PARALLEL ALGORITHMS FOR VLSI CIRCUIT EXTRACTION

Krishna P. Belkhale and Prithviraj Banerjee

Acknowledgement: This research was supported in part by the National Science Foundation Award under Grant NSF MIP 87-14646, and in part by an equipment grant NSF CCR 87-05240.

K. P. Belkhale is with the Department of Computer Science and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

P. Banerjee is with the Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

ABSTRACT

Parallel processing has recently become an extremely attractive and cost effective way of achieving orders of magnitude performance improvements in VLSI CAD applications. In this paper, we propose parallel algorithms to speed up the VLSI circuit extraction task. Given a VLSI layout as input, the problem of circuit extraction consists of determining the circuit connectivity and estimating the various electrical parameters such as the resistances of lines, capacitances of nodes, and dimensions of devices. Circuit extraction is a computationally intensive problem. The basic approach used in the parallel algorithms is the partitioning of a circuit into smaller regions, assigning each region to a processor and having the processors cooperate in performing the extraction procedures. We present a number of partitioning strategies that could be used. We have implemented the parallel algorithms on an Intel iPSC2 hypercube and an Encore 510 multimax shared memory multiprocessor.

ADDRESS FOR CORRESPONDENCE

P. Banerjee
Computer Systems Group
Coordinated Science Laboratory
University of Illinois
1101 W. Springfield Av.
Urbana, IL-61801
(217) 244-6564

LIST OF INDEX TERMS

Parallel Processing, Circuit Extraction, Load Balancing, Hypercube Multiprocessors, Shared memory multiprocessors.

LIST OF FIGURE CAPTIONS

LIST OF TABLE CAPTIONS

1. INTRODUCTION

As future VLSI circuits become more complex, the computational requirements of performing various CAD tasks such as simulation, design-rule checking, circuit extraction, cell placement and wire routing will increase tremendously. There will be an increasing need for more accuracy and capabilities from these CAD tools which would involve substantially more computations. Numerous special-purpose hardware accelerators have therefore been proposed to speed up a wide variety of CAD tasks including logic simulation, wire-routing, design rule checking, and module placement [1, 2]. However, this approach of designing a hardware accelerator for a specific task and algorithm may not be very cost effective since the technology for which the CAD tool is being developed may change in the time frame required to implement a hardware accelerator.

General-purpose multiprocessors can also offer the tremendous speed improvements that are needed to solve the complex problems of the future. They are also very versatile as a single multiprocessor can be used to speed up a number of applications. However, these machines will not be able to exploit the parallelism available in the problems unless new algorithms are developed that are well suited to a multiprocessor environment. Recently, researchers have started to investigate parallel algorithms for various CAD applications on commercially available general-purpose multiprocessors [3]. Example applications include logic simulation [4], switch level simulation [5], circuit simulation [6], testing and fault-simulation [7], cell placement [8, 9], wire routing [10], design rule checking [11-13], and circuit extraction [14-16]. In this paper, we will describe parallel algorithms for one particular CAD application, namely, VLSI circuit extraction. This work is the culmination of the work presented in [15] and [16].

Given a set of rectangles in different mask levels for a VLSI layout as the input, a circuit extractor determines the circuit connectivity and estimates various electrical parameters such as the resistances of lines, capacitances of nodes, and dimensions of devices. There are two phases in circuit extraction. The first phase is the identification of devices, determination of electrically connected regions (called nets). This phase is called *netlist extraction*. The second phase involves the estimation of the electrical parameters of nets and devices. This phase is called *parameter extraction.* The output of an extractor can be fed into a circuit, switch or logic level simulator.

Circuit extraction is a computationally intensive problem. A number of circuit extractors of varying accuracy and speed exist [17-22]. With the increasing complexity of VLSI layouts, there is a need for researching faster algorithms for performing circuit extraction.

