulated annealing [27]. The former type of hybrid algorithms are popular in the context of timing-driven placement [20, 14] and whitespace management [2] whereas the latter are more successful in congestion-driven placement [27, 28].

In a recent effort to evaluate the remaining room for improvement, researchers from UCLA created artificial placement benchmarks with known optimal wirelength [9]. State-of-the-art academic placers produce very sub-optimal solutions on those benchmarks, which leads to the conclusion that significant room for improvement remains. In this work, we show that extending a state-of-the-art academic min-cut placer [7] with analytical techniques particularly improve placement performance on benchmarks from [9]. This is somewhat consistent with results in [2], where a similar min-cut placer was extended using analytical optimization.

We observe that while there are several straightforward avenues to parallelizing min-cut placers (some of which are leveraged by major commercial tools), parallelizing analytical placers appears less obvious. We therefore explore data structures for such parallelization and relevant implementation trade-offs.

2. REGULAR STRUCTURES

Generic standard-cell placers are known to perform badly on data-path style designs. EDA vendors provide special placers for regular data-path style design, e.g. Mustang offered by Cadence. We studied the performance of several placers on regular grid structures because datapaths often behave like grids, but grids are much easier to visualize. We would like to know which placement techniques are successful in handling mixed control-path and data-path designs effectively.

We created artificial designs with cells of regular height and width. The cells were then connected with two-pin nets in a regular grid structure. Figure 1 shows one such design with 100 movables cells arranged in a 10 X 10 grid. There are 4 terminals connected to the 4 corner cells as shown, to anchor the design. We created five such designs with varying numbers of nodes and whitespace. These designs were run through four different placers and the results are summarized in Table 1 and Figure 2. Dragon [27] which combines recursive partitioning with annealing doesn't do favorably on these purely regular designs. Capo [7] which is a top-down min-cut recursive bisection placer does much better for designs with some amount of whitespace. However, for completely full designs, with 0% whitespace, Capo returns poor placements. Our work tunes Capo to handle such designs much better.

During each partitioning step with a vertical cut line, the Capo 8.5 with default parameters would use a fairly large tolerance (of the order of 10-20%) in order to find better cuts. After a good cut is found, the geometric cut line is adjusted according to the sizes of partitions, with an equal distribution of whitespace among the partitions. However, *if no whitespace is available in the block*, this technique can cause cell overlaps. Namely, since no "jagged" vertical cutlines are allowed, the set of partition balances that can be realized with a vertical cutline and no whitespace is fairly discrete.

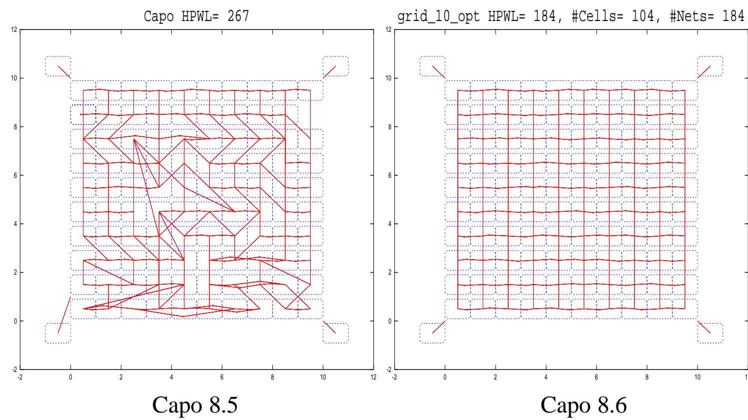


Figure 1: Capo placements for designs with regular grid connectivity. Capo 8.0 produces sub-optimal placements. Capo 8.5 with repartitioning with small tolerances, produces the optimal placement for this design. There are 4 terminals connected to the 4 corner cells to anchor the design.

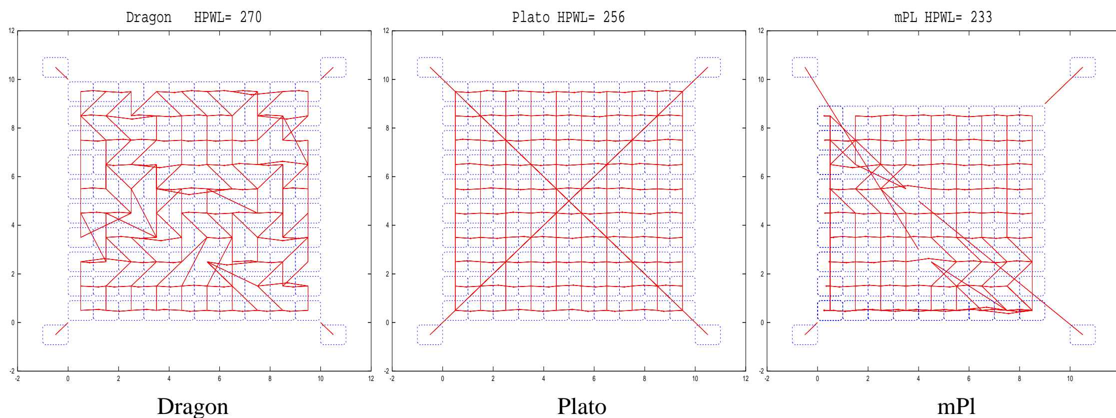


Figure 2: Placements of grid circuit 10x10 produced by different academic placers. No placer could achieve the optimal wirelength. mPl produces placements with a lot of overlaps. Plato/Kraftwerk seems to work best but the orientation of the placement is not maintained.

