Parallel Global Routing Algorithms for Standard Cells

Zhaoyun Xing
Computer and Systems Research Laboratory
University of Illinois
Urbana, IL 61801
xing@crhc.uiuc.edu

John Chandy
Cadence Design Systems
270 Billerica Road
Chelmsford, MA 01824
jchandy@cadence.com

Prithviraj Banerjee
Center for Parallel and Distributed Computing
Northwestern University
2145 Sheridan Road
Evanston, IL 60208
banerjee@ece.nwu.edu *

Abstract

In this paper, we propose three different parallel algorithms based on a state-of-the-art global router called TimberWolfSC. The parallel algorithms have been implemented by using the Message Passing Interface (MPI), and have been evaluated on a wide range of parallel platforms such as the Sun SparcCenter 1000 and the Intel Paragon. Our experimental results show good speedups and qualities from two of these parallel algorithms. We have been able to reduce runtimes of some circuits from half an hour to 5 minutes, obtained speedups of about 4.0 to 5.0 on 8 processors, with less than 2-3\% degradation of quality of the solutions.

1 Introduction

The global routing problem is an extremely important and time consuming phase of any automated layout system. For contemporary designs containing 100,000 cells and nets, global routers can easily take several hours to perform their operation. Recently, researchers have therefore turned to parallel processing to solve complex routing problems quickly and with high-quality.

Global routing in a standard cell design environment determines the feedthrough requirement distribution, the assignment of those feedthroughs, and the channels for each wire to go through. Based on existing serial algorithms, several parallel global routers have been proposed. Using the graph search approach [6, 9, 11], a parallel algorithm was implemented on a hypercube-based distributed memory multiprocessor [13]. Parallel algorithms have also been described in [8, 15, 16] which are based on an iterative improvement approach [7, 15, 16]. A hierarchical parallel algorithm was implemented on a shared memory multiprocessor [2, 4]. But those parallel algorithms had the following disadvantages: they were either targeted for a specific parallel architecture such as shared memory multiprocessors, or could only handle two-pin nets, or the qualities of the routing were poor compared to the state-of-the-art routers.

Among various serial global routing algorithms, the global router in TimberWolfSC [7] is one of the best in terms of routing quality and has the following advantages: (1) the solution quality is independent of the routing order of the nets, (2) it takes advantage of electrically equivalent pins, and (3) it handles multi-pin nets. In this paper, we propose three different parallel algorithms based on the global router in TimberWolf 6.0 \footnote{TimberWolf 7.0 is the latest version, but only the source code of TimberWolf 6.0 is available. Also, the global router in version 6 and 7 are unchanged.}. These algorithms have been implemented by using the Message Passing Interface (MPI). Our experimental results show good speedups and qualities for two of these parallel algorithms.

2 TimberWolfSC

The TimberWolfSC global router (called TWGR in the paper) fully exploits modern row-based technologies. The

*This research was supported in part by the Semiconductor Research Corporation under contract SRC 94-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G0273 administered by the Army Research Office.
goal of TWGR is to minimize the total area of the chip by minimizing the total channel density and minimizing the number of feedthroughs in various rows (which increase the row widths). TWGR roughly consists of five steps. In the first step, an approximate Steiner tree is built for each net based on the minimum spanning tree of this net. The second step is coarse global routing through which an estimation of feedthrough requirement is obtained. In this step, the core is partitioned into a coarse global routing grid. Each segment is assumed to be routed by some one bend L-shaped wire. To reduce the order dependence of the segments processed, a segment is randomly picked from the whole segment pool. By evaluating the needed feedthrough number and the channel density change when the side of an L shaped segment is switched, the L shape for this segment can be determined. After this step the feedthrough numbers needed at each grid point are roughly determined, and those needed feedthroughs will be added at each grid point. In the third step, for each row, TWGR assigns each segment which crosses this row a feedthrough from those available in this row. The fourth step connects the feedthroughs of each net with regular pins of that net by building a minimum spanning tree from a complete graph of the pins and feedthroughs in the adjacent rows. In a standard cell design, some pins may have an equivalence on the opposite side of the cell. A segment with two of this kind of pins is called a switchable net segment. The two pins of a switchable net segment can be connected by a wire in either the channel above the row or the channel below the row. To optimize the channel placement of each switchable net segment, and reduce the order dependence of the segment processed, the fifth step randomly picks one switchable net segment and determines its channel by evaluating the channel track change when the segment is flipped to the opposite channel.