There are two disjoint, but orthogonal, approaches to speed up circuit extraction. The first approach is to use the hierarchical information available in the circuit. VLSI circuits are typically designed in an hierarchical fashion in terms of cells. Each cell consists of some subcells and base level geometry. Hierarchical extractors exploit the repetition information in this representation by extracting a cell that is used many times only once and appropriately modifying it wherever it is used [20-22]. The difference in analysis time obtained by using hierarchical methods can vary from gaining a factor of several hundred on highly regular designs to losing a factor of 5 on designs that involve not much repetition. The main problem in hierarchical extraction, which is responsible for the overhead, is that the properties of a cell will change if it overlaps another cell, or some base level geometry. Some hierarchical extractors such as [23] forbid any overlapping of cells. However, this is inconvenient and unnecessarily restrictive. Other extractors like [22, 24] transform the circuit with overlapping cells to a circuit in which there are no overlapping cells. The transformation usually involves instantiating a subcell, that overlaps another subcell or base level geometry, at the higher level. This process is repeated until all the subcells of a cell do not overlap each other or the base level geometry. In the worst case, the whole circuit might be converted into a single cell.

Another approach to speed up the task of circuit extraction is to use multiprocessors [14]. A parallel circuit extractor basically divides the circuit into smaller parts and performs various tasks in circuit extraction on each of the parts in parallel, results of which are combined to complete the extraction of the whole circuit. Similar divide and conquer techniques have been used in designing extractors on a uniprocessor which make more efficient use of the memory [25]. The issues in parallel circuit extraction, therefore, are to design efficient algorithms that can be used in the partitioning and combining steps.

In this paper, we discuss parallel algorithms for circuit extraction that operate on a flattened description of a Manhattan VLSI layout. Despite the existence of good hierarchical extractors, often an actual chip design is subjected to a complete flattened extraction and netlist comparison before sending for fabrication. Thus there is a need for speeding up flat extraction. In the future, we will work on a parallel algorithm which will use the hierarchical information as well.

The organization of this paper is as follows. In Section 2, we discuss the different partitioning strategies used by our parallel extractor, *PACE*. In Section 3, we will describe the netlist extraction phase in *PACE*. In Section 4, we will describe the parameter extraction phase in *PACE*. In Section 5, we analyze the time complexity of the extractor. In Section 6, we describe the implementation and present results on an Intel iPSC2 hypercube and an

Encore 510 multimax shared memory multiprocessor.

2. CIRCUIT PARTITIONING AND MAPPING

2.1. Circuit Partitioning

There are four partitioning strategies implemented in the *PACE* algorithm. These are classified as area based and point based methods. We discuss these strategies assuming there are 2^d processors,

In the area based methods, partitioning is done based on the area. There are two strategies possible. The first strategy is called slice partitioning. The second strategy is called rectangular partitioning. In the first case, the circuit is divided into slices of equal area along one dimension. The dimension is chosen to be the one along which the circuit has larger length, in order to minimize the perimeter of each slice which determines the communication complexity. In the second case, the circuit is divided into 2^k slices in the x-dimension and 2^{d-k} slices in the y dimension. Thus the circuit is divided into rectangular regions. The value of k is determined so as to minimize the perimeter of the rectangular regions. The value of k is given as the nearest integer less than

$$\frac{(log_2 \, aspect + d + 1)}{2}$$

Here *aspect* is the ratio of the length of the circuit along x axis to the length along y axis. If the value of k in the above expression comes out less than 0 or greater than d it is changed to 0 and d respectively. The two area based partitioning strategies are shown in Fig. 1.

In the point based methods, we represent a rectangle by a point in two dimensions, for example the center of the rectangle. The objective then is to divide the domain into regions so that each region has approximately the same number of points. There are two partitioning strategies possible, namely slice and rectangular partitioning. In the first case, the domain is divided into slices in one dimension such that each slice has nearly the same number of points. Again, the direction of division is chosen as before to be along the longer dimension. In the second case, the domain is first divided in the x direction into 2^k slices with the same number of points, and each of the slices is in turn divided in the y direction into 2^{d-k} regions each carrying nearly the same number of points. The value of k is determined as before. The two point based strategies are shown in Fig. 2.