Capo 8.5 simply rounds the current balance to the closest realizable and establishes the geometric cutline accordingly. When whitespace is scarce, one of the resulting partitions may be overfull and the other may have artificially created whitespace. Needless to say, relatively few cell overlaps can be created this way, and Capo8.5 typically removes overlaps by a simply and very fast greedy heuristic. However, this heuristic increases wirelength.

In an attempt to circumvent this effect, we revised the partitioning process in Capo. When a placement block is partitioned with a vertical cutline, at first the tolerance is fairly large. As described previously, this allows Capo to determine the location of the geometric cutline. Furthermore, if the block has very little whitespace, we then repartition it with a small tolerance in an attempt to rebalance the current partitions with respect to the newly defined geometric cutline.

Another modification we implemented is related to terminal propagation. Normally, if a projection of a terminal's location is too close to the expected cutline, the terminal is ignored by Capo in an attempt to avoid excessively speculative decisions. The proximity threshold is defined in percent of the current block size, and we call this parameter "partition fuzziness". For example, suppose that the y location of a terminal is within 9% of the tentative location of the horizontal cutline. Then, with partition fuzziness of 10%, this terminal will be ignored during partitioning. Our studies of Capo performance on grids suggested that partition fuzziness should be tuned up, particularly for small blocks. For example, if a placement block has only three cell rows, then possible tentative locations of horizontal cutlines are relatively far from the center. In a neighboring block that has not been partitioned yet, all cells are "located" at the center of the block, causing all connected terminals to propagate into one partition in the current block. To avoid this, we increased partition fuzziness to 33%.

The two changes described above improve the performance of Capo on the grid designs with 0% whitespace by a factor of two.

Out of all the placers that we tried, Plato/Kraftwerk [13] seems to produce the best placements for these regular structures. This is to be expected because Plato is an analytical placer which attempts to minimize a well-defined measure of wirelength. Such a strategy

seems to be cognizant of the global structure of the design netlist. Yet, in our experiments, Plato does not handle fixed terminal connections well. As can be seen from figure 2, the placement produced by Plato for the grid circuit 10x10 seems flipped from the optimal both vertically and horizontally, or, perhaps, rotated. This results in terminal connections being unnecessarily long and can ruin path timing. In separate experiments we found out that Plato is significantly inferior to Capo on industrial random-logic netlists, when results are measured by wirelength. This further motivates our interest in hybrid placers that combine min-cut and analytical placement techniques.

3. HYBRID PLACERS

Other authors already proposed to combine multiple placement techniques [24, 17, 22]. However, "pure" placers have developed more rapidly and currently represent state of the art. We believe that this leaves an opportunity for hybrid placers that have not kept up with pure placers. In particular, it has been shown that the min-cut placer Cap performs poorly on placement benchmarks with known optimal wirelength (PEKO) [9] that have been recently added to the GSRC Bookshelf [6]. Capo produces placements with wirelength up to two times the optimal wirelength. The Dragon placer that uses min-cut and simulated annealing produces even poorer placements. However, the experimental multi-level analytical placer mPl 1.2 from UCLA is typically within 50% of optimum. mPl 1.2 completes roughly as quickly as Capo on all benchmarks, but performs poorly on the more realistic IBM benchmarks. The authors of PEKO benchmarks admit that their benchmarks may not be representative of real-world VLSI circuits, and mostly use them to show a significant gap between optimal and achievable placements.

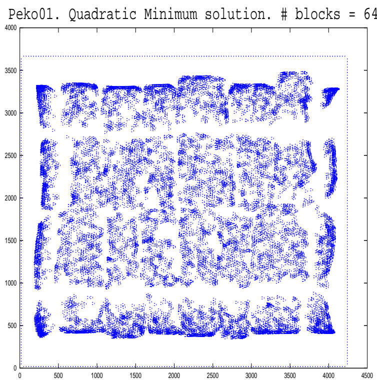


Figure 3: Quadratic minimum solution for benchmark PEKO01 after 6 partitioning steps. The design has 64 blocks. Using the quadratic minimum locations of cells during terminal propagation for partitioning helps achieve better HPWL for PEKO benchmarks.

