A Parallel Integer Programming Approach to Global Routing

Tai-Hsuan Wu, Azadeh Davoodi
Electrical and Computer Engineering

Jeffrey T. Linderoth
Industrial and Systems Engineering

University of Wisconsin - Madison WI 53706
{twu3,adavoodi,linderoth}@wisc.edu *

ABSTRACT

We propose a parallel global routing algorithm that concurrently processes routing subproblems corresponding to rectangular subregions covering the chip area. The algorithm uses at it core an existing integer programming (IP) formulation—both for routing each subproblem and for connecting them. Concurrent processing of the routing subproblems is desirable for effective parallelization. However, achieving no (or low) overflow global routing solutions without strong, coordinated algorithmic control is difficult. Our algorithm addresses this challenge via a patching phase that uses IP to connect partial routing solutions. Patching provides feedback to each routing subproblem in order to avoid overflow, later when attempting to connect them. The end result is a flexible and highly scalable distributed algorithm for global routing. The method is able to accept as input target runtimes for its various phases and produce high-quality solution within these limits. Computational results show that for a target runtime of 75 minutes, running on a computational grid of few hundred CPUs with 2GB memory, the algorithm generates higher quality solutions than competing methods in the open literature.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids

General Terms

Algorithms, Design

Keywords

Global Routing, Integer Programming, Parallelism

1. INTRODUCTION

Global routing is a major step in the design flow of Integrated Circuits and can influence crucial factors such as manufacturability and timing. In the past few years, it has received high attention from the research community, resulting in continuously-improving algorithms such as BoxRouter[3], FGR[7], NTUgr[2], NTHU-Route[1] and FastRoute[10].

*This research is supported by National Science Foundation under award CCF-0914981.

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

Recently, a routing method called GRIP [8] was proposed that relies heavily on integer programming (IP) techniques. GRIP works by decomposing the chip into rectangular subregions together with their net assignments to form smaller-sized "subproblems". It then applies an IP-based procedure to solve the subproblems in a systematic order.

By applying IP in a systematic manner, GRIP obtained a significant improvement in solution quality for benchmark instances but with prohibitively long execution runtimes.

The focus of this paper is to eliminate the bottlenecks to a parallel implementation based on IP. An obvious way to parallelize GRIP would be to parallelize the branch-and-bound search of each subproblem. However, achieving high efficiency from a parallel IP solver running on hundreds of concurrent processors is a difficult task and an area of active research [9, 6]. Similar to GRIP, the approach taken here works by decomposing the chip into subproblems but one in which subproblems may be routed independently, ensuring (through a one-time synchronization) that resulting routings of the subproblems can be effectively patched together. The patching itself is also accomplished by IP. The end result of the work is a parallel global router that is based on an extended IP procedure of GRIP, but allows for concurrent processing of the subproblems and significant parallelism.

There are several challenges to obtaining high-quality solutions from a parallel global router that relies on concurrent processing of subproblems. The first challenge is effective decomposition of the routing problem into subproblems—this step can significantly impact the final solution quality. The second challenge is to generate the subproblem solutions in a manner that later facilitates their connectivity and avoids overflow. Our work addresses both of these challenges. Specific contributions of our work include the following items.

1. To form the rectangular subregions and the corresponding subproblems, we extend GRIP to include a formal procedure for the initial estimation of the distribution of the nets. This step is crucial to obtain a high quality routing solution and to achieve balanced subproblems.
2. In order to effectively achieve concurrent processing of individual subproblems, we employ a *one-time* synchronization approach so that significant portions of the computation can occur completely without centralized control. This synchronization is via our novel use of an integer programming "patching" procedure.
3. Our procedure can accept as input a target runtime and produce a high-quality solution within this limit. The runtime can alternatively be expressed as limits on number of iterations of each computational step.

We also extend the IP formulation of GRIP to explicitly minimize overflow, and use various instances of it as a core component at different phases of our massively parallel procedure to avoid overflow. We also introduce a parallel procedure to independently connect neighboring subproblems.

Similar to GRIP, our approach has low memory requirements as it loads individual subproblems within the local memory of each CPU or core. Specifically in our experiments, cores with a maximum of 2GB of memory were required. The resulting algorithm was highly scalable, concurrently using up to 725 cores while solving the ISPD 2007/2008 benchmarks. In GRIP, parallelism was limited to roughly 20 concurrent processes. Our routing procedure also achieves high quality solution with a runtime limit of 75min, both in terms of wirelength and overflow. *Our goal is to demonstrate a practical global router which can work on distributed computing or future many-core computing platforms.*

The remainder of the presentation is organized into four sections. Section 2 summarizes GRIP. Section 3 explains details of our procedures. Simulation results are in Section 4, and conclusions are offered in Section 5.