Let $p = G(i,j)$ be any processor. For both area and point based methods, we define $R(p)$ to be the rectangular region $(x_{min}, y_{min}, x_{max}, y_{max})$ assigned to processor. More over we assume if $i=0$ or $j=0$ then $x_{min} = -\infty$ and $y_{min} = -\infty$

respectively. Similarly, if $i=2^k-1$ or $j=2^{d-k}-1$ we assume that $x_{\max}=+\infty$ and $y_{\max}=+\infty$ respectively. Also, we define $R'(p)$ to be the region obtained by expanding $R(p)$ by unit distance in all directions, i.e the region $(x_{\min}-1, y_{\min}-1, x_{\max}+1, y_{\max}+1)$. This concept of extended region is also used in parallel Design Rule Checking, where the regions are expanded by the maximum design rule interaction distance [11].

2.2. Mapping and Implementation on a Hypercube Multiprocessor

We now discuss how the circuit partitioning strategies are implemented on a hypercube multiprocessor. A hypercube of dimension d consists of 2^d processors that are connected by a binary d-cube interconnection. The processors are consecutively numbered from 0 to 2^d-1. Each processor is directly connected to d other processors whose binary tags differ from its own by exactly one bit.

In both the area and point based methods, the slice partitioning is a special case of rectangular partitioning. Hence, we discuss the implementation only for rectangular partitioning. The region $A(i,j)$ is assigned to the processor

$$G(i,j)=B(i)*2^{d-k}+B(j)$$

where $B(i)$ is the binary reflected gray code for i. A similar region assignment has been described in [26]. The region assignment is illustrated in Fig. 1 and Fig. 2. The advantage of this assignment is that processors $G(i,j)$ and $G(i+1,j)$ (similarly, processors $G(i,j)$ and $G(i,j+1)$) are physically adjacent, i.e., connected by direct links in the hypercube. Also, processors $G(i,j)$ and $G(i+2^l,j)$, where $l \geq 1$, are within distance two of each other (i.e., they are connected via only one intermediate processor). A similar result holds for the y-dimension. This region assignment results in processors communicating with each other in the *PACE* algorithm to also be physically near each other. Since a lot of communication occurs between processors that are assigned regions that are physically near each other, this partitioning will result in more localized messages and hence improved performance.

Implementing the area based methods is relatively easy. The host reads the input and sends rectangles in batches to processor $G(0,0)$. These rectangles are then distributed to the processors using a tree with $G(0,0)$ as the root. A processor retains all those rectangles that intersect or abut its assigned region. We omit the details of this approach for the sake of brevity.

Implementing the point based methods is an interesting problem. The partitioning could be done by the host or by a node processor after the whole circuit has been read. However, there are two problems with this approach.

The first problem is that the processor may take a lot of time in doing the partitioning. The second problem is that there may not be enough memory at a processor to read in the whole circuit. In *PACE* this problem is solved by storing the points as a distributed data structure, based on the partitioning strategy. The host reads the circuit in batches of K rectangles, and issues an add instruction to the distributed data structure for every batch. Addition or deletion of points from the data structure involves restructuring the partitions so that the resulting regions have the same number of points. We have obtained an efficient algorithm for this problem for the case of a general recursive partition. This is described in [27]. The main idea in the algorithm is to recursively decompose an addition/deletion operation on a region R into addition/deletion operations on its subregions, at the immediate lower level, so that the sub regions are balanced. In *PACE* we have applied this algorithm for the special case of slice and rectangular partitioning. In this case, an addition of w points followed by the restructuring can be done in time $O(dt_\alpha + wdt_\beta + w\log w)$, where t_α, t_β are the start up time and per byte time for communication respectively. The advantage of this approach is that the load balancing and reading of the circuit can be done in parallel. Since the former is experimentally determined to be faster, the load balancing comes at no extra cost. Once all the rectangles have been read by the host, the nodes go through some post processing steps so that each of them gets all the rectangles that overlap or abut its assigned region. This step is required as rectangles were considered as points during the partitioning step. This step can be done in d message passing stages per processor. We again omit the details of this method for the sake of brevity.

2.3. Mapping and Implementation on a Shared Memory Multiprocessor

The mapping of regions to processors is not a crucial issue here. Also, an approximate point based partitioning strategy can be implemented in a straightforward fashion, as illustrated in Fig. 3. The circuit region is divided into different bins in the x dimension. The number of points that fall in each bin is determined and stored in an array. The entries in the array can then be used to determine the partitioning lines along the x dimension. The process can then be repeated in each of the resulting subregions, by the appropriate processors, to determine the partitioning lines along the y dimension.