Analytical Terminal Propagation. Terminal propagation [11] is essential to the success of a top-down min-cut partitioning based placement approach. During placement, when a particular placement block is split into multiple subregions, some of the cells inside may be tightly connected to external cells ("terminal"). Ignoring such connections allows a bigger discrepancy between good partition solutions and solutions that result in better placements. However in classic partitioning formulations, external terminals are irrelevant as they cannot be freely assigned to partitions, since they are fixed. One solution to this problem is by using an extended formulation of "partitioning with fixed terminals", where terminals are considered fixed in ("propagated to") one or more partitions. Terminal propagation has been described in [11, 8, 23] and is typically driven by the geometric proximity of terminals to partitions. During top-down placement, there is no proper information about the placement of cells assigned to a particular partition. Most placers assume some ad hoc techniques to assign location to these cells, like the center of the partition or assign random legal locations to all the cells. This may affect the terminal propagation and hence the final placement quality.

In our experiments, we were able to significantly improve the performance of Capo on those benchmarks (see Table 2) by combining Capo with a simple SOR-based analytical placer along the lines of [25]. This placer is called after every round of min-cut partitioning in Capo and finds locations of cells that minimize a quadratic wirelength objective, subject to region constraints (a cell must be within the top-down block where it is assigned by recursive partitioning) and center-of-gravity constraints (weighted locations of cells in a given block must average to the geometric center of the block). The only way in which this addition affects the result of Capo placement is terminal propagation. Normally, terminal propagation assumes that cells are placed in the centers of their regions. Our modified version of Capo used the locations produced by the analytical placer for terminal propagation. A sample analytical placement is shown in Figure 3. As the results in Table 2 indicate, this hybrid placer achieves much better wirelengths than Capo.

4. PARALLEL LINEAR SOLVERS

Solving large linear system of equations fast is essential for a analytical placer. Successive Over Relaxation (SOR) is an iterative solution method for solving linear systems of equations of the form $AX = B$. We parallelized the SOR engine in context of solving large linear systems derived from VLSI physical design problems. For linear systems of equations derived from VLSI physical design instance, matrices are fairly sparse and intelligent ways of partitioning the problem are devised so as to minimize communication between processors. We demonstrate a near-linear speedup for SOR in such a context.

Direct methods for solving linear system of equations such as LU decomposition [18] or Gaussian elimination techniques [18] become very inefficient as problem size increases. They also require large storage space to perform the intermediate steps. Therefore robust and fast iterative methods such as the Krylov subspace methods, Conjugate Gradient, Successive Over Relaxation etc become highly useful. Iterative solvers require less memory and can yield an approximate solution significantly faster than direct methods. We implemented a relatively simple iterative solver based on the SOR technique and parallelized the SOR engine.

4.1 Objective Function

In Analytical Placement the algorithm tries to minimize a certain objective function while trying to make sure that locations of objects are legal and there are no overlaps. Minimizing the total quadratic wire-length of the design is a popular objective.

A VLSI design net-list can be characterized by N nodes and V nets. Each net is an hyperedge connecting two or more nodes. A sample netlist is shown in figure 4. The quadratic wirelength objective function can be derived as follows. The length L_v of a net v is measured by the sum of the squared distances from its pins to the nets center coordinates (x_v, y_v).

$$L_v = \sum_{u \in v} [(x_u - x_v)^2 + (y_u - y_v)^2]$$

Circuit	#Nodes	#Nets	WS %	Optimal HPWL	Dragon HPWL	Plato HPWL	Capo Default HPWL	Capo + repart HPWL
10x10	100	184	0	184	293	202	267	184
95x95	9025	17864	5	17884	39687	18302	21828	22764
100x100	10000	19804	0	19804	46066	20519	38352	21314
190x190	36100	71824	5	71864	175623	75384	90665	89814
200x200	40000	79604	0	79604	198182	82335	193167	100041

Table 1: Wirelength achieved by several placers on regular grids of varying size and with varying whitespace.

Circuit	#Nodes	OPT HPWL	Capo 8.5(Default)			Capo 8.5(+ATP)		
			HPWL	WL/OptWL	Time(sec)	HPWL	WL/OptWL	Time(sec)
PEKO01	12506	0.814	1.48	1.81	42	1.29	1.58	74
PEKO02	19342	1.26	2.37	1.88	78	2.03	1.61	177
PEKO03	22853	1.50	2.78	1.85	101	2.66	1.77	159
PEKO04	27220	1.75	3.08	1.81	121	3.12	1.78	197
PEKO05	28146	1.91	3.59	1.85	133	3.16	1.65	260
PEKO06	32332	2.06	4.0	1.94	152	3.57	1.73	258
PEKO07	45639	2.88	5.6	1.94	220	5.07	1.76	374
PEKO08	51023	3.14	5.79	1.84	257	5.57	1.77	574
PEKO09	53110	3.64	7.22	1.98	276	6.47	1.77	489
PEKO10	68685	4.73	9.22	1.94	407	8.0	1.69	726
PEKO11	70152	4.71	9.25	1.96	411	7.8	1.65	634
PEKO12	70439	5.00	9.4	1.88	413	8.3	1.76	804
PEKO13	83709	5.87	11.3	1.92	528	10.42	1.77	860
PEKO14	147088	9.01	17.7	1.96	1052	15.86	1.76	1627
PEKO15	161187	11.50	23.4	2.03	1383	20.54	1.78	2274
PEKO16	182980	12.50	24.54	1.96	1614	21.6	1.72	2544
PEKO17	184752	13.40	27.0	2.01	1820	24.27	1.81	2795
PEKO18	210341	13.20	26.55	2.01	2080	23.66	1.79	2868

