Parallel Global Routing Algorithms for Standard Cells

Zhaoyun Xing, John Chandy, and *Prith Banerjee*

Technical Report No. CPDC-TR-97-10

July 1997

Center for Parallel and Distributed Computing

Department of Electrical & Computer Engineering

Northwestern University

Technological Institute

2145 Sheridan Road, Evanston IL-60208

Abstract

For a given placement of cells in a standard cell design, the global routing problem determines how the cells will be interconnected, using which channels, and using which feedthroughs. 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. In this paper, we propose three different parallel algorithms based on a state-of-the-art global router called TimberWolf 6.0. The first algorithm partitions the pins in a row-wise manner among the processors. The second algorithm partitions the nets and their corresponding pins among the processors. The third algorithm uses a hybrid approach. 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 DMP. 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

Automated layout consists of two primary functions: (1) determining the positions of the modules (standard cells, gates, macro cells) on the VLSI chip, called *placement*, and (2) interconnecting the modules with wiring, called *routing*. Although placement and routing are intimately related and interdependent, they have been traditionally separately solved because of their computational complexities. Hence placement algorithms use some crude measures to estimate the routing complexities of various placements. The routing algorithms subsequently take a given placement and perform the exact wiring of the nets. Both placement and routing problems are NP complete; hence a large number of heuristics exist for solving the problems, which vary in their runtime requirements and qualities of layout produced. Unfortunately, routing algorithms that produce high quality layouts often take large amounts of computation time. Recently, researchers have therefore turned to parallel processing to solve complex routing problems quickly and with high-quality. In this paper, we will describe various approaches to parallelize one of the more efficient and state-of the-art global routers of standard cell designs based on TimberWolfSC [7].

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. Among the strategies proposed to solve the global routing problem are graph search [6, 9, 11], iterative improvement [7, 15, 16], hierarchical decomposition [2, 4], and integer linear programming [12, 14]. Based on existing serial algorithms, several parallel global routers have been proposed. Using the graph search approach, a parallel algorithm was implemented on a hypercube-base distributed memory multiprocessor [13]. Parallel algorithms have also been described in [8, 15, 16] which are based on an iterative improvement approach. A hierarchical parallel algorithm was implemented on shared memory multiprocessors [2]. 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. These algorithms have been implemented by using the

Message Passing Interface (MPI), which has become a standard for portable parallel programming and is available on a range of MIMD parallel architectures. Our experimental results show good speedups and qualities for two of these parallel algorithms. As shown in Section 4, 5, and 6, a careless parallelization of a complex algorithm such as TimberWolfSC global router could lead to a poor performance. The task of parallelization is to explore the parallelism as much as possible such that we can get the maximum possible speedup, and at the same time we should keep the quality of results controlled.

This paper is organized as follows. Sections 4, 5, and 6 propose three different parallel global routers based on TimberWolfSC. Section 7 tries to speedup the global routing process for the circuits with some very large nets. Section 8 shows the implementation results. Section 9 concludes this paper.

2 Overview of the Serial Global Routing Algorithm in TimberWolfSC

The TimberWolfSC global router (called TWGR in the paper) fully exploits modern row-based technologies. TWGR is based on the total channel density model. The total channel density is the sum of the densities over all of the channels. In standard cell layout, the length of the longest row determines the core width of the chip, and the height of the chip is determined by the heights of the cell rows (which is fixed by the standard cells and the placement), and the total heights of all the channels. Therefore, the 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, as listed in Algorithm 1 and illustrated in Figure 1. 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 needed feedthrough number and 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

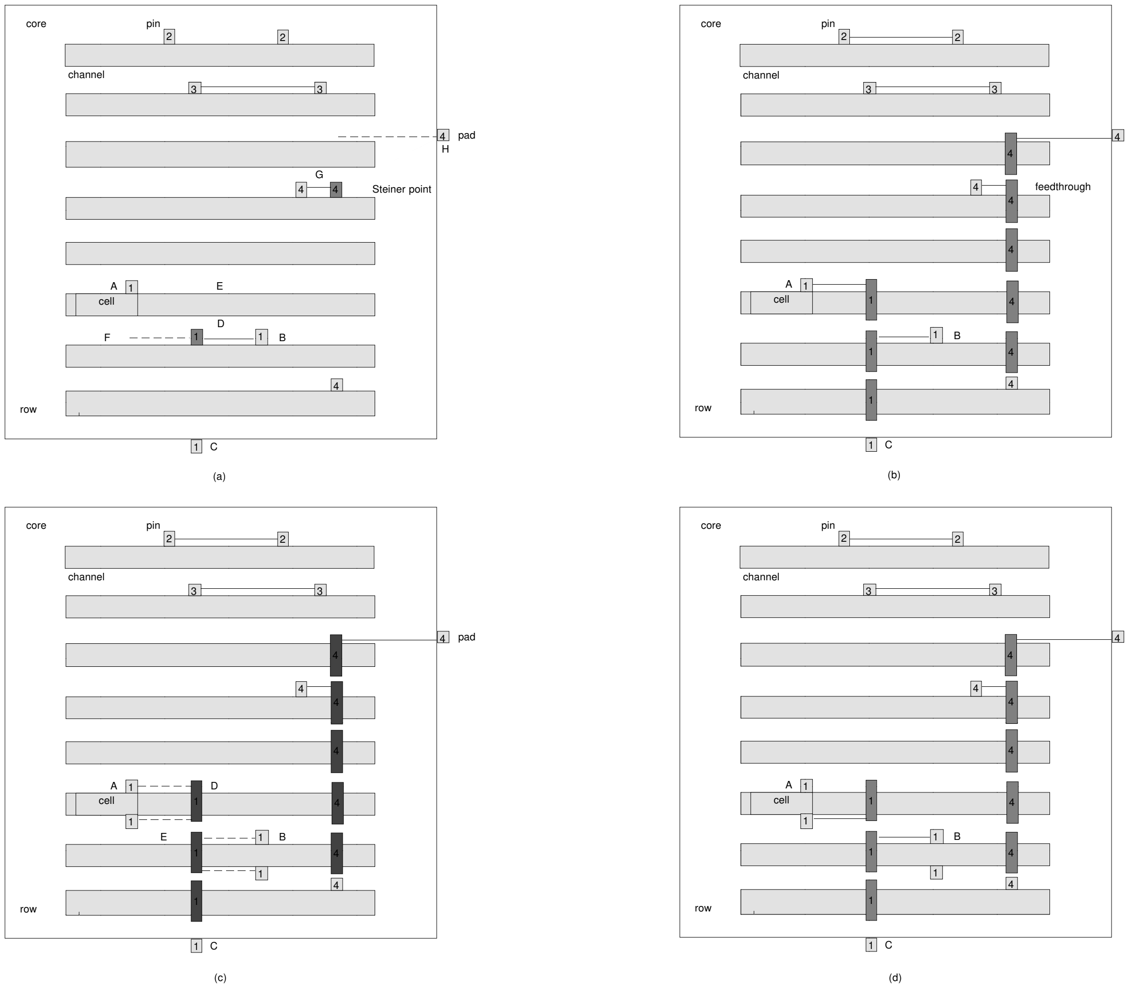

Figure 1: TimberWolf global routing algorithm.

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.

Figure 1 illustrates the execution of TWGR on a simple circuit. There are 8 rows, 4 nets, and 10 pins. The number on each pin indicates the net it is on. (a) Step 1 builds the approximate Steiner tree for each net. The Steiner points D and G are added for nets 1 and 4, respectively. Step 2 determines the shape of each L-shape switchable segment. Since H is a pad pin, tilted segment GH of net 4 can be routed only by a up L-shaped wire. Segment AD is the only switchable segment. (b) Step 3 assigns the feedthroughs for each net. Only nets 1 and 4 require feedthroughs. Step 4 connects each net. (c) Step 5 flips switchable net segments to optimize their channels. Assume pins A and B have equivalences on the opposite sides of their cells, then AD and EB of net 1 are the switchable net segments. (d) Final routing connections of 4 nets.

Algorithm 1 (TWGR)

1. Approximate Steiner Tree Generation
   for each net **do**
   (a) Build the complete graph G for this net with pins as vertices
   (b) Build the minimum spanning tree of G by Prim's algorithm
   (c) Build the approximate Steiner tree of G
2. Coarse Global Routing
   (a) Build coarse global routing grid for the core
   (b) **repeat** M times
      i. Randomly pick a net and then randomly pick a switchable L shape segment from that net
      ii. Evaluate the change in needed feedthrough number and horizontal channel density of each grid on the path of L-shaped wire when the side of L shape is switched
      iii. Determine the side of L shape for that segment and update needed feedthrough numbers and horizontal channel densities of each grid point which the old and new L-shaped wires go through
   (c) Add in all feedthroughs needed at each grid point
3. Feedthrough Assignment
   for each row **do**
   (a) Assign each crossing segment an available feedthrough in that row
4. Net Connection (Net Graph Generation)
   for each net **do**
   (a) Build a graph G which consists of the complete subgraphs of all pins and feedthroughs in any adjacent pair of rows
   (b) Find the minimum spanning tree of G by Kruskal algorithm
5. Switchable Net Segment Placement Optimization
   repeat N times
   (a) Randomly pick a net and then randomly pick a switchable net segment from the net

(b) Evaluate the change in channel track density when this switchable net segment is flipped from the upper/lower channel to the lower/upper channel

(c) Determine the upper/lower channel this segment should go through and update the track numbers of the upper and lower channels

Throughout this paper, we assume n is the number of nets; n_i is the number of pins on net i; and n_0 is the maximum number of pins in any net. Also, in order to give an analytical measure for runtime and memory requirement of our parallel algorithms, we give a simple measure of those for TWGR and assume r is the number of rows, n_c is the number of cells, r_i is the number of rows net i crosses, C_t is runtime complexity, C_s is the space (memory) complexity, and p is the number of processors.