3. PARALLEL NETLIST EXTRACTION

A VLSI layout is described as a set of Manhattan rectangles in different mask levels. The netlist extraction phase involves determination of devices and determination of connected regions called nets. Devices are obtained by boolean operations on the different masks in the circuit as specified by a technology file. The nets are then obtained by the connectivity information in the technology file, which specifies the layers that connect electrically on overlap.

3.1. Outline of Netlist Extraction in *PACE*

In both the methods of partitioning, each processor p gets all those rectangles that touch or overlap $R(p)$. Before starting the net list extraction, these rectangles are cut so that the processor retains only the portion within $R'(p)$ (the expanded version of $R(p)$). It should be noted that the regions $R'(p)$ for different processors p overlap. Hence, the processors will have some overlapping information of the circuit at the borders of its region. This is crucial and is used later in the algorithm. We now present the steps in the netlist extraction phase of *PACE*. We describe the steps at each processor p.

P1: Boolean mask manipulations are performed locally on the different layers to derive new layers as specified in the technology file. The new layers include layers for the different devices. For example in CMOS technology, N type transistors are obtained by intersecting the poly, diffusion and pwell layers. The P type transistors are obtained by intersecting poly, diffusion and complement of pwell layers. A new diffusion layer is obtained by intersecting the old layer with the complement of poly. This step is illustrated in Fig. 4, for CMOS technology.

P2: The rectangles in the device layers are grouped into maximal connected groups, which form the local devices. All rectangles, other than those in device layers, are grouped into electrically connected sets which form the local nets. The local nets, and devices are collectively referred to as LCS (locally connected set) sets. These are labeled so that they are unique across processors and so that the processor number is a part of the label (the processor's address bits can be the d least significant bits of the label).

P3: For each layer and boundary of $R(p)$ a list of border segments is created. These segments are formed by the intersection of rectangles of the layer with the boundary. For example, consider the set of rectangles in Fig. 5, where a rectangle is shown as a four tuple $(\min x, \min y, \max x, \max y)$. The border segments at processor 0 for the top y border (y=17) are (4,6), (5,7) and (12,15). With each border segment, we have an associated LCS set that it belongs to. If a border segment is from a device layer then it is associated with the device it connects to. Similarly, if a border segment is from a non device layer, then it is associated with the net it belongs to. All the rectangles in all layers are then cut so that only the portion of the rectangles within $R(p)$ is retained. The rectangles of zero area formed in this step are discarded. Steps P1-P3 are done in a single scan line algorithm.

P4: For each layer and boundary, the border segments are converted to maximal border segments by combining overlapping or abutting border segments. This step is illustrated in Fig. 5. Thus, for example, the maximal border segments for top y border ($y=17$) at processor 0 are (4,7) and (12,15). This is because the border segments (4,6) and (5,7) overlap giving rise to the segment (4,7). A single list is then constructed for each boundary by combining the maximal border segments for all the layers at the boundary. This list is then sorted lexicographically, first on the starting points of the segments, and second on the layer number. Each of the entries in the list points to the LCS set it belongs.

For area based method of partitioning, the list constructed by a processor at a border will correspond exactly with the list constructed by the processor sharing the border. For point based method, this observation is only true for the y borders, i.e for those that partition in the y direction. For example, it can be seen from Fig. 5 that the maximal border segments at processor 0 for the top y border ($y=17$) are exactly identical to the one at processor 1 for that border. However, the maximal border segments at processor 0 for the right x border ($x=14$) do not correspond with the one at processor 2 for the left x border.

P5: Let $R(p)=(x_{\min},y_{\min},x_{\max},y_{\max})$. In case of point based partitioning, for each of the x borders the maximal border segments (a,b) that satisfy $a \le y_{\min}$ are deleted. The other segments are left in the original order. For example in Fig. 5, the segment (16,21) is deleted from the right x border ($x=14$) list of processor 1. Similarly, the segment (13,15) is deleted from the left x border ($x=14$) list of processor 3.