Table 2: Effect of analytical terminal propagation (ATP) on placement of PEKO benchmarks. In the default mode, during global placement all the cells are assigned to the center of the blocks they belong to at each level of partitioning. For analytical terminal propagation, at each level of partitioning, a global quadratic minimum solution is computed. This solution satisfies the block boundary constraints and center of gravity constraints. This illegal locations of cells is used for terminal propagation. This effects the placement quality significantly.

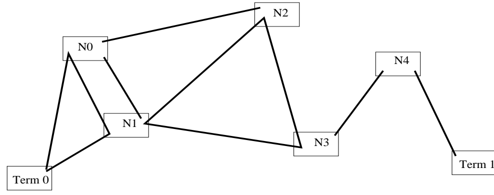


Figure 4: Sample hypergraph/netlist. Term0 and Term1 are terminals or pads which are fixed. N0, N1, N2, N3 and N4 are movable nodes connected by hyperedges.

Let w_v be the net-weight for net v. A high net-weight indicates the criticality of the net and groups the connected modules closer together. The objective function can then be expressed as the sum of quadratic wirelength of all nets.

$$\Phi = \frac{1}{2}\sum_{v \in V} L_v.w_v$$

To reduce the number of variables the coordinates of the net center are modeled as the mean values of the coordinates of the pin. This is equivalent to replacing each net by all two-point connections of its pins (a clique). The edges of the clique are given a weight $e = 2/p$, where p is the degree of the net. The objective function which now depends only on the module coordinates can be written in matrix form as follows.

$$\Phi(x,y) = \frac{1}{2}x^T Ax + B^T x + \frac{1}{2}y^T Cy + D^T y$$

The vectors x and y represent the coordinates of the movables nodes. Since the above equation is separable into $\Phi(x,y) = \Phi(x) + \Phi(y)$, we can write the objective function in the x component as follows.

$$\Phi(x) = \frac{1}{2}x^T Ax + B^T x$$

Matrix A represents the connectivity between the objects and the vector B represents the fixed components like terminals of the design. $\Phi(x)$ is convex function and has a unique global minimum. The minimum can be found by solving a linear system of the form $Ax = B$. Figure 5 shows the quadratic minimum solution for design ibm05.

For netlists with millions of movable objects, solving the linear system of equations $Ax = B$ fast is critical to the performance of placement algorithms. Usually iterative solvers like Conjugate Gradient or Successive Over Relaxation (SOR) [18] are employed in this context. In particular, SOR can be implemented directly on the netlist without creating explicit matrices. In our work we seek to partition the netlist among processors such that the SOR iterations are efficiently parallelized.

4.2 Serial SOR

Let $Ax = B$ be the linear system of equation to solve. An iterative solver starts from an initial guess to the solution and gradually refines it till it reaches the solution. Let x be the exact solution of the $Ax = B$. Let x' be an inaccurate (or estimated) solution vector, such that $x = x' + \delta x$. Inserting this into the given equation we find $A\delta x = B - Ax'$. We can represent this as an iterative formula $A(x_{k+1} - x_k) = B - Ax_k$. Thus new values of iterates can be found

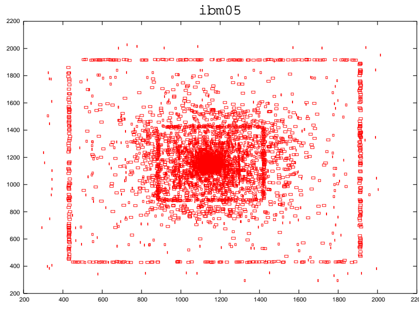


Figure 5: Quadratic minimum solution for design ibm05.

from previous values. This process is continued till it converges to the solution. In the Jacobi relaxation scheme the new value of the iterate is calculated as below.

$$a_{ii}x_i^{(k+1)} = b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}; i = 1, \ldots, N$$

Thus during each iteration the previous estimated value of the variables is used. Gauss-Seidel Relaxation has a faster convergence rate than Jacobi. In Gauss-Seidel Relaxation the new value of the iterate is calculated as follows.

$$a_{ii}x_i^{(k+1)} = b_i - \sum_{j>i} a_{ij}x_j^{(k)} - \sum_{j<i} a_{ij}x_j^{(k+1)}; i = 1, \ldots, N$$

Thus some of the right hand terms refer to the present iteration $k+1$ instead of the previous, k; since, each term is available as soon as it is calculated. To further speedup the convergence rate SOR is employed. The formula for calculating the next iterate in SOR is as follows.

$$x_{k+1}^{SOR} = \omega x_{k+1}^{GSR} + (1-\omega)x_k$$

x_{k+1}^{GSR} is the value of the iterate computed by Gauss-Seidel Relaxation. The "relaxation parameter" ω may be varied within the range $0 \leq \omega \leq 2$ for convergence.