Based on the above notations, we have the following estimations for each net i. (1) Since the approximate Steiner tree of net i tries to minimize the vertical hops of net i, the average number of feedthroughs needed by net i is about r_i; (2) On the average, a net has n_i/r_i regular pins in each row it crosses; and (3) The approximate Steiner tree has roughly n_i edges

To simplify the notations of analysis, we only measure the dominant portions of the runtime and space complexities of TWGR. All complexity expressions are derived from the implementation of TWGR although it is possible that they are not the best in theory. Since memory is allocated at one time and may be released at another, we assume that the memory complexity means the maximum memory requirement at any time. All the summations are done from 1 through n, if not listed otherwise. If a step executes some computation n times, we say the runtime complexity of this step is n. For the simplicity of writing, the constant weight of each computation of the step is added later when the total is given. Likewise, if a step needs to store m items of some data structure, we say the space complexity of this step is m, and the constant weight of each item will be added later when the maximum memory requirement is given.

The minimum memory requirement is storing information on all those rows, cells, nets, and pins. If we assume the memory for storing each of them is the same size, then the base memory requirement is $r + n_c + n + \sum n_i$. In addition, as TWGR proceeds, additional memory is required which will be analyzed in the following discussion.

In the approximate Steiner tree generation step, the dominant part of both computation and memory is the building of the minimum spanning tree for a complete graph of a net with n_i pins. Its runtime is $\sum n_i^3$, and its memory complexity is n_0^2 to store the cost.

Since the total number of segments of all approximate Steiner trees is $\sum n_i$ and in TWGR

$M = C_m \sum n_i$ for some $C_m > 0$, The coarse global routing step takes $C_m \sum n_i$ time. Compared with that of the approximate Steiner tree generation step, there is no significant memory requirement in this step.

Since in total there are $\sum r_i$ feedthroughs, each row will have an average of $\sum r_i/r$ feedthroughs. In the feedthrough assignment step, each row is divided into C_f feedthrough blocks for some $C_f > 0$. Each block has $\sum r_i/(rC_f)$ feedthroughs. The feedthrough assignment of each block is a quadratic algorithm in its number of feedthroughs. Therefore, the feedthrough assignment step takes $rC_f(\sum r_i)^2/(rC_f)^2 = (\sum r_i)^2/(rC_f)$ time. The memory requirement of feedthrough assignment of each row is a square of the number of feedthroughs in that row. Therefore, the space complexity of the feedthrough assignment step is $(\sum r_i)^2/r^2$.

In the net graph generation step, for each adjacent pair of rows, there are $2n_i/r$ regular pins and $2r_i/r$ feedthroughs. The runtime of building the minimum spanning tree using Kruskal's algorithm for net i is $(n_i + r_i)^2 \log(n_i + r_i)$. Since compared with n_i, r_i is small, the runtime complexity of this step can be written as $\sum n_i^2 \log(n_i)$. The largest net has n_0/r number of pins and feedthroughs in each row, it takes $rn_0^2/r^2 = n_0^2/r$ space to store the cost for the graph.

In TWGR the number of the random switchable row segment selections is $N = C_n n$ for some $C_n > 0$. The total runtime in this step is $C_n n$. In the switchable net segment placement optimization step, regular pins and feedthroughs break each channel into track intervals. Since there are $\sum(n_i + r_i)/r$ pins and feedthroughs in each row, omitting the term r_i, it takes $\sum n_i/r$ space to store that information for each row. Therefore, the space complexity of this step is $\sum n_i$.

In summary, there exist some nonnegative constants c_1, c_2, c_3, c_4, and c_5 such that the runtime complexity of TWGR can be written as

$$C_t = c_1 \sum n_i^3 + c_2 \sum n_i + c_3(\sum r_i)^2/r + c_4 \sum n_i^2 \log(n_i) + c_5 n \tag{1}$$

and there exist nonnegative constants m_1, m_2, m_3, m_4, and m_5 such that the space complexity can be expressed as

$$C_s = m_1(r + n_c + n + \sum n_i) + \max\{m_2 n_0^2, m_3(\sum r_i)^2/r^2, m_4 n_0^2/r, m_5 \sum n_i\} \tag{2}$$

3 Parallelization of TWGR

Starting from a simple parallelization of TWGR, in the following sections, we will propose several approaches for the parallelization of TWGR with better quality and good speedups.

In standard cell VLSI design, a circuit consists of four components: row, cell, pin, and net. A row is 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 into the same partition of their rows. 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.

The task of a global router is to connect the nets. For each net, if we use those pins as the vertex set to form a graph, then the job of a global router is to determine the edge set such that the graph forms a single connected component. Note that the vertex set of the graph may be enlarged because some feedthroughs may be added in. For each net, it is not possible that all of its pins are on cells and rows of the same processor. The problem of partitioning the nets to reduce the pin set intervention with row-wise cell partition and to balance the working load according to computational features of TWGR is one of the problems we will solve in this paper.

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 effect 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 Partitioned Parallel Routing Algorithms

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.

4.1 The Algorithm

Figure 2 shows the scenario after a row-wise pin partition. There are four sub-circuits. If we can run TWGR on those sub-circuits, we have the most possible coarse grain parallelism. Unfortunately, this parallelization will not do all the work. After we run TWGR on each sub-circuit, each sub-net is connected within each partition. We have to complete the connection of each net. For example, in Figure 2, pins A, B, and C belong to the same net and thus should be connected. But after the pin partition and execution of TWGR on each sub-circuit, we can only guarantee the connection of pins B and C. Also note that pin A is not connected with any pin since it is alone on that sub-circuit.

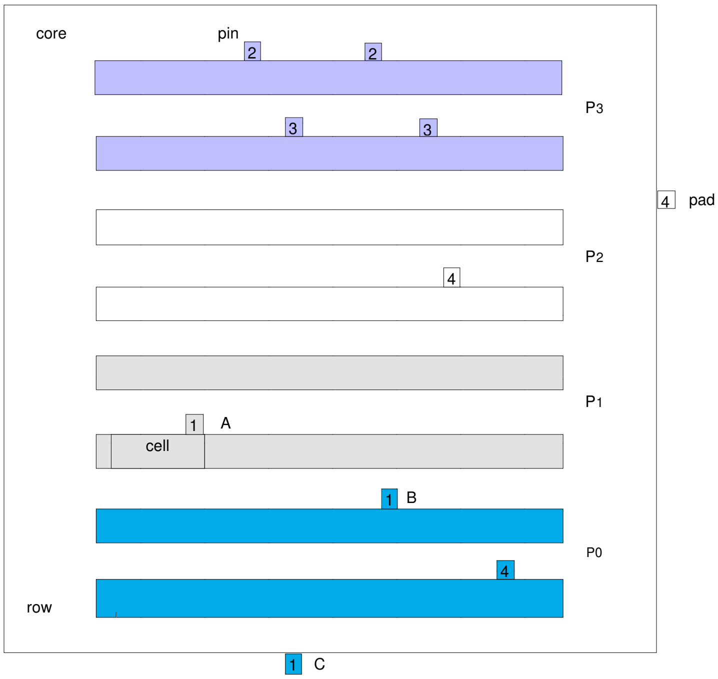

Figure 2: An example of a simple row-wise pin partition. The shading indicates the partition of an object, such as row, cell, net, and pin, indicating which processor it belongs to.

To make sure all the nets are connected, at the boundary of each pair of partitions we add

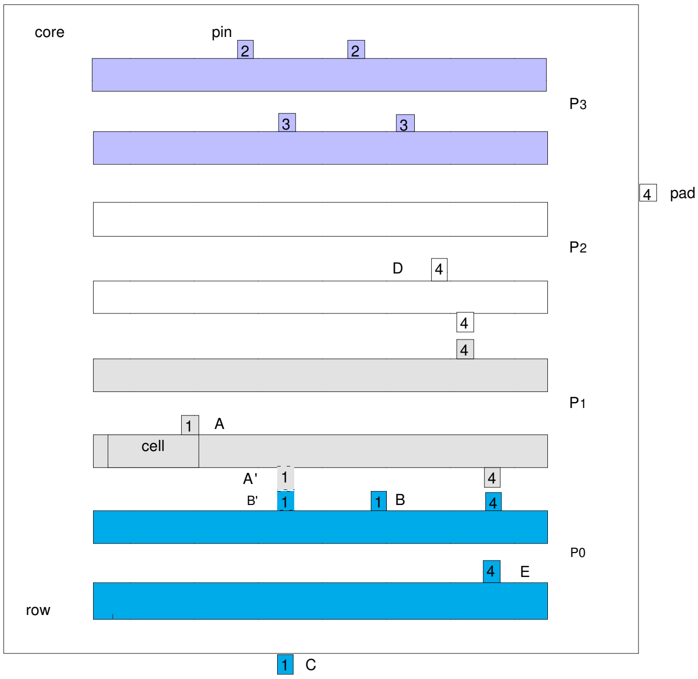

Figure 3: Hook each net by adding in fake pins at partition boundaries

in two fake pins. Those two pins will not move throughout routing. For example, in Figure 3, to connect pins A, B, and C we add two fake pins A' and B'. A' is at the bottom of the upper partition and B' is at the top of the lower partition. 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. After we execute TWGR on each sub-circuit, the pins in the upper partition (pin A) connect with pins in the lower partition (pins B and C) through fake pins A' and B'. Therefore, we have ensured that all the nets are connected after the routing is executed in parallel.