2. SUMMARY OF GRIP

A mathematical description of the global routing problem, which is an extension of the one given in GRIP [8], goes as follows. We are given a grid-graph $G = (V, E)$ describing the network topology, a set of (multi-terminal) nets given by $\mathcal{N} = \{T_1, T_2, \ldots, T_N\}$, (with $T_i \subset V$), and edge capacities u_e and weights c_e $\forall e \in E$. Denote by $\mathcal{T}(T_i)$ the collection of *all* Steiner trees (routes) connecting the terminals in T_i, and let the parameter $a_{te} = 1$ if Steiner tree t contains edge $e \in E$, $a_{te} = 0$ otherwise. Define the binary decision variable x_{it} that is equal to 1 if and only if net T_i is routed with route $t \in \mathcal{T}(T_i)$. An integer program for the global routing problem can be written as

$$\min_{x,s} \sum_{i=1}^{N} \sum_{t \in \mathcal{T}(T_i)} c_{it} x_{it} + \sum_{\forall e \in E} Q_e o_e \qquad \text{(ILP-GR)}$$

$$\begin{cases} \sum_{t \in \mathcal{T}(T_i)} x_{it} = 1 & \forall i = 1, \ldots, N \\ \sum_{i=1}^{N} \sum_{t \in \mathcal{T}(T_i)} a_{te} x_{it} \leq u_e + o_e & \forall e \in E \\ x_{it} = \{0, 1\} & \forall i = 1, \ldots, N, \forall t \in \mathcal{T}(T_i), \\ o_e \geq 0 & \forall e \in E. \end{cases}$$

The parameter c_{it} is the cost of route t for net T_i which is computed as the total length of the route, given by $c_{it} = \sum_{e \ni t} c_e$, where the notation $e \ni t$ denotes that edge $e \in E$ is contained in route $t \in \mathcal{T}(T_i)$. The costs of vias are also incorporated to compute the cost of a 3D route.

In the above formulation, the first set of equations enforces the routing of each net; for each net T_i exactly one route will be selected. The second set of equations enforces the edge capacity constraint. The decision variable o_e will be positive if routing of the nets on edge e results in overflow, and the objective function trades off the total routing length with the degree of overflow. Typically Q_e is chosen sufficiently large to avoid overflow as much as possible. An advantage of this formulation is that it works with 3D routes $\mathcal{T}(T_i)$, avoiding a traditional layer assignment phase, similar to [7].

The formulation ILP-GR is slightly different than the formulation used in GRIP. The IP formulation of GRIP does not contain overflow variables o_e. Rather, the GRIP formulation has a slack variable for each net, and maximizes the number of routed nets by heavily penalizing unrouted nets along with minimizing the wirelength in its objective.

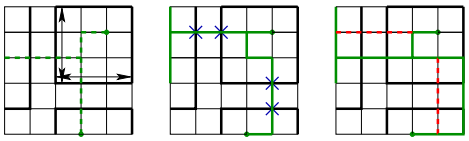

Figure 1: GRIP solves a subproblem with some flexibility in routing the "inter-region" nets.

Maximizing the number of routed nets does not correspond exactly to minimizing overflow. The formulation ILP-GR directly attempts to minimize overflow. Moreover, different penalties Q_e can be assigned for each edge, which will be helpful in avoiding overflow in our parallel implementation.

We approximately solve (ILP-GR) by the two-phase price-and-branch procedure of GRIP. Even though the formulation (ILP-GR) is slightly different than that of GRIP, the procedure to solve it is identical to the one discussed in [8].

First, we perform a *pricing* phase based on an iterative column generation [5] to (approximately) solve the linear programming relaxation of (ILP-GR). This phase identifies a set of promising candidate routes for each net generated by the package Flute [4] in conjunction with a weighted shortest-path procedure. Restricted routing rules may also be incorporated within the pricing phase. Finally, a branch-and-bound-based IP solver is used to find one route for each net from the set of its candidate routes.

Doing this price-and-branch procedure for a large global routing benchmark is computationally impractical. Therefore GRIP defines rectangular subregions as subproblems to be individually solved using the IP-based procedure. A fundamental obstacle to this spatial decomposition approach is routing the nets which have terminals in multiple subproblems. GRIP solves the IP formulation on a modified grid wherein these "inter-region" nets can be routed anywhere through a specified boundary of a subproblem as shown by the arrows in Fig. 1 (left). After solving a subproblem, fixed "pseudo-terminal" locations will be known on the subproblem boundaries, and must be honored by the neighboring subproblems that have not been solved yet (Fig. 1 (middle)). To obtain a complete solution, the subproblems are solved in a sequential order, with limited parallelism, allowing non-neighboring subproblems to be concurrently processed.