Throughout this paper, we assume $n$ is the number of nets, and $n_t$ is the number of pins on net $i$.

3 Parallelization of TWGR

In standard cell VLSI design, a circuit consists of four components: row, cell, pin, and net. A row is a set of cells, each cell contains a list of pins, and each net consists of a list of pins. Pins are elements of both the cell and the net.

In order to solve large routing problems which require considerable amount of memory, we partition the rows of the circuit among the processors. Since there are computation localities among rows in TWGR, the rows are partitioned contiguously. Throughout global routing, most of the cells will stay in their rows and a cell can be moved horizontally due to the addition of feedthrough cells. To keep track of the cell information efficiently, cells are partitioned by rows among processors. That is, if a processor owns a row, then it owns all the cells in that row at the same time. Therefore, this row-wise cell partition favors horizontal parallelism.

Since a pin must be on some cell and at the same time it must be on some net, in each step of routing, the ownership of the pins will affect the memory scalability and work load balancing. We deal with the problem of partitioning the pins across the processors in our parallel algorithms. Pins can be partitioned in different ways during all the steps of TWGR except the feedthrough assignment step. In the feedthrough assignment step, since their positions may be changed along with their cells, the pins are partitioned in a row-wise manner. Depending on the way nets are partitioned and the occurrence of the pin partition, we propose several parallel algorithms which explore the parallelism implied in the pin partition.

4 Row-wise Pin Partition

The first parallel global router we will look at uses row partitioning of pins. Row partition means that in the most steps of the router, the pins are partitioned row-wise. Since the cells and blocks are partitioned in a row-wise manner already, the pin’s row-wise partition will conform with the cell and row partition. That is, if a processor owns a row then it will own all the cells in that row; likewise, if a processor owns a cell then it will own all the pins on that cell. If each net is split into sub-nets across processors, each processor will own a row-wise partitioned sub-circuit. To get the most possible coarse grain parallelism, we expect to be able to run TWGR on those sub-circuits concurrently. To ensure connectivity of a net across partitions, it might be necessary to introduce fake pins in addition to normal pins. For example, if a net has only two normal pins, but it crosses partitions $p_0$, $p_1$, and $p_2$. We cannot find any normal pins of the net which sit on the boundary of partition $p_1$. To solve this problem, if necessary, we add fake pins at the boundaries of each partition that net crosses. We let TWGR treat those pins the same as regular pins except that those pins are not attached to any
cells so that those pins do not have to be shifted along with cells when feedthroughs are added. Now we must determine where to put those fake pins on the boundary of each pair of partitions. Note that TWGR uses the Steiner tree of each net as the basis of its final connection. To remove blindness in the fake pin addition, we let one of the processors build the Steiner tree for each whole net, and then we add the fake pins according to the segments of the Steiner trees. If a segment crosses the boundary of a partition, then we add a fake pin at the crossing point, as shown in Figure 2. The building of the Steiner trees can be done in parallel. To build the Steiner trees in parallel requires net partitioning. The procedure to do this net partition in parallel is given in Section 5. After the segments are broken, each processor will have its own sub-circuits. Those broken segments will become the net segments of the processor which owns its two end points. To complete the connection of each net, we need only connect those pins on the partition boundaries for each net. It should be noted that TWGR optimizes the placements of the switchable net segments of the processor which owns its two end points. Also, to make a better optimization, the track information in the shared channel is synchronized between two adjacent processors.

![Figure 2: To break segment AB, we add fake pins C and D on the partition boundaries.](image)

5 Net-wise Pin Partition

In this section, we will derive a parallel algorithm using a net based pin partition. We propose several heuristics to partition the nets. The goal of this task is to balance the work load and to make the pins on the same partition have as much data locality as possible.