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. 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

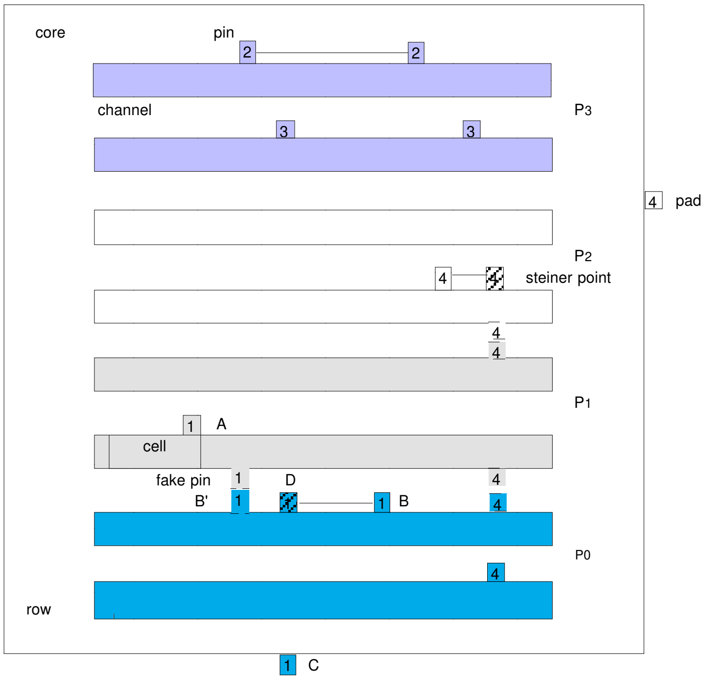

Figure 4: Illustration adding fake pins according to the approximate Steiner tree of each net. There are two partition crossing segments. Two fake pins are added for net 1, and 4 are added for net 4.

become the net segments of the processor which owns its two end points. In Figure 4 shows the circuit partition after the addition of the fake pins.

Although now we can guarantee that all the nets are connected after each processor finishes its TWGR, addition of fake pins may unnecessarily increase the channel densities. For example, in Figure 5, we want to connect pins A and B for net 1. Assume pin A has an equivalence at the bottom of its cell. Then connecting A and B requires no horizontal tracks. But if pin A is moved along with its cell to the same side of fake pins as B, connections AA' and BB' will give two more horizontal tracks. To solve this problem, we do two things. First, for each segment we do not add fake pins at the row boundary of a partition if one of the end pins of the segment is on the boundary already. For example, in Figure 4, since pin D is on the boundary of processor 0 we don't have to add fake pin B'. By doing this, after each processor executes TWGR, if both processor j and $j+1$ have pins of some net then processor j has pins of the net on its upper boundary and processor $j+1$ has pins of the net on its lower boundary. We have to make more connection since, in some cases, there are no fake pin pairs. But we need only connect those pins on the partition boundaries for each net. Those connection wires are required and not switchable net segments, and

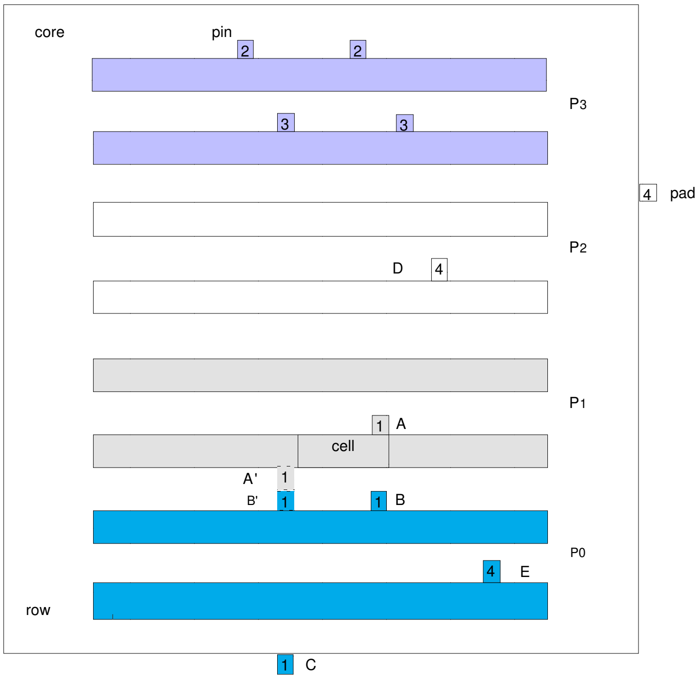

Figure 5: A possible track increase in row-wise pin partition

note that when TWGR optimizes the placements of the switchable net segments, if more accurate information about the tracks in each channel is known then more correct placements of switchable net segments can be made. Therefore, we should make those connections as early as possible. But it cannot be made before feedthroughs are added since before then the pin's positions may change. After each processor has its own sub-circuit, there are $p-1$ shared channels. For $i = 1, \cdots, p-1$, the processors $i-1$ and i share the channel $i * r/p + 1$. When TWGR optimizes the placements of the switchable net segments, since two adjacent processors could assign the same switchable net segments to the shared channel, the channel density information should be synchronized. The frequency of this synchronization is controlled by the number f_t, the channel density information synchronization frequency. Algorithm 2 summarizes the row-wise pin partitioning parallel routing algorithm.

Algorithm 2 (Row-wise Pin Partitioning Parallel Routing Algorithm)

1. **do** net-partition
2. **forall** processors and nets it owns **do** the approximate Steiner tree generation
3. **forall** processors and net segments it owns **do**
   (a) Find the range of processors the segment crosses their boundaries and send those processors the crossing points.
   (b) Receive all the crossing points.

(c) Add one fake pin at each crossing point.

(d) Add one fake pin at own crossing boundary if neither of the end pin of the segment is not on the boundary.

4. **forall** processors **do**

   (a) Execute Steps 2-4 of TWGR

   (b) **for** each net **do**

      i. Find the pin set on upper boundary. If this set is nonempty then send those pins to upper processor.

      ii. Receive pin sets from the lower processor and for each boundary find the pair with the shortest Euler distance from two sets. Make connection between the pair and add this to the channel density information.

   (c) **for** i = 1 to f_t **do**

      i. **repeat** parts (a) - (b) of the switchable net segment placement optimization step of TWGR $N/(f_t p)$ times

      ii. Update the channel track density information for each shared channel

4.2 Analysis

First, let us consider the steps that will be also executed by both TWGR and Algorithm 2. The approximate Steiner tree generation step will be executed in parallel. If the working load is balanced, for each processor it will take $\sum n_i^3/p$ amount of runtime and n_0^2 amount of memory. After circuit is partitioned in Step 3, on each processor, there are r/p rows, n_c/p cells, $\sum n_i/p$ pins, and n_i/p pins for each big net i. There are $\sum n_i/p$ segments in each partition. Therefore, the coarse global routing step takes $\sum n_i/p$ and does not require significant memory. In the feedthrough assignment step, since there are only r/p rows in each partition, the running time of this step is $(\sum r_i)^2/(rC_f p)$; and since there are $\sum r_i/r$ feedthroughs in each row, the space complexity of this step is is $(\sum r_i)^2/r^2$. When building the net graph, the dominant portions of both computation and memory are from the big nets. Since the number of pins of each big net is scaled by a factor of p, this step has the runtime complexity of $\sum n_i^2 \log(n_i)/p^2$; and since the largest net has $n_0/(pr)$ pins and feedthroughs in each row of each partition and crosses r/p rows, the space complexity of this step is $n_0^2/(rp^3)$. The number of the switchable net segments is scaled by a factor of p in the switchable net segment placement optimization step. Although the number of nets may not change, the number of flips needed to reach the channel track stable state should be scaled by a factor of p. Therefore, this step takes n/p amount of time; and since there are $\sum n_i/p$ pins in each partition, we need this amount of memory to store the track information for each pin, hence the space complexity of this step is $\sum n_i/p$.

Next, let us consider the overhead of Algorithm 2. Step 1 partitions the net. Step 3 breaks the segments. Since both of them take a small portion of computation time and memory, we ignore those steps. For each $1 \leq i \leq n$, net i spans across $r_i p/r$ processors. Each of those processors has $n_i r/(r_i p)$ pins and feedthroughs. For each processor pair, the determination of the minimum pair will take time proportional to the square of number of pins on each processor. Therefore, part (b) of Step 4 takes $r^2(\sum n_i^2/r_i^2)/p^2$ time and no significant memory requirement. In part (c) of Step 4 of Algorithm 2, there will be $\sum(n_i + r_i)/r$ regular pins and feedthroughs in each channel. It takes $\sum(n_i + r_i)/r$ to store the track information of each channel. To update this information, each processor needs to send $\sum(n_i + r_i)/r$ amount of information to the processor which shares the channel with it. In a row-wise pin partition, each channel can be shared by only two processors. There are r/p shared channels. Hence, the runtime complexity of this step is $f_t \sum(n_i + r_i)/p$. The space complexity of this step is $\sum(n_i + r_i)/r$ to buffer the message. In a summary, the runtime and space complexities of Algorithm 2 are

$$\begin{aligned}
C'_t = \quad & \tfrac{1}{p}(c_1 \sum n_i^3 + c_2 \sum n_i + c_3(\sum r_i)^2/r + c_4 \sum n_i^2 \log n_i/p^2 + c_5 n \\
& + c_6 r^2/p \sum(n_i^2/r_i^2) + c_7 f_t \sum(n_i + r_i)) \\
C'_s = \quad & m_1(r/p + n_c/p + n + \sum n_i/p) + \\
& \max\{m_2 n_0^2, m_3(\sum r_i)^2/r^2, m_4 n_0^2/(rp^2), m_5 \sum n_i/p, m_6 \sum(n_i + r_i)/r\}
\end{aligned}$$

for some nonnegative numbers c_6, c_7, and m_6. From the above analysis, we can see the almost perfect runtime scalability of Algorithm 2. Although the memory is not perfectly scaled, the majority of the memory consumptions will be scaled by a factor of p.