Subproblems in GRIP were solved in descending order of "difficulty," and the ordering was crucial component in obtaining high-quality solutions. The intuition behind this observation is that the more "difficult" subproblems require higher flexibility in their pseudo-terminal locations on the boundary, and the flexibility decreases as more subproblems fix pseudo-terminal locations. Once all subproblems are processed, to further improve the solution quality, the segments connecting the inter-region route fragments in adjacent subproblems are removed and then more effectively reconnected using a similar IP-based procedure (Fig. 1 (right)).

3. PARALLEL GLOBAL ROUTING

In this section, we discuss the details of our parallel global router that removes the requirement of sequential processing of subproblems. Similar to GRIP, we first generate a routing solution for each subproblem, and then attempt to connect these partial routing solutions. Unlike GRIP, these computations can be done almost completely independently. Our parallel global router is also fundamentally different from GRIP in the way it uses the IP formulation (ILP-GR) at different stages of the algorithm, and in the manner in which candidate routes to populate the IP (ILP-GR) are generated.

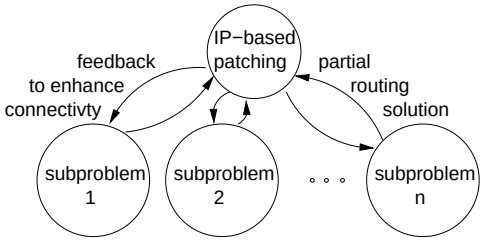

Figure 2: Overview of our parallel global router

Figure 2 gives an overview of our approach. When solving individual subproblems, we modify the pricing procedure for candidate route generation so that each subproblem receives one-time feedback encoding information about the candidate routes of its neighboring subproblems. Given this information, the subproblem solver "reprices" routes in order to generate candidate route fragments that are more likely to connect without overflow.

More specifically, the subproblems first undergo a quick, initial phase to generate a small set of candidate routes. Next, each subproblem sends information on the utilization of its boundaries by inter-region candidate routes to one or more "master" CPU(s). The master CPU(s) then considers pairs of neighboring subproblems. For each pair, a "patching" integer program is solved that locates a desired window on the subproblem boundary for the pseudo-terminal for each inter-region net. The subproblems then incorporate this feedback in a (longer) reprice procedure to generate candidate routes that obey these restrictions on location of the pseudo-terminal and are more likely to connect neighboring subproblems without overflow. After adjusted pricing, a routing solution is generated for each subproblem using a branch-and-bound based IP solver. In a final phase, a parallel and distributed IP-based procedure is applied to connect the route fragments from neighboring subproblems.

Another important aspect of our parallel global router is generation of the individual subproblems. Defining the initial subproblems can highly impact the final solution quality (as we show in our simulations).

In this section, we provide more details about each step of our procedure—subproblem generation (Sec. 3.1), initial pricing (Sec. 3.2), patching (Sec. 3.3), repricing (Sec. 3.4), and parallel connection of neighboring subproblems (Sec. 3.5).

3.1 Defining Subproblems

Defining the subproblems is a crucial step in our procedure, significantly affecting the solution quality and runtime. Poorly defined subproblems contain highly congested areas with many nets. Congested subproblems are usually difficult and may result in overflow. In addition, the subproblems typically take much longer to solve, resulting in idle time in our parallel procedure that relies on finishing all subproblems before connecting them.

Two tasks are accomplished by subproblem definition—subproblem boundaries are specified and nets are assigned to the subproblems. There are different ways to accomplish these tasks. One way is to first define the boundaries, for example via recursive bi-partitioning of the chip area. The assignment of each net is defined next, for example based on its 2D-projected route given by the package Flute [4]. However, defining the assignments solely based on the "Flute estimate" can result in highly-congested subproblems.

To mitigate the congestion, one could attempt to detour the routes generated by Flute into less congested subregions, for a better assignment. However, detouring requires knowledge of the subproblem boundaries. On the other hand, defining the boundaries without considering an estimate of the routes and congestion hot-spots during bi-partitioning might significantly limit the amount of detouring. Therefore the two tasks of subproblem definition are inter-dependent.