Most of the net partitioning schemes proposed here have the same generic feature. A weight is associated with each net. The first step of the partitioning scheme is to sort the weight array, whose size equals \( n \), the number of nets. Then the nets are assigned in that order to one processor until the number of pins exceeds the average pin number \( \sum n_i / p \).

**Center partition.** For a given net \( i \), assume \((x_j, y_j), j = 1, \ldots, n_i, \) are the pin locations. The center of this net is defined as \( (\sum x_i^n x_j, \sum y_i^n y_j) \). Since nets which stay around the same rows will tend to put tracks into the same channel, there will be more runtime interaction between those vertically close nets. Therefore, in center net partition we give the net \( i \) a weight of the \( y \) coordinate of its center, i.e., \( w[i] = \sum y_i/n_i \).

**Locus partition.** This is one heuristic derived from [15]. This partition tries to cluster those nets that are geometrically related. First, we find a bounding box for each net. Let \((x_i, y_i)\) be the lower left corner of the bounding box. If \( \alpha = \max\{y_i\} \) then in the locus net partition we give net \( i \) a weight \( w[i] = \alpha x_i + y_i \). The goal is to partition the net according to \( x_i \) and use \( y_i \) to break the ties.

**Density partition.** We know that rows have already been partitioned into contiguous row blocks. In the density partition, for each net \( i \) we count the number of pins which exist in the row range of each processor. If processor \( j \) has the most pins of net \( i \), then we give the net \( i \) a weight of \( j \), i.e., \( w[i] = j \).

**Pin number weight partition.** The goal of this partition is to make a better load balancing. Among the various steps of TWGR, the building of the approximate Steiner tree has the highest computation complexity. When the maximum number of pins on a net is so large that the building of the approximate Steiner trees becomes the dominant portion of the computation, the load will not be balanced even when the total numbers of pins across the processors are even. To solve this problem, we need to give much more weights to the large nets. We give net \( i \) a weight \( -n_i^\alpha \), here \( \alpha \) is a positive number. The reason we give a negative weight to each net is because we like to schedule the large nets first. Since we do not want to put all large nets on the same processor, we evenly distribute large nets in a round-robin manner.

Our experiments show that this technique works well for \( \alpha = 1.5 \) for AVQ-LARGE, one of MCNC layout synthesis benchmark circuit. AVQ-LARGE has some very large clock line nets. One of them has more than 4000 pins. But 99% of the nets have less than 5 pins. Experimental results will be given in Section 7.

A net partition will give a corresponding pin partition. In this algorithm, the pin partition does not change throughout the course of TWGR. The algorithm begins with partitioning the nets and their corresponding pins by one of the net partitioning schemes. The building of the approximate Steiner trees can be executed in parallel. While performing coarse global routing, since all processors could contribute feedthrough and track density estimation to the same coarse global routing grid, we need to synchronize the information of each grid point periodically. After coarse global routing, each processor knows about the feedthroughs added in its own rows. To do feedthrough assignment, each proces-
sor has to own a copy of all the segments which cross its rows, hence it needs to collect those segments from the other processors which own the segments. Then each processor is ready to do the feedthrough assignment. Since after the feedthrough assignment, feedthroughs are part of the nets, each processor needs to send all those feedthroughs to the processor which owns the net. All processors can connect the nets they own in parallel. When optimizing the switchable net segment locations, since all processors could assign the same switchable net segments to the same channel, the channel density information should be updated. The problem of this algorithm is that the net and pin partition will result in the interference between processors when they try to make a decision as to which channel a switchable net segment should go. This unnecessarily increases the channel density, and may result in a severe loss of quality. Although synchronization in the switchable net segment optimization step reduce this information inaccuracy, the synchronization is very costly. This is because all the processors will share all the channels and communication is more costly than computation. If we synchronize too often, we will lose runtime performance.