5 Net-wise Pin Partitioned Parallel Routing Algorithms

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 $(x_j, y_j), j = 1, \cdots, n_i$, are the pin locations. The center of this net is defined as $(\sum_1^{n_i} x_j/n_i, \sum_1^{n_i} y_j/n_i)$. Since nets which stay around the same rows will tend to put tracks into the same channel, there will be more execution intervention 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., $w[i] = \sum_1^{n_i} y_j/n_i$.

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 sit in the row range of each processor. If processor j has the most of pins of net i, then we give the net i a weight of j, i.e., $w[i] = j$.

Random partition. This is an ad hoc partition. We only balance the load. No data locality is taken care. In this partition, we simply call a random number generator i times and give the net i a weight of the returned number, i.e., $w[i] = \text{random}()$.

Exclusive partition. In this partition, all the nets are put into a single processor. The net-wise steps such as the approximate Steiner tree generation step of TWGR will be run sequentially. This partition is used only when it is difficult for a parallel global router to control the quality. For this net partition, we do not have to use the generic partition algorithm.

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 α 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, according to order the sorted weight array, we evenly distribute large nets in a round-robin manner.

Our experiments shows 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 has less than 5 pins. Experiment results will be given in Section 8.

5.1 Synchronization Strategies

A net partition will give a corresponding pin partition. If this pin partition does not change throughout the course of TWGR, then this is called the **net-wise pin partitioned algorithm**. 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. The frequency of that is controlled by the factor f_c, the synchronization frequency for coarse global routing. After coarse global routing, each processor knows about the feedthroughs added in its owned rows. To do feedthrough assignment, each processor 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 frequency of this update is controlled by the number f_t, the channel density information synchronization frequency number. Algorithm 3 is the listing of the above algorithm.

Algorithm 3 (Net-wise Pin Partitioned Parallel Routing Algorithm)

1. **do** net-partition
2. **forall** processors **do** for all nets owned by each processor
   - (a) Execute the approximate Steiner tree generation step of TWGR
   - (b) Coarse Global Routing
     - i. **for** i = 1 to f_c **do**
       - A. **repeat** i - ii) of part(b) of the coarse global routing step of TWRC M/fc times
       - B. Update needed feedthrough and channel density information for each grid point
     - ii. Add in all feedthroughs needed at each grid point in its rows
   - (c) Feedthrough Assignment
     - i. Send out net segments which cross rows owned by other processors
     - ii. Receive all those segments
     - iii. Do the feedthrough assignment
   - (d) Execute the net connection step of TWGR

(e) **for** i = 1 to f_t **do**

   i. **repeat** (a)-(c) of the switchable net segment placement optimization step of TWGR N/f_t times

   ii. Update channel track density information of that channel

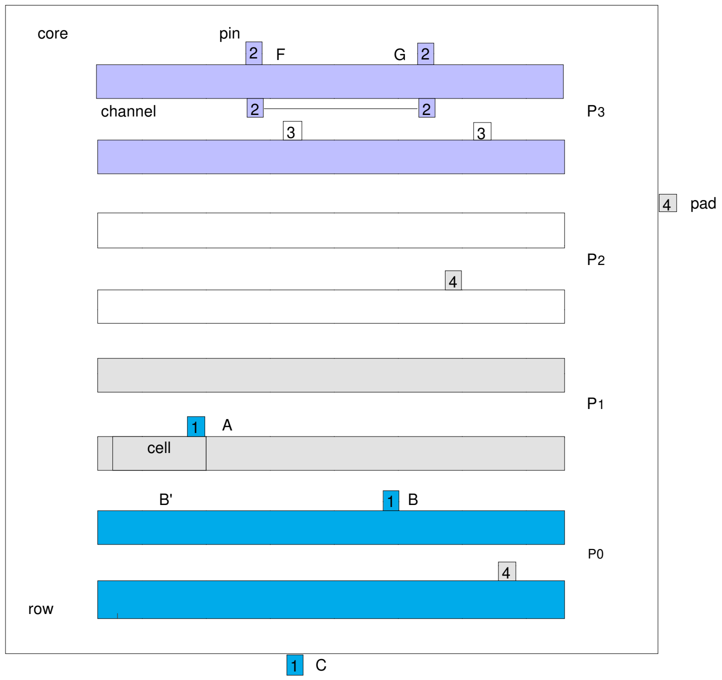

Figure 6: Lack of update track information could make processors make a wrong decision in a net-partitioned parallel routing algorithm

The problem of Algorithm 3 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. For example, Figure 6 is a result of center net partition. Assume net 2 has two pins which both have an equivalence on the bottom of their cells. Then switchable net segment FG could be placed in either the upper channel or the lower channel. Since net 2 and 3 belong to different processors, processor 3 could make a wrong move by placing FG into the lower channel where there is a segment of net 3. This unnecessarily increases the channel density, and may result in a severe loss of quality. Although synchronization in parts (b) and (e) of Step 2 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.

5.2 Analysis

First, let us consider the steps that will be executed by both TWGR and Algorithm 3. The approximate Steiner tree generation step will be executed in parallel. If the working load is balanced, this step takes $\sum n_i^3/p$ amount of runtime and n_0^2 amount of memory. After circuit is partitioned in Step 1, on each processor, there are r/p rows, n_c/p cells, $\sum n_i/p$ pins, and n/p nets. There are $\sum n_i/p$ segments in each partition. Therefore, the coarse global routing step takes $\sum n_i/p$ time and does not require significant memory. In the feedthrough assignment step, since there are only r/p rows in each partition, the running time of this step is $(\sum r_i)^2/(rC_f p)$ and since there are $\sum r_i/r$ feedthroughs in each row, the space complexity of this step is $(\sum r_i)^2/r^2$. Since there are only n/p nets on each partition, the net connection step has the runtime complexity of $\sum n_i^2 \log(n_i)/p$; and the largest net has n_0/r pins and feedthroughs in each row, this step has the space complexity of n_0^2/r. Since the number of nets is scaled by a factor of p in the switchable net segment placement optimization step, this step takes n/p amount of time; and since there are $\sum n_i/p$ pins and feedthroughs in each partition, we need this amount of memory to store the track information for each pin, hence the space complexity of this step is $\sum n_i/p$.

Next, let us consider the overhead of Algorithm 3. Step 1 partitions the net. It takes a small portion of computation time and memory, we ignore this step. There are two synchronizations. The first is in coarse global routing step and the second is in the switchable net segment optimization step. Since the grid number of each row during the coarse global routing is much less than the number of pins and feedthroughs in each row, the runtime and space complexity of this operation will be shadowed by that of the synchronization of the track information during the switchable net segment flip stage of the global router. In the following we will do an analysis for the latter synchronization.

There are $\sum n_i$ segments needed to be sent out and received by each processor in part (c) of Step 2 of Algorithm 3. This needs $\sum n_i$ space to buffer it. In part (e) of Step 2 of Algorithm 3, there will be $\sum(n_i + r_i)/r$ regular pins and feedthroughs in each channel. It takes $\sum(n_i + r_i)/r$ to store the track information of each channel. To update this information, we need to broadcast $\sum(n_i+r_i)/r$ amount of information to all other processors. Assuming the best possible broadcasting strategies among p processors, the runtime complexity of this update is $\log(p)\sum(n_i + r_i)/r$, using all to all broadcast wormhole routing[5], and the space requirement is the message buffer size, which is $\sum(n_i + r_i)/r$.

Therefore the runtime and space complexities of Algorithm 3 is

$$\begin{aligned}
C_t' = \; & \tfrac{1}{p}(c_1 \sum n_i^3 + c_2 \sum n_i + c_3 \sum r_i^2/r + c_4 \sum n_i^2 \log(n_i) + c_5 n) + \\
& c_6 \sum n_i + c_7 f_t \log(p) \sum (n_i + r_i)/r \\
C_s' = \; & \tfrac{1}{p} m_1 (r + n_c + n + \sum n_i) + \\
& \max\{m_2 n_0^2, m_3 (\sum r_i)^2/r^2, m_4 n_0^2/r, m_5 \sum n_i/p, m_6 \sum n_i, m_7 \sum (n_i + r_i)/r\}
\end{aligned}$$

for some c_6, c_7, m_6, and m_7. From the above analysis, we can see that except synchronization overheads Algorithm 3 has an almost perfect runtime scalability. Although the memory is not perfectly scaled, the majority of the memory consumptions will be scaled by a factor of p. But as shown by experimental results in Section 8, when f_t is too large, the synchronization overhead will offset the scalability of Algorithm 3.