The matrix A of the linear system of equation $Ax = B$ generated during quadratic minimization of wirelength in VLSI placement is very sparse. A sparse matrix is a matrix that has relatively few nonzero values. Most sparse matrix software takes advantage of this "sparseness" to reduce the amount of storage and arithmetic required by keeping track of only the nonzero entries in the matrix. Thus in the placement context SOR operates directly on the netlist hypergraph without explicitly forming the system of equations $Ax = B$. The pseudo code for the SOR loop to find the quadratic minimum solution is shown in figure 6.

To parallelize the SOR we first need to partition the data efficiently between the processors. We did this using the ParMetis package [21]. We explain in brief the ParMetis package in the following section.

4.3 PAR Metis

ParMetis is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs. The algorithms in ParMetis are based on multi-level partitioning that are implemented in the widely-used serial package Metis [19]. However, ParMetis extends the functionality provided by Metis and included routines that are specially suited for parallel computations and large-scale numerical simulations.

```
1    SER_SOR(N,V) /*N nodes, V nets*/
2     while(converged)
3     begin
4      for i = 1 to N
5      begin
6        newLocX[i] = getOptXLoc(i);
7        newLocY[i] = getOptYLoc(i);
8        xloc[i] = newLocX[i]ω + xloc[i](1 − ω);
9        yloc[i] = newLocY[i]ω + yloc[i](1 − ω);
10     end
11    end
```

Figure 6: Pseudo code for serial version of SOR operating on a VLSI netlist (N,V). Procedure $getOptXLoc(i)$ returns the next estimated value of the x-location of node i. It does this by going over all the connections of the node i and returning the mean value of the x-locations. $getOptYLoc(i)$ does the same computation in the y-direction. By operating directly on the graph, sparsity of the matrix is utilized by just going over non-zero values.

However ParMetis operates only on regular graphs. VLSI netlist are expressed as hypergraphs. Each hyperedge in a hypergraph can have 2 or more than 2 connections. We decided to model the hyperedges as cliques in which all connected nodes are connected to each other by a 2-pin net. An example of such a decomposition is shown in figure 7. All of the graph routines in ParMetis take as input the adjacency structure of the graph, the weights of the vertices and edges, and an array describing how the graph is partitioned among the processors. We had to write a converter from the internal format used by our placement/floorplanning tool to the internal format used by ParMetis to be able to use ParMetis to partition the original netlist among the processors. hMetis [16] is a serial partitioner which works directly on hypergraphs. We decided to use ParMetis for partitioning in favor of hMetis since we wanted minimal time overhead when converting from serial to the parallel version for the SOR solver. The reason for using a min-cut partitioner to partition the data among the processors was to minimize the communication required between different processors during an SOR iteration. During an SOR iteration only connected components need to communicate their present locations and using a min-cut partitioner to partition the data seems to be the ideal choice for this application. From our experiments we conclude that ParMetis is highly optimized for time and produces high quality partitions.

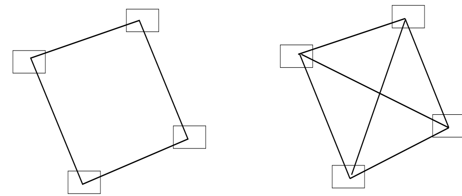


Figure 7: Modeling of Hyperedges as cliques.

4.4 Parallel SOR

In this section we describe the data-structures used for the parallel version of the SOR engine. We also describe the communication involved between different processors during an SOR iteration.

For scalability reasons, we decided to implement the parallel SOR engine for a distributed computing environment using the MPI library. On a shared memory machine, it is trivial to implement the

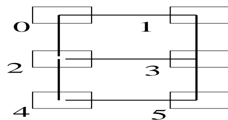
parallel version of the SOR engine after the data has been properly partitioned. However, for a distributed memory machine we need to build special data structures to represent the distributed netlist. We represent the structure of the graph by the Compressed Storage Format (CSR) extended for parallel distributed-memory computing [21]. We first describe the serial CSR format and then extend it for storing graphs that are distributed among processors.