GRIP performs subproblem definition by first defining the subproblem boundaries via recursive bi-partitioning. When partitioning, GRIP attempts to balance the average utilization of the grid-edges (AEU) of the 2D-projected route of each net generated by Flute. GRIP then orders the subproblems based on the value of the total edge overflow (TEO) (estimated using the 2D Flute routes). The most difficult subproblems (the subproblems with the highest TEO), are solved first. Before solving a subproblem, GRIP quickly applies a perturbation to some of the Flute routes in order to detour them outside the subproblem into its neighboring ones that have lower TEO. This detouring is done in the sequential order, right before each subproblem is solved.

A primary contribution of our work is to extend GRIP to obtain a more effective and formal procedure for subproblem definition. The procedure works as follows:

1) The first step is to generate a routing of all nets to guide the bi-partitioning. GRIP relies solely on Flute to generate this routing, while we combine Flute with the IP formulation (ILP-GR) in the following manner. First, Flute is used to generate projected 2D routes for each net. The short nets are fixed in place, and the linear programming relaxation of (ILP-GR) is solved. In (ILP-GR), the parameter Q_e corresponding to edge overflow O_e is set to 1 for all $e \in E$. In practice, we provide as input a target runtime limit (controlling the number of iterations of column generation) after which we stop the procedure to get a fractional solution to (ILP-GR). We then associate a weight with each route proportional to its fractional value in the solution to the relaxed problem. The weights are used to select one route for each net via a random procedure where the probability of selecting a route is proportional to its weight. This is a well-known "randomized rounding" procedure and is better than selecting the route with highest fractional value for each net, as we also verified in our implementation.

2) Using the estimated routing generated in Step (1), recursive bi-partitioning is applied to define the problems boundaries. When partitioning, the total number of nets is used as the metric to balance at each step of the bi-partitioning. Our computational experience indicated that this metric (as opposed to the AEU used by GRIP) was more highly correlated with the final solution quality and did a better job of balancing the computational effort (pricing and branch-and-bound) for solving each subproblem. The bi-partitioning is terminated when the number of nets in a subproblem is smaller then 4000, a value empirically determine based on observing the runtime of many subproblems.

3) After fixing the boundaries, we traverse the subproblems sequentially and apply the detouring procedure of GRIP [8]. The subproblems are processed in order of their estimated TEO from the solution obtained in Step (1). Note that we are not *solving* the subproblem, but merely perturbing some of the assignments made in Step 1 to obtain more balanced subproblems. This step has negligible contribution to the runtime of our routing procedure.

3.2 Initial Pricing at the Subproblems

After defining the subproblems, we apply an initial procedure to estimate the utilization of boundaries and the location of the pseudo-terminals to connect inter-regions nets for each subproblem. This initial procedure is done independently for each subproblem, which implies that we allow the generation of candidate routes for inter-region nets that may connect *anywhere* on the boundary of the subproblem. After the initial pricing is completed, information from adjacent subproblems is sent to a "patching" process (see Section 3.3) that determines a window (restricted region) on the boundary for the location of each pseudo-terminal.

The initial pricing is done by solving the (linear programming relaxation) of the formulation (ILP-GR) as described in Section 2. A time-bound (or iteration limit) is imposed on the initial pricing phase. In our experiments, a limit of five minutes was used for this step.

The (ILP-GR) formulation requires the definition of parameters Q_e for each edge overflow variable o_e. In the initial pricing phase, we set Q_e to be equal to the Manhattan distance of edge e from the center of the subproblem. Thus, grid edges that are closer to the boundaries have a larger overflow penalty. As we have previously noted, a major goal (and challenge) of the concurrent processing of subproblems is to avoid overflow in the boundaries when connecting the subproblems. The weighted overflow penalization is an important factor towards achieving this goal.

In the initial pricing phase, inter-region nets are allowed to have a pseudo-terminal anywhere on the corresponding subproblem boundary (see Fig. 1(a)). In order to assess the utilization of boundaries by pseudo-terminals in a subproblem, it is important to generate candidate routes for *all* the nets in the subproblem, not only the inter-region ones.

3.3 Distributed IP-based Patching

Patching is an IP-based procedure that receives as an input two neighboring subproblem boundaries and the locations of pseudo-terminals on the boundaries from the initial pricing phase. A separate patching procedure is applied for each pair of neighboring boundaries. The purpose of patching is to generates feedback to the corresponding two subproblems to enhance their connectivity through subsequent repricing phase.