6 Hybrid Pin Partitioning Parallel Algorithm

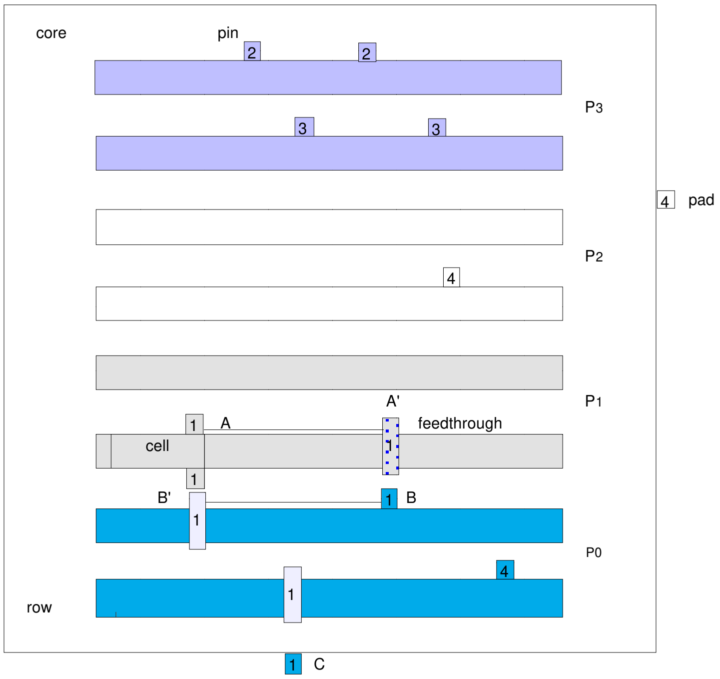

Figure 7: An example of a possible track increase in the row-wise pin partitioning routing parallel algorithm

In this section, we describe an algorithm which combines the benefits of the previous two algorithms. Before describing the algorithm, we illustrate a problem of the row-wise pin partitioning Algorithm 2 in Figure 7. Since net 1 is partitioned into two nets, the net graph will be generated independently on each processor independently and B A' will be connected. Net 1 will be routed as shown in Figure 7. If pin A has an equivalence at the bottom of its cell, then we have a better connection CB', B'A, AA', and A'B. This connection will save the horizontal track BB'. This track is added because the net graph is built on a subnet basis. To further reduce the quality degradation of Algorithm 2, instead of letting each processor build the net graph for the subnets, we let one processor build the net graph for each whole net. After that any segment can cross at most two rows. For every segment we assign it to the processor if its left end pin is on that partition. Then all processors execute the switchable net segment location optimization step of Algorithm 2 in parallel. Algorithm 4 reflects this modification.

Algorithm 4

1. **do** Step 1 - part (d) of Step 2 of the net-wise pin partition parallel routing Algorithm 3
2. **forall** processors **do**
   (a) Send out all the net segments to the processor which owns the left end pins of the segments
   (b) Receive all segments
3. Execute the switchable net segment location optimization step of the row-wise pin partition parallel routing Algorithm 2

As shown in Section 8, Algorithm 4 provides the best quality. Though it could not provide as much gain in runtimes as Algorithm 2, the speedups are still quite good.

6.1 Analysis

First, let us consider the steps that will be executed by both TWGR and Algorithm 4. From the analysis of net-wise pin partition Algorithm 3, the approximate Steiner tree generation step takes $\sum n_i^3/p$ amount of runtime and n_0^2 amount of memory; the coarse global routing step takes $\sum n_i/p$ time and does not require significant memory; the running time of the feedthrough assignment step is $(\sum r_i)^2/(rC_fp)$ and its space complexity is $(\sum r_i)^2/r^2$; the net connection step has the runtime complexity of $\sum n_i^2 \log(n_i)/p$ and the space complexity of n_0^2/r. By the analysis of row-wise pin partitioning Algorithm 2, the switchable net segment placement optimization step takes n/p amount of time and $\sum n_i/p$ amount of space.

Next, let us consider the overhead of Algorithm 4. By the analysis of net-wise pin partitioning Algorithm 3, the feedthrough assignment step takes $\sum n_i$ amount time and $\sum n_i$ amount of space to collect all the information about crossing segments. By the analysis of row-wise pin partitioning

Algorithm 2, the switchable net segment placement optimization step takes takes $f_t \sum(n_i + r_i)/p$ time and $\sum(n_i + r_i)/r$ space to synchronize the track information of each shared channel. The overhead of Step 2 of Algorithm 4 is not significant. Therefore, the complexities of running time and space are

$$\begin{aligned} C_t' = \; & \tfrac{1}{p}(c_1 \sum n_i^3 + c_2 \sum n_i + c_3 \sum r_i^2/r + c_4 \sum n_i^2 \log(n_i) + c_5 n) + \\ & c_6 \sum n_i + c_7 f_t \sum(n_i + r_i)/(rp) \\ C_s' = \; & \tfrac{1}{p} m_1 (r + n_c + n + \sum n_i) + \\ & \max\{m_2 n_0^2, m_3 (\sum r_i)^2/r^2, m_4 n_0^2/r, m_5 \sum n_i/p, m_6 \sum n_i, m_7 \sum(n_i + r_i)/r\} \end{aligned}$$

for some nonnegative number c_6, c_7, m_6, and m_7. From the above analysis, we can see the almost perfect runtime scalability of Algorithm 4. Although the memory is not perfectly scaled, the majority of the memory consumptions will be scaled by a factor of p.

7 Algorithms for Handling Circuits with Large Nets

When the circuits have very large nets, the Steiner tree building can dominate the running time and the memory requirement of TWGR. For example, circuit AVQ.LARGE from MCNC has a net with more than 4000 pins. When TWGR builds the approximate Steiner tree, it needs to create a 2 dimensional array with at least $4,000 \times 4,000$ entries to store the cost data. If each entry takes 4 bytes, then the total memory requirement for this array alone is 64 megabytes! To reduce the excessive runtime and memory requirement, we compute the approximate Steiner trees in parallel for large nets. Since the majority of the nets will have only several pins, it will not be beneficial if we parallelize the Steiner tree building for those small nets. Since the minimum spanning tree building is the core of the approximate Steiner tree building, we compute the minimum spanning trees in parallel for the large nets. To control the minimum spanning of which net tree is computed in parallel, we use a positive number as the threshold. When the pin number of a net is larger than the threshold, we use the parallel minimum spanning tree algorithm to build the minimum spanning tree for this net.

7.1 Analysis

By [5] the running time of computing the minimum spanning tree in parallel $O(n^2/p) + O(n \log(p))$, where n is the number of vertices in G. Also, the memory requirement is $O(n^2/p)$. The parallelization of the approximate Steiner tree can be applied in conjunction with any of global routing algorithms from serial algorithm TWGR to Algorithms 2, 3, and 4. In all the cases it will provide a more balanced load and more memory scalability.

8 Experiment 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.

circuit	number of rows	number of pins	number of cells	number of nets
primary2	28	18407	2907	3029
biomed	36	26947	6417	5742
industry2	64	125555	12142	13419
industry3	64	176584	15059	21940
avq.small	64	82601	21854	22124
avq.large	64	92282	25114	25384

Table 1: Characteristics of test circuits

8.1 Row-wise Pin Partition

We first report the results of Algorithm 2 based on the pin number weight net partition with weight $\alpha = 1.5$ and $f_c = f_t = 5$. Table 2 lists the scaled track results over the serial run. Table 3 lists the scaled area results over the serial run. Figure 8 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%).

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.02	1.00
4	1.05	1.02	1.01	1.01	1.03	1.03
8	1.08	1.05	1.04	1.04	1.06	1.06

Table 2: Scaled track results of row-wise pin partition algorithm

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.00	1.00	1.00	1.00	1.01	1.00
4	1.02	1.01	1.01	1.00	1.01	1.01
8	1.03	1.02	1.01	1.01	1.02	1.02

Table 3: Scaled area results of row-wise pin partition algorithm

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.08	1.02	1.03	1.01	1.04	1.02
4	1.15	1.08	1.06	1.03	1.09	1.05
8	1.19	1.15	1.13	1.07	1.18	1.12

Table 4: Scaled track results of net-wise pin partitioning algorithm

8.2 Net-wise Pin Partition

We next report the performance of the net-wise pin partitioning algorithm. The routing quality is controlled by frequent synchronization but this reduces the runtime performance and is very costly. We report the results on Algorithm 2 based on the pin number weight net partition with weight 1.5 and $f_c = f_t = 5$. Table 4 lists the scaled track results over the serial run. Table 5 lists the scaled area results over the serial run. Figure 9 shows the speedups obtained. It can be

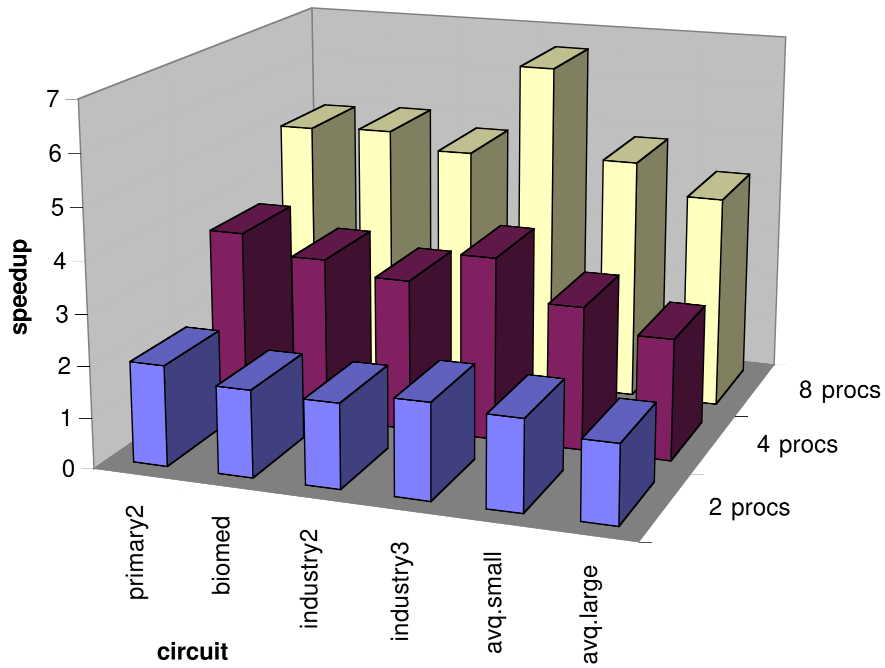

Figure 8: Speedup results of row-wise pin partition algorithm

seen that the net-wise partitioned algorithm causes significant degradation in quality (15% on the average 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.