4.5 Serial CSR Format

The CSR format is a widely used scheme for storing sparse graphs. Here the adjacency structure of the graph is represented by two arrays $xadj$ and $adjncy$. A graph with n vertices and m edges can be represented by using arrays of sizes $xadj[n+1], adjncy[2m]$. Array $adjncy$ is of size $2m$ because every edge is listed twice. The adjacency structure is stored as follows. The adjacency list of vertex i is stored in array $adjncy$ starting at index $xadj[i]$ and ending at (but not including) $xadj[i+1]$. The array $xadj$ is used to point where the adjacency list for each specific vertex begins and ends. Figure 8 shows a sample graph in (a) and its serial CSR formal in (b).

4.6 Distributed CSR Format

This is an extension of the serial CSR format that allows the vertices of the graph and their adjacency lists to be distributed among processors. We assume that each processor P_i stores n_i consecutive vertices of the graph and the corresponding m_i edges. Here, each processor stores its local part of the graph in the arrays $xadj[n_i+1]$ and $adjncy[m_i]$, using the CSR storage scheme. In addition to these two arrays, each processor also requires the array $vtxdist[P+1]$ that indicates the range of vertices that are local to each processor. In particular processor P_i stores the vertices from $vtxdist[i]$ up to (but not including) vertex $vtxdist[i+1]$ Figure 8 (c) shows the distributed CSR format for sample graph (a) distributed among 3 processors. Each processor also holds other data such as the weight of each node and net, the locations of internal nodes and external nodes, information whether a node is a terminal (fixed) or not, etc. The local data for each processor is shown in figure 9.



(a) Sample Graph

xadj : 0 2 4 6 8 10 12
adjncy : 1 2 0 3 0 3 1 3 2 5 3 4

(b) Serial CSR

Processor 0:	xadj :	0 2 4
	adjncy :	1 2 0 3
	vtxdist :	0 2 4 6
Processor 1:	xadj :	0 2 4
	adjncy :	0 3 1 3
	vtxdist :	0 2 4 6
Processor 2:	xadj :	0 2 4
	adjncy :	2 5 3 4
	vtxdist :	0 2 4 6

(c) Distributed CSR

Figure 8: Sample Graph and corresponding serial CSR format and distributed CSR format

```
int * xadj;        xadj array for processor
int * adjncy;      adjncy array for processor
int * vtxdist;     vtxdist for all processors
float * adjwgt;    weight for local edges
int * extNodes;    external nodes connected to local nodes
float * xlocs;     x-locations for local+external nodes
float * ylocs;     y-locations for local+external nodes
bool * isTerm;     which local nodes are terminals(fixed)
```

Figure 9: Local Data for SOR iterations on each processor

4.7 Distributed SOR scheme

As explained in section 4.2 SOR performs iterations untill convergence. At each iteration the solver traverses all nodes and changes the location of every node to a locally-optimal location. In the distributed parallel version we employ the Jacobi scheme (explained in section 4.2) to update the locations of each node. Processor 0 acts as the master processor responsible for synchronization. Its reads in the netlist and converts to ParMetis format. Then ParMetis is used to partition the data among remaining processors. The netlist is reordered according to the results of the partitioner. The master processor converts this reordered netlist into the distributed CSR format and distributes the required parts to remaining processors.

Each processor knows which nodes are external for it, i.e., not on the processor but are adjacent to nodes assigned to the processor. At the start of each iteration the master processor sends the current locations of external nodes for each processor to the corresponding processor. The individual processors perform SOR iterations based upon the current locations of the local nodes and external nodes. At the end of the iterations the processors send locations of local nodes back to the master processor — the only processor to have correct information about all node locations. Iterations continue untill the solver reaches a predetermined convergence criterion. Having a master processor synchronize all communication may lead to a bottleneck. However, regardless of the number of processors, the amount of information transmitted to the master processor during each iteration remains constant and has size $O(N)$ where N is the number of nodes. Experimental results suggest that this potential bottleneck is not a problem for input data and processor configuration.

4.8 Results

We summarize our results for the parallel linear solvers in this section. For our experiments we used the placement benchmarks published in [1]. These benchmarks are actual industrial benchmarks made public by IBM Corporation for research [5]. All our experiments were conducted on IBM-SP2 machines at the Center for Advanced Computing at the University of Michigan. The SP2 primary node has 176 160 MHz Power2 super chip processors. IBM's High Performance Switch II in the SP2 achieves low latency (as little as 64 microseconds) for MPI with a peak bidirectional bandwidth per CPU of up to 160 MB/second. All code was compiled with the xlC compiler from IBM with -O3 optimization.