Consider the example in Fig. 3(a). Two neighboring boundaries are shown along with pseudo-terminals of candidate routes at each boundary. Here M_i and L_i reflect the pseudo-terminals of net i based on the generated candidate routes of subproblems SP_M and SP_L, respectively. Here, net 1 has two pseudo-terminals (denoted by M_1) on one boundary, and two pseudo-terminals (denoted by L_1) on the other. Net 2 is also given. It has one pseudo-terminal on one boundary and two on the other one.

Patching simultaneously considers the connection combinations for *all* the nets crossing the two boundaries (e.g., the four combinations of net 1 along with the two combinations of net 2 in Fig. 3(a)). Each of these combinations is encoded as a spanning window on the boundaries as shown in Fig. 3(b). The output of patching is one "restricting" window for each net, describing the permissible range of locations of its two pseudo-terminals on the two boundaries. This restricting window is selected from the set of existing windows. For each net, one window is generated for the two boundaries, but different nets can have different windows.

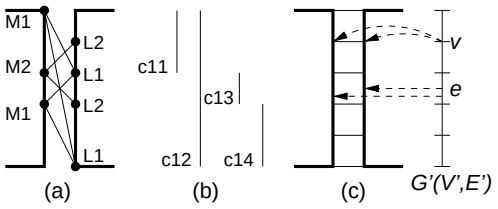

Figure 3: Patching procedure.

These windows are then passed as the feedback to the two subproblems during the repricing phase.

The patching problem can be posed as an integer program. Assume a routing grid-graph $G' = (V', E')$, where $v \in V'$ if v is a vertex, and $e \in E'$ is an edge on the boundary of one of the subproblems. An example graph is given Fig. 3(c). Edges $e \in E'$ are given a modified capacity that is the sum of the capacities of the boundary edge in one subproblem and its "mirror" edge in its neighboring one.

For each net i, the IP considers $|L_i| \times |M_i|$ possible combinations for connecting its two portions. For net i, each possible combination is denoted by a "virtual route" t spanning its virtual terminals in V'. Define the parameter $a_{te} = 1$ if virtual route t contains edge $e \in E'$, $a_{te} = 0$ otherwise. For each virtual route t for net i, define the binary decision variable x_{it} that will equal 1 if route t is selected for net i, and 0 otherwise. Define the parameter c_{it} which is the length of the virtual route t of net i, in terms of number of edges in G'. For example, in Fig. 3(b), for net 1, we have 4 combinations (virtual routes) with spans $c_{11} = 2$ to $c_{14} = 2$.

For N nets to connect, the patching problem for two neighboring boundaries is mathematically described as the following integer program:

$$\min_x \sum_{i=1}^{N} \sum_{t=1}^{|L_i|\times|M_i|} c_{it}x_{it} + \sum_{i=1}^{N} Qs_i \qquad \text{(ILP-PATCH)}$$

$$\begin{cases} \sum_{t=1}^{|L_i|\times|M_i|} x_{it} + s_i = 1 & \forall i = 1, \ldots, N \\ \sum_{i=1}^{N} \sum_{t=1}^{|L_i|\times|M_i|} a_{te}x_{it} \le u_e & \forall e \in E_v \\ x_{it} = \{0, 1\} & \forall i = 1, \ldots, N, \forall t = 1, \ldots, |L_i||M_i|. \end{cases}$$

The first set of equations enforces selection of one virtual route for each net. The parameter Q is set be large enough to force all "slack variables" s_i to take value zero, if possible. The second set of equations ensures that the given virtual edge capacities are not exceeded. If $s_i = 0$ for net $i \in N$, then exactly one x_{it} variable will be 1 for net i. The corresponding virtual route t, characterized by its two vertices in V', specifies the window on the boundaries of subproblem for net $i \in N$. All subsequent routes generated by the next pricing phase must obey this constraint. If $s_i > 0$ for net $i \in N$ in the solution to (ILP-PATCH), this indicates that inter-region net i is very difficult to route effectively, so its window is set to the entire boundary of the subproblem.

The parallel routing algorithm solves one patching problem for each pair of subproblem boundaries that share at least one inter-region route. These instances of the patching IP are independent from one another and can be solved in a distributed manner by many CPUs or, since the CPU time required to solve the patching IPs is minimal, by a single designated processor, as in our implementation.

Table 1: Results for ISPD 2007 and ISPD 2008 benchmarks. The wirelength (WL) is scaled to 10^5.