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.03	1.01	1.01	1.00	1.01	1.01
4	1.05	1.02	1.02	1.01	1.03	1.02
8	1.07	1.05	1.05	1.01	1.06	1.03

Table 5: Scaled area results of the net-wise pin partitioning algorithm

8.3 Hybrid Pin Partition

We finally report the results of Algorithm 4 based on pin number weight net partition with weight 1.5 and $f_c = f_t = 5$. Table 6 lists the scaled track results over the serial run. Table 7 lists the scaled area results over the serial run. Figure 10 shows the speedups obtained. It is clear that the hybrid

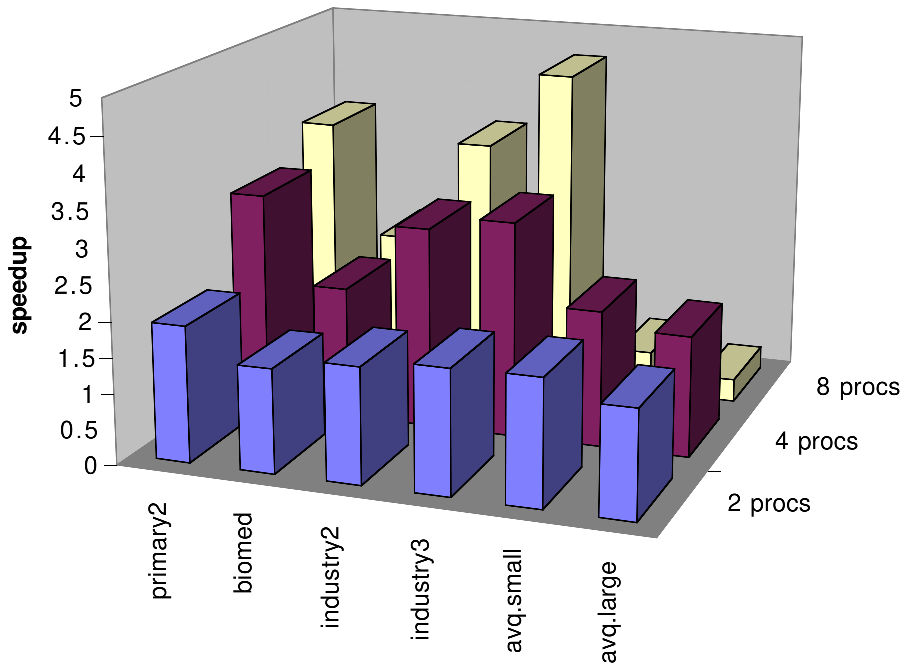

Figure 9: Speedup results of the net wise pin partition algorithm

pin partitioning 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). We have successfully combined Algorithms 2 and 3.

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.02	0.99	1.00	1.00	1.01	1.01
4	1.01	1.01	1.01	1.01	0.99	0.99
8	1.04	1.01	1.01	1.02	1.03	1.02

Table 6: Scaled track results of hybrid pin partition algorithm

8.4 Comparison of Three Parallel Algorithms

Based the results of above three sections, we can conclude the followings. Row-wise pin partitioning algorithm is the best in runtime performance; hybrid pin partition gives the best routing quality; and net-wise pin partition is the worst in both categories.

processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.01	1.00	1.00	1.00	1.00	1.00
4	1.00	1.00	1.01	1.00	1.00	1.00
8	1.01	1.00	1.00	1.00	1.01	1.01

Table 7: Scaled area results of hybrid pin partition algorithm

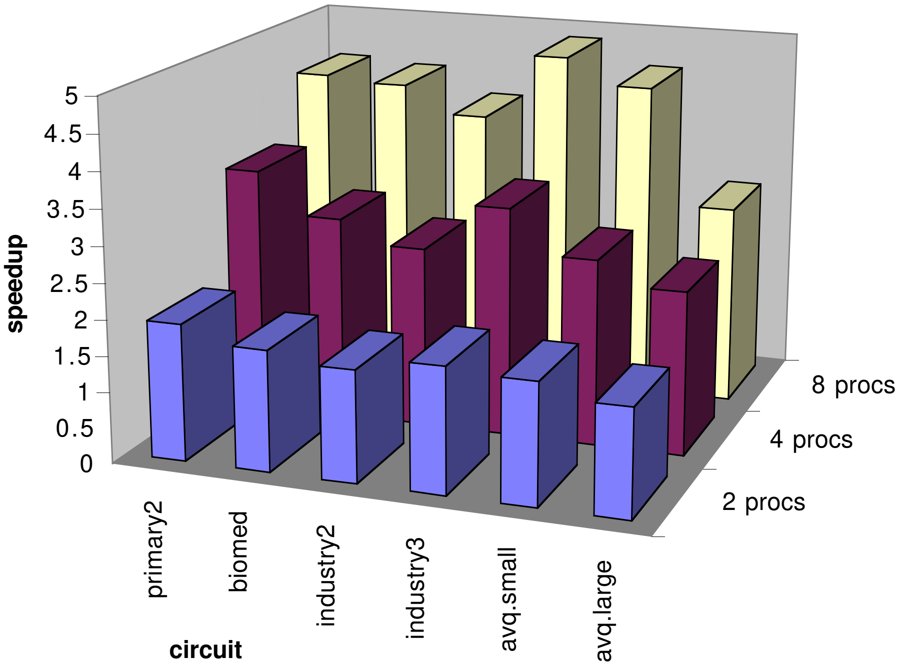

Figure 10: Speedup results of hybrid pin partition algorithm

8.5 Impact of Pin Assignment

In the pin assignment stage of TWGR, rows are processed in increasing order. When a feedthrough is assigned to a segment of some net, the lower end point of that segment will be substituted by the feedthrough. The new segment goes into the pin assignment of the new row. In a parallel algorithm, there are two ways to do the pin assignment. First, each processor statically receives all segments which cross any of its rows and simultaneously does the pin assignment without updating the lower end points of segments. This assignment is called **parallel pin assignment**. Otherwise, the pin assignment will be done in a ripple fashion among processor, called **serial pin assignment**. After one processor is finished it will send all the updated segments to the next processor. Since the row partitioned Algorithms 2 breaks all the segments, there are no segments which cross different processors. Therefore, Algorithms 2 always performs the parallel pin assignments. In this section, we show the experimental results on those two methods for the hybrid pin partitioned Algorithm 4. It is quite intuitive to see that parallel pin assignment is faster but may produce slightly worse quality. Normally, pin assignment is not a computationally significant portion of TWGR. Therefore, even if we do pin assignment in parallel we do not get much runtime speedups. Table 8 lists the scaled track results over the serial run. Figure 11 shows the speedup results.

assign method	processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
	1	1.00	1.00	1.00	1.00	1.00	1.00
serial	2	1.01	0.98	1.00	1.00	1.00	1.00
	4	1.02	1.00	1.00	1.00	1.01	1.00
	8	1.03	1.00	1.01	1.01	1.01	1.00
parallel	2	1.01	0.99	1.00	1.00	1.01	0.99
	4	1.02	1.01	1.00	1.00	1.02	1.01
	8	1.03	1.01	1.01	1.01	1.03	1.04

Table 8: Impact of the pin assignment on total routing tracks of pin hybrid partition algorithm

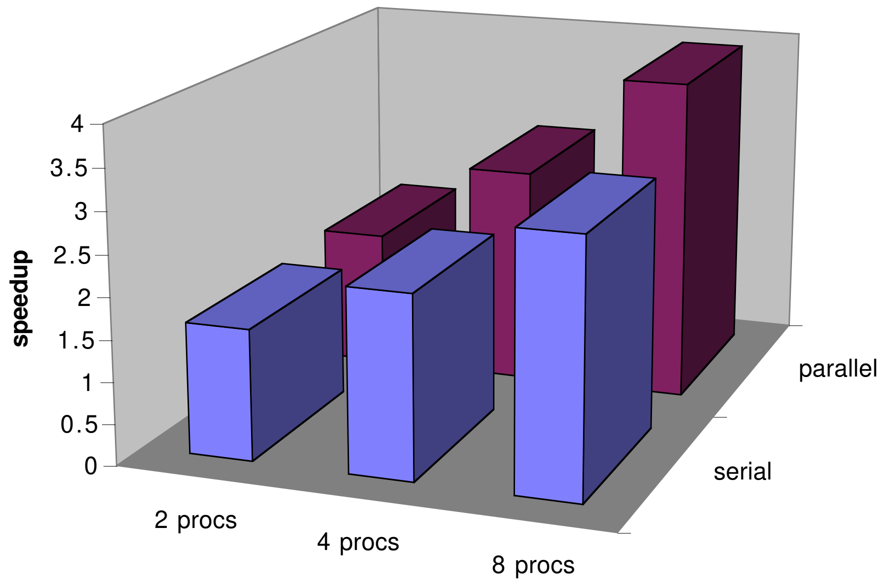

Figure 11: Impact of pin assignment

8.6 Impact of Net Partition with Pin Number Weight

power weight	processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
	1	1.0	1.0	1.0	1.0	1.0	1.0
1.0	2	1.01	0.99	1.00	1.00	1.00	1.01
	4	1.03	1.00	1.01	1.01	1.00	1.04
	8	1.05	1.00	1.02	1.02	1.03	1.08
1.5	2	1.02	0.99	1.00	1.00	1.01	1.01
	4	1.01	1.01	1.01	1.01	0.99	0.99
	8	1.04	1.01	1.01	1.02	1.03	1.02
2.0	2	1.01	0.99	1.00	1.00	1.01	1.01
	4	1.02	1.00	1.01	1.01	0.99	0.99
	8	1.04	1.00	1.01	1.01	1.02	1.01
2.5	2	1.01	0.99	1.00	1.00	1.01	1.02
	4	1.02	1.00	1.00	1.00	1.01	0.98
	8	1.04	1.00	1.01	1.01	1.02	1.00