6 Hybrid Pin Partition

In this section, we describe an algorithm which combines the benefits of the previous two algorithms. In row-wise pin partition algorithm, a large net is broken into a couple of subnets, independently connecting pins in adjacent row may result in some unnecessary tracks. In Figure 3 the extra track CD will be added because processor 0 connects pins C and D for one sub-net and processor 1 connects pins A and B for another sub-net independently. To further reduce the quality degradation of row-wise parallel algorithm, instead of letting each processor connect the pins of a net in adjacent rows for the subnets, we let one processor do it for each whole net. As shown in Section 7, this algorithm provides the best quality. Though it could not provide as much gain in runtimes as row-wise pin partition parallel algorithm, the speeds are still quite good.

![Figure 3: A possible track increase in row-wise pin partition](image)

7 Experimental Results

We implemented our parallel algorithms for global routing described in the previous sections by using the Message Passing Interface (MPI)[10], which has become a standard for portable parallel programming and is available on a range of parallel architectures. We tested our implementations using 6 circuits from the MCNC layout synthesis benchmarks on an 8 processor SUN SparcCenter 1000 shared memory multiprocessor (SMP) and 20 processor Intel Paragon distributed memory multiprocessor (DMP). Table 1 lists some important characteristics of these 6 circuits. In this section, we report our experiments over those 6 circuits. Unless otherwise stated, all experimental results are run on the SUN SparcCenter 1000 SMP, and we use pin number weight net partition with \( \alpha = 1.5 \).

![Table 1: Characteristics of test circuits](image)

Table 2 lists the scaled track results of the row-wise pin partitioned parallel algorithm over the serial run. Figure 4 shows the speedups obtained. It can be seen that while the speedups obtained are quite high (average 5.5 on 8 processors across the circuits), the qualities of the routing become inferior with increasing number of processors (on average about 5% worse quality in track count). We note however that the scaled area results (measure of the routing quality) are not much worse (1-2%), which is not listed.

![Table 2: Scaled track results of row-wise pin partition algorithm](image)

Table 3 lists the scaled track results of the row-wise pin partitioned parallel algorithm over the serial run. Figure 4 shows the speedups obtained. It can be seen that while the speedups obtained are quite high (average 5.5 on 8 processors across the circuits), the qualities of the routing become inferior with increasing number of processors (on average about 5% worse quality in track count). We note however that the scaled area results (measure of the routing quality) are not much worse (1-2%), which is not listed.

7.1 Row-wise Pin Partition

![Table 2: Scaled track results of row-wise pin partition algorithm](image)

7.2 Net-wise Pin Partition

We next report the performance of the net-wise pin partitioned algorithm. The routing quality is controlled by frequent synchronization but this reduces the runtime performance and is very costly. In our experiment, the synchronization frequency is not very high. Table 3 lists the scaled track results over the serial run. Figure 5 shows the speedups obtained. It can be seen that the net-wise partitioned algorithm causes significant degradation in quality (15% on the
average on 8 processors) with poor speedups (average about 2.5 on 8 processors). This is due to the costly synchronization across all the channels and the blindness of each processor while doing the optimization of the switchable net segment placement.

Table 3: Scaled track results of net-wise pin partitioned algorithm

<table>
<thead>
<tr>
<th>circuit</th>
<th>1 proc</th>
<th>2 procs</th>
<th>4 procs</th>
<th>8 procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary2</td>
<td>1.00</td>
<td>1.08</td>
<td>1.15</td>
<td>1.19</td>
</tr>
<tr>
<td>biomed</td>
<td>1.00</td>
<td>1.02</td>
<td>1.08</td>
<td>1.15</td>
</tr>
<tr>
<td>industry2</td>
<td>1.00</td>
<td>1.05</td>
<td>1.06</td>
<td>1.13</td>
</tr>
<tr>
<td>industry3</td>
<td>1.00</td>
<td>1.01</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>avq.small</td>
<td>1.00</td>
<td>1.04</td>
<td>1.09</td>
<td>1.18</td>
</tr>
<tr>
<td>avq.large</td>
<td>1.00</td>
<td>1.02</td>
<td>1.06</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Figure 5: Speedup results of the net wise pin partition algorithm

7.3 Hybrid Pin Partition

We finally report the results of hybrid pin partitioned parallel routing algorithm. Table 4 lists the scaled track results over the serial run. Figure 6 shows the speedups obtained. It is clear that the hybrid pin partitioned routing algorithm obtains the best quality control (average quality is 2% worse on 8 processors) and good speedups are obtained (average 4.5 on 8 processors).