Benchmark	PGRIP							GRIP			FGR[7]		FastRoute[10]		NTHU-Route[1]	
	TOF	WL	Edge	Via	#SP	WCPU	TCPU	TOF	WL(%)	TCPU	TOF	WL(%)	TOF	WL(%)	TOF	WL(%)
adaptec1(07)	0	82.3	36.5	45.8	90	76	2101	0	-1.56	2247	0	7.00	0	9.60	0	7.38
adaptec2(07)	0	83.4	33.8	49.6	110	76	2704	0	-1.24	2677	0	7.20	0	8.90	0	8.21
adaptec3(07)	0	186.5	97.5	88.9	211	77	6319	0	-0.58	5168	0	6.61	0	8.87	0	7.15
adaptec4(07)	0	173.2	91.5	81.7	221	79	5221	0	-0.52	5258	0	3.44	0	7.36	0	6.88
adaptec5(07)	0	241.5	104.8	136.6	280	77	3175	0	-1.07	7133	0	7.13	0	10.79	0	7.20
newblue1(07)	0	84.9	25.0	59.9	122	76	2306	0	-1.14	3076	526	9.97	0	7.46	0	6.71
newblue2(07)	0	123.3	48.2	75.1	215	77	4192	0	-1.55	5228	0	4.73	0	9.11	0	8.43
newblue3(07)	41K	156.3	76.0	80.3	258	82	14590	53K	-1.03	6768	30K	10.02	32K	14.17	31K	6.38
Avg. Impr.									-1.09			6.58		8.87		7.42
newblue4(08)	132	124.9	83.4	41.4	255	77	2944	152	-0.44	3974	262	3.65	144	6.78	138	4.29
newblue5(08)	0	223.9	147.7	76.0	504	80	4953	0	-0.44	6598	0	3.95	0	5.47	0	3.38
newblue6(08)	0	172.0	102.5	69.5	459	78	2219	0	-0.88	5096	0	4.61	0	5.83	0	2.78
newblue7(08)	54	338.4	189.8	148.6	725	86	4788	74	-0.83	5377	1458	3.37	62	5.17	68	4.22
bigblue1(08)	0	54.0	37.3	16.7	124	76	956	0	-0.54	2770	0	5.81	0	6.72	0	3.49
bigblue2(08)	0	86.5	48.4	38.1	243	77	3411	0	-0.64	3793	0	5.38	0	9.50	0	4.50
bigblue3(08)	0	126.5	78.7	47.8	326	78	2690	0	-0.24	3448	0	4.20	0	3.24	0	3.22
bigblue4(08)	176	221.1	122.0	99.1	453	82	3096	186	-0.22	4400	414	4.54	152	8.50	162	4.30
Avg. Impr.									-0.53			4.44		6.40		3.77

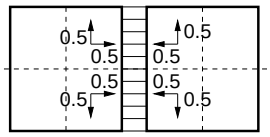

Figure 4: Uniform allocation of routing resources for parallel-connecting the boundaries.

3.4 Adjusted Pricing at the Subproblems

After each patching IP (ILP-PATCH) is solved, the solution, in the form of restricted windows for each inter-region net is sent back to the processors responsible for the subproblems. At this point, previously generated candidate routes that do not connect within the specified window range are filtered from further consideration. Next, a new pricing phase begins wherein candidate routes are generated for each net while imposing the constraint that the nets can only connect to the boundaries within their specified windows. This is for the same IP formulation as explained in Section 3.2. In earlier experimental work, we tried connecting subproblems using a heuristic method, but found the IP to generate solutions of much higher quality. A time limit is imposed on the adjusted pricing phase (e.g., a limit of 20min in our experiments). Once the adjusted pricing is over, a commercial IP solver is called to generate a solution to each region's subproblem that obeys the patching window constraints.

3.5 Parallel Connecting of Subproblems

The parallel global routing procedure concludes with a final connection and polishing phase. Specifically, after concurrently solving all the subproblems, for each inter-region net, the final segment that connects its "backbone" to the subproblem boundary is removed, as shown in Fig. 2(middle).

We then fix all the nets that fall completely inside the subproblems. We also fix the backbones of the inter-region nets, and implement an IP-based price and branch procedure similar to GRIP to connect the backbones of the inter-region nets. Here we show this connection phase can be done in a distributed manner for each pair of neighboring boundaries.

As shown in Fig. 4, we divide each subproblem into quadrants. Each quadrant is adjacent to two neighboring subproblems (e.g., top-right quadrant is adjacent to the top and the right neighboring subproblems). For each routing edge, we divide its remaining capacity (not utilized by the fixed routes) into equal portions to be allocated for solving two "connection" problems of its two corresponding subproblems. For each of the two neighboring boundaries, we solve an IP-based connection problem. For example, for the two boundaries shown in Fig. 4, we use the top-left and bottom-left quadrants of the right subproblem with the top-right and bottom-right quadrants of the left one. For each edge, we use half of its remaining capacity. We then solve the (ILP-GR) procedure to connect the inter-region nets.