Table 9: Impact of pin number weight on total routing tracks of pin hybrid partition algorithm

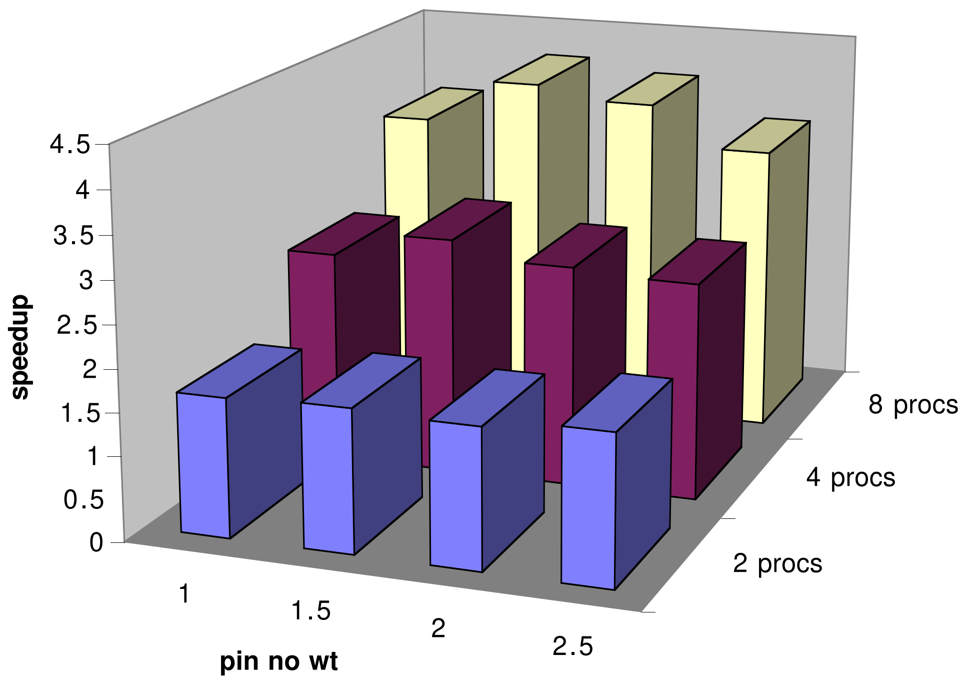

Figure 12: Impact of pin number weight net partition on pin hybrid partition algorithm

As pointed out in Section 5, to speedup the routing of circuits with large nets, the net partition with the pin number weight should be a good choice. This section lists the experimental results for different pin number weights. Since the minimum spanning tree algorithm for a complete graph is a cubic algorithm on the number of pins on the net, we should attempt an increase of power weight up to only 2.5.

For the pin hybrid partition algorithm, Table 9 lists the scaled track results over the serial run. Figure 12 shows the speedup results. For the pin row partition algorithm, results are similar.

From above results, we can see that the best weight value is 1.5, which confirms our theoretical analysis.

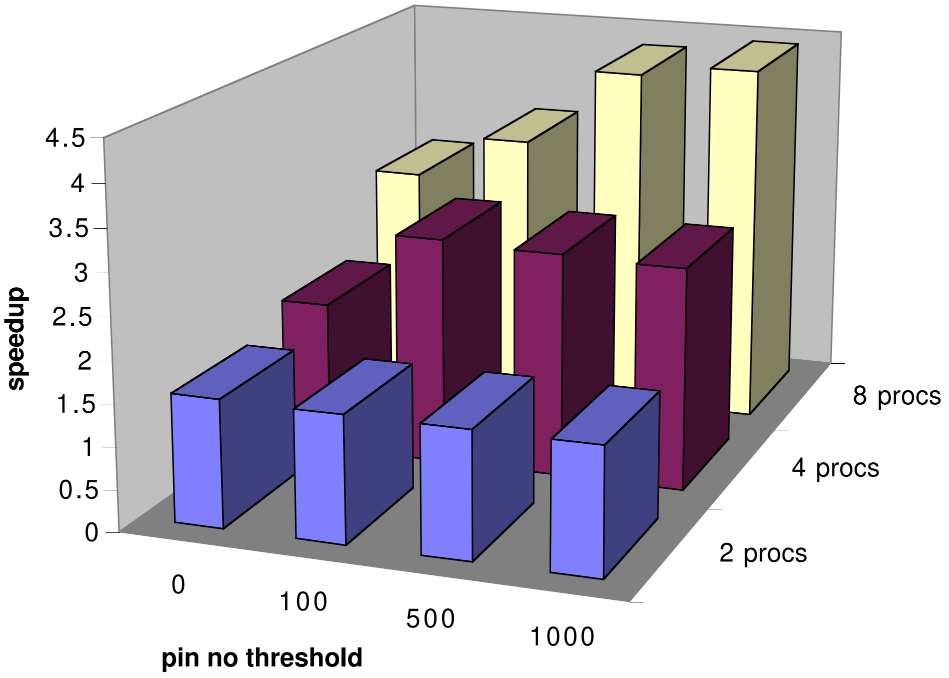

Figure 13: Impact of parallel approximate steiner tree on pin hybrid partition algorithm

8.7 Parallel Approximate Steiner Tree

When the number of pins on some nets of a circuit is too large, the Steiner tree building becomes the dominant partition of the global router. To speedup the global router for such nets, we parallelize the Steiner tree building of large nets. This section lists the results on this parallelization. For the pin hybrid partition algorithm, Figure 13 shows the speedup results.

From the above results, we can see that the bigger the nets we parallelize the more speedups we can get. The problem is that not all the circuits have very big nets. Also, since the parallel algorithm for computing the minimum spanning tree for a graph is not a linear parallel algorithm, even when we compute the approximate Steiner trees in parallel we still can not expect a linear speedup in runtime performance.

track sync no	processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
	1	1.00	1.00	1.00	1.00	1.00	1.00
0	2	1.02	0.99	1.00	1.00	1.00	1.01
	4	1.03	1.01	1.01	1.01	1.04	1.01
	8	1.06	1.02	1.02	1.02	1.06	1.05
10	2	1.01	0.99	1.00	1.00	1.00	1.00
	4	1.02	1.00	1.00	1.00	0.99	1.01
	8	1.03	1.01	1.01	1.01	1.03	1.03
50	2	1.01	0.98	1.00	1.00	0.99	1.01
	4	1.01	0.99	1.00	1.00	0.99	1.01
	8	1.03	1.01	1.01	1.01	1.02	1.02

Table 10: Impact of synchronization on total routing tracks of pin hybrid partition algorithm with $f_c = f_t$

8.8 Impact of Synchronization

Synchronization is very expensive in any parallel application, yet it is important to control the quality of the results. There are two update synchronizations in the routing algorithms. The first is in the stage of coarse global routing and the second is in the stage of the switchable net segment optimization. We list the experimental results for updates in both stages and assume the update frequencies to be the same, i.e., $f_c = f_t$.

For the pin hybrid partitioned algorithm, Table 10 lists the scaled track results over the serial runs. Figure 14 shows the speedup results.

From the above results, we can see that the cost of synchronization increases rapidly when the number of synchronization increases. When the synchronization number is bigger than 10, the gain in quality will compensate the loss in runtime performance. The best synchronization number should be in range of 5-10.

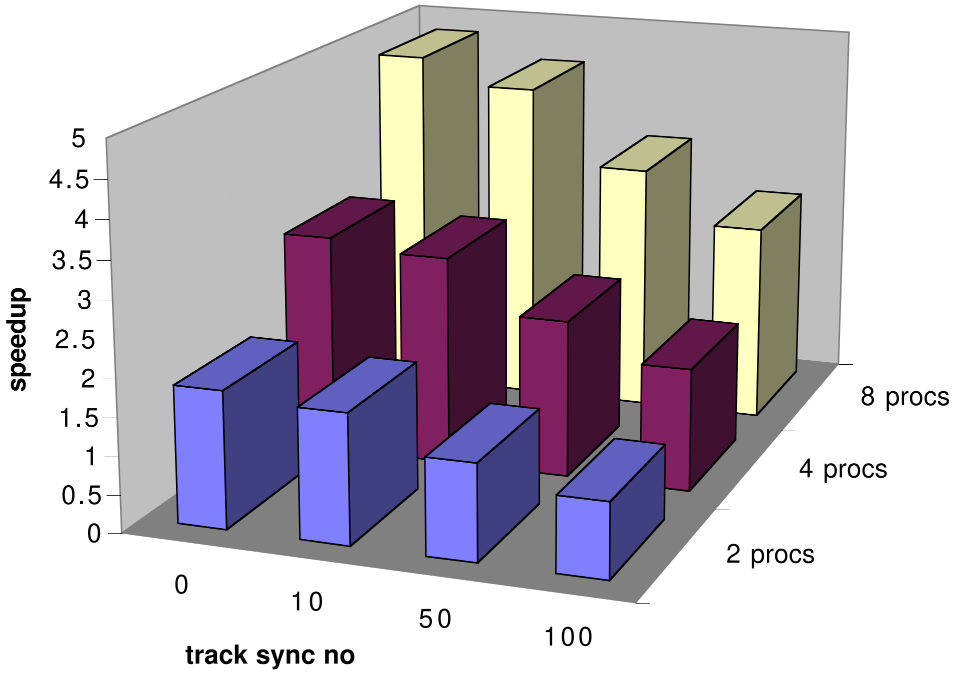

Figure 14: Impact of synchronization on pin hybrid partition algorithm

8.9 Impact of Net Partition

Both Algorithm 2 and Algorithm 4 use some net partitioning approaches as the first step. This section evaluates the impact of those different net partitions on the quality and speedups of those two algorithms.