We first compare the original serial implementation rather than the parallel implementation running on one processor. The legacy code was written in C++ and the SOR engine was operating on a netlist representation used by other placement routines. While parallelizing the original code we had to convert from the original netlist format to the CSR format explained in section 4.4. According to

Circuit	#Nodes	#Nets	#Pins	#Iter	#Proc 1		#Proc 2		#Proc 4		#Proc 8	
					Total Time(s)	SOR Time(s)	Total Time(s)	SOR Time(s)	Total Time(s)	SOR Time(s)	Total Time(s)	SOR Time(s)
ibm05	29347	28446	126308	792	125	121	32	26	31	24	34	28
ibm06	32498	34826	128182	2044	340	337	91	86	64	59	68	63
ibm07	45926	48117	175639	2094	494	490	194	188	85	79	61	53
ibm08	51309	50513	204890	2132	1059	1051	441	429	140	126	117	105
ibm09	53395	60902	222088	2345	743	738	271	263	85	78	74	67
ibm10	69429	75196	297567	2077	1110	1102	488	476	160	147	87	76
ibm11	70558	81454	280766	2387	864	858	371	361	128	121	79	72

Table 3: Results of SOR engine on multiple processors. The characteristics of the benchmarks are given. "#Iter" is the number of iterations required to converge to a solution. "Total Time" is the time taken by the solver including the ParMetis runtime and the time required to construct the new database from old legacy format. "SOR Time" is the time taken by the SOR engine in the solver.

Circuit	#Nodes	#Nets	#Pins	#Iter	Serial Time(s)	Parallel (1 proc) Time(s)
ibm05	29347	28446	126308	792	709	125
ibm06	32498	34826	128182	2044	1727	337
ibm07	45926	48117	175639	2094	2220	490
ibm08	51309	50513	204890	2132	4140	1051
ibm09	53395	60902	222088	2345	3240	738
ibm10	69429	75196	297567	2077	4260	1102
ibm11	70558	81454	280766	2387	3720	858

Table 4: Legacy serial code vs. Parallel code run on single processor. Significant differences in run-time are due to more efficient data-structures and thus improved cache-performance.

Table 4, using the new data format lead to significant speed-ups. We feel that this is effect is primarily because of improved cache performance. At each node, the older representation stores a lot of data not used by the linear solver, and these data degrade cache performance.

Table 3 shows the results for the parallel version of the linear solver on different processors. We have given the different characteristics of the designs used in the table. As seen from table 3 we achieved very good speedup on large benchmarks. For small benchmarks (ibm05) the speedup obtained was not very good for large number of processors. This is to be expected as for small problems the time is spent on book-keeping, rather than actual solver. We also notice that the percentage of time spent on partitioning the data by ParMetis is very small compared to the time spent on the SOR solver. Also the partitions generated by ParMetis seem to be of very high quality since we are getting very good speedups. That implies that there is not a lot of communication overhead between the processors and that most nodes are local to respective processor. Figure 10 shows the efficiency vs. # processors plot for all the data sets. It shows a consistent super-linear speedup for the larger benchmarks. We believe this is because as we divide problems between processors using min-cut partitioners, we are increasing the locality and hence the cache performance. Since the inner loop of the SOR iteration is very tight, cache effects seem to dominate the computation time as seen from tables 4 and 3. This also shows that ParMetis is producing high quality partitions.

We have demonstrated that we can successfully parallelize the SOR algorithm to iteratively solve large sparse linear systems. We have applied our methods to only a single-bin case in which the nodes are free to move anywhere. However, in VLSI placement problems we encounter problems with additional constraints like multiple bins and nodes belonging to a particular bin are constrained to only stay within that bin. Our method can be extended to handle such constraints.

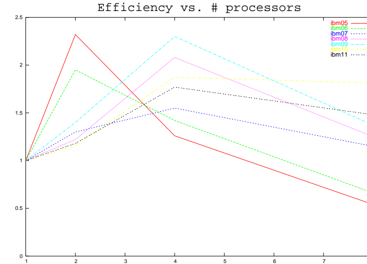


Figure 10: Efficiency vs. #Processors. As seen we were able to achieve super-linear speedup for most of the large benchmarks. We believe this is because of better locality and hence better cache performance.

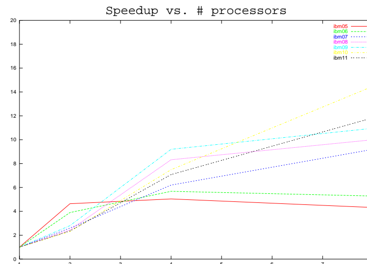


Figure 11: Speedup vs. #Processors.

Given that Krylov Sub-Space solvers like the may achieve faster convergence than SOR, our techniques of using ParMetis to partition the problem and using the distributed CSR format can potentially be applied to other high performance linear iterative solvers.

5. CONCLUSIONS

In this work we studied the performance of VLSI placers on several types of netlists, and discovered that each existing algorithms performs poorly on some type of netlists. Aside from particular improvements to the min-cut placer Capo, this motivates our interest in hybrid placers, similarly to other recent works [14, 2] that emphasize timing-driven placement and whitespace manage-

ment. To this end, we show that a hybrid placer that extends the well-known min-cut framework with analytical terminal propagation significantly outperforms plain min-cut. While min-cut placers have been parallelized before, our work addresses the parallelization of analytical placers and demonstrates that, with proper datastructures, one can achieve very attractive scalability as the number of processors increases.