4. SIMULATION RESULTS

Our parallel global routing procedure, named PGRIP, was implemented in C++. For solving individual linear programs (for pricing) and integer programs (for branch-and-bound), the software packages MOSEK 5.0 and CPLEX 6.5, respectively, were used. Parallel processing of subproblems was performed by submitting jobs to a grid of hundreds of heterogeneous CPUs of 2GB memory, managed by the Condor resource management system. The algorithm was evaluated for the ISPD 07 and ISPD 08 benchmarks. This is specifically allow full comparison with the GRIP solutions.

A 10min runtime limit was imposed on solving the relaxed (ILP-GR), to define subproblems (Sec. 3.1). For the initial pricing (Sec. 3.2), repricing (Sec. 3.4), and pricing to connect the subproblems (Sec. 3.5), we set runtime limits of 5min, 20min, and 20min, respectively. For solving the IP using branch-and-bound after candidate route generation we used a runtime limit of 10min. We did not limit the patching procedure since this step was very fast. (In general the number of nets crossing between two subproblems is fairly small). As a result we report slight variation in the runtimes of our algorithm on different benchmark instances.

In Table 1, we compare the solution quality of PGRIP with existing approaches. For each benchmark, the total overflow (indicated by TOF), total cost of wirelength and via (indicated by "WL") and the breakdown between wirelength and via (indicated by "Edge" and "Via" respectively) is reported for PGRIP. For other approaches, we report the percentage improvement in total cost of wirelength and via (indicated by %WL), and the TOF. Our solutions were evaluated using the ISPD 2008 script and are available online[1].

Excluding GRIP, the PGRIP solutions improve significantly in WL of each instances (ranging from 3.37% to 10.79%). Compared to GRIP, which has the best reported solution but impractical runtimes, on-average it only has 1.1% and 0.5% degradation of ISPD 07 and 08, respectively[2].

[1]http://wiscad.ece.wisc.edu/gr/ for PGRIP solutions

[2]The GRIP solutions were downloaded from the same link.

Furthermore, the solutions obtain zero overflow for any benchmark that already had zero overflow solution from other tools. For benchmarks newblue4 and newblue7, the solutions from the parallel global router have the smallest overflows reported so far (even better than GRIP). This is likely due to a better definition of the initial subproblems and measuring overflow directly in the IP formulation.

Table 1 also reports the PGRIP's wall clock runtimes in minutes indicated by WCPU ranging from 76 to 86 minutes. These wall clock times are computed for the case when the grid would not be shared with other users. The actual wall clock time for solving the instances was larger, as jobs submit to the Condor-controlled grid often waited in the job queue while higher-priority jobs were run. PGRIP required an average walltime of 79min.

The number of subproblems (#SP) are also reported in the table for PGRIP. All the subproblems were concurrently processed so the number of parallel CPUs in our experiments were equal to the number of subproblems, on average 281 processors, indicating we were able to take advantage of a significant amount of parallelism. GRIP can on average use only 23 processors effectively [8]. Another feature of PGRIP is that its runtimes are similar over different benchmarks, indicating a high level of scalability. We also compare the total CPU times (indicated by TCPU) between PGRIP and GRIP. We regenerated these values for GRIP. As can be seen, both runtimes are quite comparable with each other.

We didn't have access to the codes of all the other routing tools to make fair runtime comparison. Additionally, NTHU-Route required 10GB memory. Thus, we don't report runtimes of the other competing methods, but simply note that the runtimes of our approach on 2GB machines is quite comparable with other tools, specifically for the benchmarks that are highly congested.

Another feature of our approach is that by changing its runtime limit, we can explore the tradeoff between runtime and solution quality; for example by allowing more runtime on the pricing phase, we can generate more candidate routes which in turn can improve the solution quality. Take adaptec1 as an example. Changing the runtime limit of the pricing phases to 30min (instead of 20min in our first experiment) results in an additional 1.13% improvement in WL. The total runtime is 96min. Reducing the runtime limit of pricing to 10min, reduces the total runtime to 57 min, but degrades the WL by 4.41%.