For the hybrid pin partition Algorithm 4, Table 11 lists the scaled track results over the serial run for various partitioning approaches. Figure 15 shows the speedups obtained. From these results, we conclude that the pin number weight partitioning algorithm is the best.

9 Conclusion

In this paper, we developed three parallel algorithms for global routing based on TimberWolf 6.0. Starting with a simple parallel algorithm with poor routing qualities, we developed two parallel algorithms, pin row-wise partitioned algorithm and pin hybrid algorithm, which have good speedup and quality. From our theoretical analysis and our experiments, we can conclude the best algorithm should be row-wise pin partitioning Algorithm 2 with the pin number weight net partition with synchronization done around 5-10 times. If we want a slightly better quality and are willing to

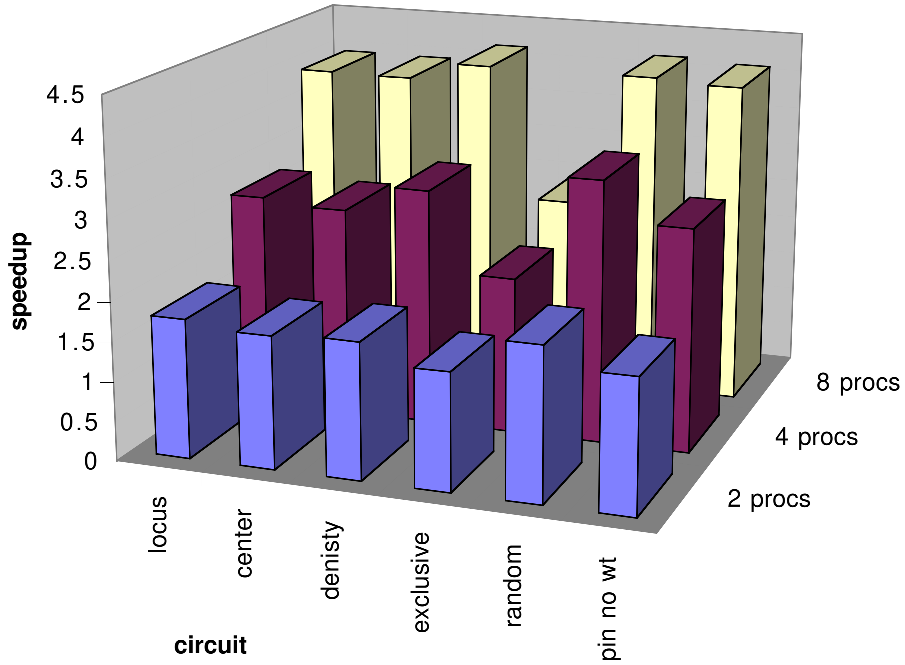

Figure 15: Impact of net partition on pin hybrid partition algorithm

sacrifice some runtime performance, then we should use the pin hybrid partitioning Algorithm 4 instead of pin row partitioning Algorithm 2. Table 12 list the results of the hybrid pin partitioning parallel routing algorithm based on pin number weight net partition with weight 1.5 and $f_c = f_t = 5$. 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. In our experiment, each of 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.

References

[1] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design,* PTR Prentice Hall, Englewood Cliffs, New Jersey 07632, 1994.

[2] R. J. Brouwer and P. Banerjee. "PHIGURE: A Parallel Hierarchical Global Router," *Proc.*

partition	processor no	primary2	biomed	industry2	industry3	avq.small	avq.large
	1	1.00	1.00	1.00	1.00	1.00	1.00
locus	2	1.01	0.98	1.01	1.00	1.00	0.99
	4	1.02	1.00	1.01	1.01	1.03	1.00
	8	1.04	1.01	1.02	1.01	1.03	1.02
center	2	1.01	0.99	1.00	1.00	1.01	0.99
	4	1.02	1.01	1.00	1.00	1.02	1.01
	8	1.03	1.01	1.01	1.01	1.03	1.04
density	2	1.01	0.98	1.00	1.00	0.99	1.02
	4	1.02	0.99	1.01	1.01	1.02	1.03
	8	1.04	1.00	1.01	1.02	1.03	1.11
exclusive	2	1.01	0.99	1.00	1.00	1.01	0.98
	4	1.02	1.00	1.01	1.00	1.00	0.99
	8	1.04	1.01	1.01	1.01	1.02	1.01
random	2	1.01	0.99	1.00	1.01	1.01	0.99
	4	1.03	0.99	1.01	1.01	1.00	1.04
	8	1.05	1.00	1.01	1.02	1.02	1.12
pin no wt	2	1.02	0.99	1.00	1.00	1.01	1.01
	4	1.01	1.01	1.01	1.01	0.99	0.99
	8	1.04	1.01	1.01	1.02	1.03	1.02

Table 11: Impact of net partitions on total routing tracks of pin hybrid partition algorithm

27th Design Automation Conf., pp. 360–364, June 1990.

[3] R. J. Brouwer and P. Banerjee. "PARAGRAPH: A Parallel Algorithm for Simultaneous Placement and Routing Using Hierarchy," *Proc. European Design Automation Conf. (EDAC-92),* pp. 328–332, Mar. 1992.

[4] M. Burstein and R. Pelavin. "Hiearchical Wire Routing," *IEEE Trans. Computer-aided Design Integrated Circuits Systems,* vol. CAD-2, pp. 223–234, Oct. 1983.

[5] V. Kumar, *Introduction to Parallel Computing,* Benjamin Cummings Press, 1994.

[6] C. Lee. "An Algorithm for Path Connections and its Applications", *IRE Trans. Electronic Computers,* vol. VEC-10, pp. 346–365, Sept. 1961.

platform	proc no	results	primary2	biomed	industry2	industry3	avq.small	avq.large
Sun SparcCenter 1000MP	1	Track No	567	1,192	1,475	2,555	1,358	1,687
		Area (μm^2)	25,666,283	51,068,160	113,497,800	674,215,300	83,633,285	103,529,700
		Time (seconds)	65	162	446	845	1693	1539
	8	Track No (scaled)	1.04	1.01	1.01	1.02	1.03	1.02
		Area (scaled)	1.01	1.00	1.00	1.00	1.01	1.01
		Time (seconds)	15	38	115	175	379	402
		Speedup	4.33	4.26	3.88	4.83	4.47	3.83
Intel Paragon DMP	1	Time (seconds)	124	674	98,497	timeout	237,137	timeout
	4	Track No (scaled)	1.01	1.00	1.01	1.01	0.99	1.00
		Area (scaled)	1.00	1.00	1.00	1.00	1.00	1.00
		Time (seconds)	69	118	5,075	7,768	12,162	12,060
		Speedup	1.78	5.7	19.4	2.27*	19.5	4.04*
	16	Track No (scaled)	1.06	1.01	1.03	1.02	1.05	1.04
		Area (scaled)	1.02	1.00	1.01	1.00	1.01	1.01
		Time (seconds)	31	61	1,893	3,409	2,152	2,981
		Speedup	4.00	9.36	52.03	5.19*	110.21	16.37*

Table 12: Results of the hybrid pin partitioning parallel global routing algorithm on different platforms.

[7] K. Lee and C. Sechen, "A New Global Router for Row-Based Layout", *Proc. of ICCAD-88,* pp. 180–183, Nov. 1988.

[8] M. Martonosi and A. Gupta. "Tradeoffs in Message Passing and Shared Memory Implementations of a Standard Cell Router," *Proc. Int. Conf. Parallel Processing (ICPP89),* pp. III:88–96, Aug. 1989.

[9] E. E. Moore. "The Shortest Path through a Maze. *Annals of the Computation Laboratory of Harvard University,* vol. 30, pp. 285–292, 1959.

[10] Message-Passing Interface Forum, "Document for a Standard Message-passing Interface," University of Tennessee, Knoxville, TN, Tech. Rep. CS-93-214, 1993.

[11] R. Nair, S. J. Hong, S. Liles, and R. Villani. "Global Wiring on a Wire Routing machine," *Proc. 19th Design Automation Conf.,* pp. 224–231, June 1982.

[12] A. Ng, P. Raghavan, and C. Thompson. "Experimental Results for a Linear Program Global Router," *Computers and Artificial intelligence,* 1987.

[13] O. A. Olukotun and T. N. Mudge. "A Preliminary Investigation into Parallel Routing on a Hypercube," *Proc. Design Automation Conf.,* pp. 814–820, June 1987.

[14] P. Raghavan, and C. Thompson. "Provably Good Routing in Graphs: Regular Array," *Proc. 17th ACM Symposium on Theory of Computing,* 1985.

[15] J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells" *Proc. of 25th DAC,* pp. 189–195, June 1988

[16] J. Rose, "Parallel Global Routing for Standard Cells," *IEEE Trans. Computer-aided Design Integrated Circuits Systems,* pp. 1085–1095, Oct. 1990.