Table 4: Scaled track results of hybrid pin partition algorithm

<table>
<thead>
<tr>
<th>circuit</th>
<th>1 proc</th>
<th>2 procs</th>
<th>4 procs</th>
<th>8 procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary2</td>
<td>1.00</td>
<td>1.02</td>
<td>1.01</td>
<td>1.04</td>
</tr>
<tr>
<td>biomed</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>industry2</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>industry3</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.02</td>
</tr>
<tr>
<td>avq.small</td>
<td>1.00</td>
<td>1.01</td>
<td>0.99</td>
<td>1.03</td>
</tr>
<tr>
<td>avq.large</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Figure 6: Speedup results of hybrid pin partition algorithm

8 Conclusion

In this paper, we developed three parallel algorithms for global routing based on TimberWolf 6.0. Our experimental results show good speedups and quality from two parallel algorithms, pin row-wise partitioned algorithm and pin hybrid algorithm. From our theoretical analysis and our experiments, we can conclude the best algorithm should be row-wise pin partitioned algorithm with the pin number weight net partition. If we want a slightly better quality and are willing to sacrifice some runtime performance, then we should use the pin hybrid partitioned algorithm instead of pin row partitioned algorithm. Table 5 list the results of the hybrid pin partitioned parallel routing algorithm based on pin number weight net partition with weight 1.5 on various platforms: Sun SparcCenter 1000 SMP and Intel Paragon DMP. We report actual runtimes and qualities of results. It is clear that our algorithm is very effective in the parallelization of an extremely difficult and important problem on a wide range of platforms.

References

[1] P. Banerjee, Parallel Algorithms for VLSI Computer-
<table>
<thead>
<tr>
<th>Platform</th>
<th>Proc No</th>
<th>Results</th>
<th>Primary2</th>
<th>Biomed</th>
<th>Industry2</th>
<th>Industry3</th>
<th>Avg. Small</th>
<th>Avg. Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun SparcCenter 1000MP</td>
<td>1</td>
<td>Track No</td>
<td>25,666,283</td>
<td>51,068,160</td>
<td>113,497,300</td>
<td>674,215,300</td>
<td>83,633,285</td>
<td>103,529,700</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Area (µm²)</td>
<td>65</td>
<td>162</td>
<td>446</td>
<td>845</td>
<td>1693</td>
<td>1539</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (seconds)</td>
<td>1.04</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.03</td>
<td>1.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Area (scaled)</td>
<td>1.01</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (seconds)</td>
<td>4.33</td>
<td>4.26</td>
<td>3.88</td>
<td>4.83</td>
<td>4.47</td>
<td>1.83</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Speedup</td>
<td>0.92</td>
<td>8.04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Paragon DMP</td>
<td>1</td>
<td>Track No</td>
<td>124</td>
<td>674</td>
<td>98,497,300</td>
<td>237,137</td>
<td>12,162</td>
<td>12,060</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Area (µm²)</td>
<td>69</td>
<td>118</td>
<td>5,075</td>
<td>7,768</td>
<td>12,162</td>
<td>12,060</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (seconds)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>1.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Track No</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Area (µm²)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (seconds)</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>4.47</td>
<td>4.47</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>Track No</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Area (µm²)</td>
<td>1.02</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Time (seconds)</td>
<td>1.06</td>
<td>1.01</td>
<td>1.01</td>
<td>1.00</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speedup</td>
<td>3.04*</td>
<td>3.04*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Results of the hybrid pin partitioned parallel global routing algorithm on different platforms. Each node of Intel Paragon DMP has 32 megabytes of memory. Due to the memory limitation, we could not get serial results on industry3 and avq.large in 4 days. The speedups for those two circuits (marked with a *) are calculated by assuming that speedup is proportional to the number of processors.