Next we analyze the quality of defining subproblems before starting their concurrent processing. We measure it based on an estimate of overflow after defining the subproblems. Recall in defining subproblems we apply a 3-step approach (see Sec. 3.1): 1) initial routing using a relaxed ILP, 2) defining boundaries, and 3) detouring routes of step 1 to distribute the net assignments. We estimate the edge overflow based on the routes generated in step 3, and calculate the average and maximum edge overflows in each subproblem. We report the average of each of the two quantities over all the subproblems in columns 2 and 3 of Table 2. We also report these two quantities, assuming detouring is not applied (so the edge overflow is estimated only using the routes of relaxed ILP). These are reported in columns 4 and 5. As can be seen the average and maximum overflows of a subproblem based on the estimate of routes in step 1 and step 3 are not very different. This means detouring does not result in much improvement after applying the relaxed ILP.

Table 2: Estimated overflow of initial subproblems.

Benchmark	step1+step2+step3		step1+step2		Flute+step2	
	Avg.	Max.	Avg.	Max.	Avg.	Max.
adaptec1	1.29	59	1.41	70	3.56	144
adaptec2	1.13	85	1.19	94	2.50	183
adaptec3	0.72	49	0.77	52	2.71	158
adaptec4	0.27	54	0.31	59	0.97	111
adaptec5	2.17	100	2.34	118	4.77	306
newblue1	0.43	40	0.49	43	0.94	73
newblue2	0.41	79	0.45	83	0.96	131
newblue3	0.84	660	0.92	748	2.16	1119
newblue4	1.12	88	1.13	101	2.14	147
newblue5	2.15	75	2.80	89	4.94	155
newblue6	1.03	93	1.19	113	2.68	192
newblue7	7.64	252	8.33	302	13.17	574
biglue1	19.31	87	22.78	99	28.77	167
biglue2	5.83	59	6.28	62	8.61	93
biglue3	9.43	126	10.27	147	17.26	373
biglue4	7.02	71	7.95	82	11.33	221

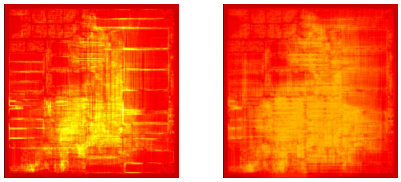

Figure 5: Congestion map of adaptec1; after subproblem generation (left), final solution (right)

To better show that the relaxed ILP is helpful to define subproblems, we also report the average and maximum overflow of the subproblems if 2D routes are taken from Flute, and then subproblems are generated using step 2. These are reported in columns 6 and 7. Now we can see that our procedures can significantly improve overflow in initial subproblems before starting their concurrent processing. Fig. 5 shows the congestion map of adaptec1 before concurrent processing of defined subproblem, and of the final solution.

5. CONCLUSIONS

A fundamental contribution of our work is to demonstrate a global routing tool which can remove the synchronization barriers between subproblems, effectively utilizing many more processors and reducing runtimes to an acceptable practical level. The parallel implementation highly relied on integer programming techniques, resulting in much flexibility and significant solution improvement. We believe this can be a very promising model for utilizing cloud computing or many-core platforms of near future.

6. REFERENCES

[1] Y.-J. Chang, Y.-T. Lee, and T.-C. Wang. NTHU - route 2.0: A fast and stable global router. In *IEEE Intl. Conf. on Computer-Aided Design*, pages 338–343, 2008.

[2] H.-Y. Chen, C.-H. Hsu, and Y.-W. Chang. High-performance global routing with fast overflow reduction. In *IEEE Asia and South Pacific Design Automation Conf.*, pages 582–587, 2009.

[3] M. Cho, K. Lu, K. Yuan, and D. Z. Pan. Boxrouter 2.0: architecture and implementation of a hybrid and robust global router. In *IEEE Intl. Conf. on Computer-Aided Design*, pages 503–508, 2007.

[4] C. C. N. Chu and Y.-C. Wong. Flute: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(1):70–83, 2008.

[5] J. Desrosiers and M. E. Lübbecke. A primer in column generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, chapter 1. Springer, 2005.

[6] J. T. Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, Georgia Institute of Technology, 1998.

[7] J. A. Roy and I. L. Markov. High-performance routing at the nanometer scale. In *IEEE Intl. Conf. on Computer-Aided Design*, pages 496–502, 2007.

[8] T.-H. Wu, A. Davoodi, and J. T. Linderoth. Grip: Scalable 3d global routing using integer programming. In *Design Automation Conference, to appear.*, 2009.

[9] Y. Xu, T. K. Ralphs, L. Ladányi, and M. Saltzman. Computational experience with a software framework for parallel integer programming. *INFORMS Journal on Computing*, 23:383–397, 2009.

[10] Y. Xu, Y. Zhang, and C. Chu. Fastroute 4.0: global router with efficient via minimization. In *IEEE Asia and South Pacific Design Automation Conf.*, pages 576–581, 2009.