6. REFERENCES

[1] S. N. Adya and I. L. Markov, "Consistent Placement of Macro-Blocks using Floorplanning and Standard-Cell Placement", *International Symposium of Physical Design (ISPD), 2002.*

[2] C. J. Alpert, G.-J. Nam and P. G. Villarrubia, "Free Space Management for Cut-Based Placement", *Intl. Conf. on Computer-Aided Design (ICCAD), 2002.*

[3] C. J. Alpert and A. B. Kahng, "Recent direction in netlist partitioning: A survey", *INTEGRATION: The VLSI J. vol. N 19*, pp. 1-81, 1995.

[4] C. J. Alpert, J.-H. Huang and A. B. Kahng,"Multilevel Circuit Partitioning", *DAC 1997*, pp. 530-533.

[5] C. J. Alpert, ISPD-98 circuit benchmarks, "The ISPD98 Circuit Benchmark Suite", http://vlsicad.cs.ucla.edu/~cheese/ispd98.html

[6] A. E. Caldwell, A. B. Kahng, I. L. Markov, "VLSI CAD Bookshelf" http://vlsicad.eecs.umich.edu/BK

[7] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Can Recursive Bisection Alone Produce Routable Placements?", *DAC 2000*, pp. 477-482.

[8] A. E. Caldwell, A. B. Kahng and I. L. Markov, "Improved Algorithms for Hypergraph Bisection", *ASP-DAC 00*, pp. 661-666

[9] C. C. Chang, J. Cong and M. Xie, "Optimality and Scalability Study of Existing Placement Algorithms" *To appear in Proc. ASP DAC, 2003*

[10] K. Doll, F. M. Johannes and K. J. Antreich, "Iterative Placement Improvement By Network Flow Methods". *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol.13, (no.10), Oct. 1994. pp. 1189-1200.

[11] A.E. Dunlop and B. W. Kernighan, "A Procedure for Placement of Standard Cell VLSI Circuits", *IEEE Transactions on Computer Aided Design 4(1), 1985*, pp. 92-98

[12] S. Dutt, "Effective Partition-Driven Placement with Simultaneous Level Processing and a Global Net View", *ICCAD 2000*, p. 254.

[13] Hans Eisenmann and Frank M. Johannes, "Generic Global Placement and Floorplanning", *DAC 1998*, p. 269

[14] A. B. Kahng, S. Mantik and I. L. Markov, "Min-max Placement For Large-scale Timing Optimization" *ISPD 2002*, pp. 143-148.

[15] C. Fidducia and R. Mattheyses, "A linear time heuristic for improving network partitions", *DAC 1982*

[16] G. Karypis, R. Agarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design", *DAC '97*, pp. 526-529

[17] J. Kleinhans, G. Sigl, F. Johannes and K. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 10 (3), March 1991. pp. 356-365.

[18] W. H. Press, et al "Numerical Recipes in C, The Art of Scientific Computing", 2nd ed.

[19] G. Karypis and V. Kumar, "A coarse-grained parallel multi-level k-way partitioning algorithm", *8'th SIAM conference on Parallel Processing for Scientific Computing*

[20] S.-L. Ou and M. Pedram, "Timing-driven Placement based on Partitioning with Dynamic Cut-net Control", *Proc. Design Automation Conf. 2000*, pp. 472-476.

[21] G. Karypis and V. Kumar, "Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs", *SuperComputing 96*

[22] G. Sigl, K. Doll and F. M. Johannes, "Analytical Placement: A Linear or Quadratic Objective Function?" *Proc. Design Automation Conf. '91*, pp. 57-62.

[23] P. R. Suaris and G. Kedem, "Quadrasection: A New Approach to Standard Cell Layout", ICCAD '87, pp. 474-477

[24] R.-S. Tsay, E. Kuh, and C. P. Hsu, "PROUD: A Sea-Of-Gate Placement Algorithm", *IEEE Design and Test of Computers*, vol.5, (no.6), Dec. 1988. pp. 44-56.

[25] J. Vygen, "Algorithms for Large-Scale Flat Placement", *Proc. ACM/IEEE Design Automation Conf.*, June 1997, pp. 746-751.

[26] M. Wang, X. Yang and M. Sarrafzadeh, "Dragon2000: Standard-cell Placement Tool for Large Industry Circuits", *ICCAD 2000*.

[27] X. Yang, B.-K. Choi and M. Sarrafzadeh, "Routability Driven White Space Allocation for Fixed-Die Standard-Cell Placement" *ISPD 2002*, p. 42.

[28] X. Yang, B.-K. Choi and M. Sarrafzadeh, "Timing-Driven Placement using Design Hierarchy Guided Constraint Generation", *ICCAD 2002*, pp. 42.