Let $leftlist(i)$ ($rightlist(i)$) denote the list obtained by combining in order the maximal border segments for left (right) x border at all processors $G(i,m)$, $0 \le m < 2^{d-k}$. Referring to Fig. 5, $rightlist(0)$ = {(4,6),(12,15),(16,18),(23,24)} (note (16,21) has been deleted in step P5). Similarly, $leftlist(1)$ = {(4,6),(12,15),(16,21),(23,24)}. Also, let $L(i)$ ($R(i)$) denote the sorted list of maximal border segments of the whole circuit for all layers at the left (right) x border that separates slice i and slice $i-1$ (slice $i+1$). Thus referring to Fig. 5, $L(1)=R(0)$ = {(4,6),(12,15),(16,21),(23,24)}.

It can be easily proved that the border segments in $leftlist(i)$ ($rightlist(i)$) are in one to one correspondence with the border segments in $L(i)$ ($R(i)$), though a segment in the second list may be represented in the first by a smaller segment which starts at the same position. For example referring to Fig. 5, $rightlist(0)$ and $R(0)$ are in one to one correspondence. The segment (16,21) in the second list is represented in the former list by a smaller segment (16,18). Since, $L(i)=R(i-1)$, $rightlist(i-1)$ and $leftlist(i)$ will be in exact correspondence for $0<i<2^k$. This observation will be used later.

P6: The lists constructed in P4, P5, for each of the borders, and the LCS sets that touch a border are passed to a Merge algorithm. The Merge algorithm determines for each of the LCS sets a unique label which identifies its global connected component (GCS). It also determines the number of LCS sets that are part of the GCS set. The unique label of a GCS set is chosen by the Merge algorithm from the labels assigned to its LCS components in step P2. The details of the Merge algorithm is described in the next sub section. The processor that has the LCS set with the unique label is said to "own" the GCS set. All LCS sets at a processor belonging to the same GCS set are combined to form a single LCS set. This is useful for the subsequent steps.

The local nets, devices for a four processor CMOS example is shown in Fig. 6. Their final labels are shown in brackets in a table.

At the end of step P6, we have a list of local nets and devices at each of the processors. Also, for each of the local nets and devices, we have a label denoting its global component, and the number of LCS sets comprising the global component. This information will be utilized in the parameter extraction phase. It should be noted that at this point we already have the structural information about the circuit. If only the netlist is desired we could output the information at this stage.

3.2. Merge Problem

A very important part of our parallel algorithm is the divide and conquer paradigm, which involves partitioning the problem, solving on the partitions and then merging the results. The inputs to the Merge problem, as described before, consists of:

(1) The LCS sets at the processor that touch the border. These are specified by their unique labels.

(2) For each of the borders, a list of border segments. Each border segment is represented simply as a pointer to its LCS set.

The Merge problem involves determining for each of the *LCS* sets, its *GCS* set. A *GCS* set is specified by a unique label, and all its component *LCS* sets must get the same label for their *GCS* set label. Also, the Merge algorithm must return for each of the *LCS* sets, the number of *LCS* sets that comprise its *GCS* set.

3.3. Merge Algorithm on a Hypercube Multiprocessor

We now present a simple algorithm for the Merge problem on a hypercube multiprocessor. Other interesting algorithms for the problem are given in [28]. These algorithms try to exploit the locality of connected sets in order to minimize the communicational requirements. The simple merge algorithm has d stages of message passing. Each stage involves sending and receiving a message at each of the processors followed by a computation step. The basic idea in this algorithm is that at each stage a processor doubles its area of knowledge in the circuit. Initially, a processor knows only about its region. After d stages a processor has knowledge about what it needs to know in the entire circuit.

The region of information of a processor in a certain stage is a rectangular sub region of the circuit. At any stage a processor maintains the following information.

(1) List of connected sets in the information region that touch a border. These are called *CCOMP* sets. The *CCOMP* sets are specified by labels.

(2) List of border segments for the borders of the information region. Each border segment points to the *CCOMP* set it is part of.

(3) List of *LCS* sets at the processor that connect to at least a border of the information region. Each of them points to the *CCOMP* set they are part of. These *LCS* sets are called active *LCS* sets.

The sets above are represented by their labels alone. The information region before the first stage is nothing but the processor's region. Hence, the *CCOMP* sets before the first stage are the same as the *LCS* sets that touch a border. The border segments are those that have been passed to the Merge algorithm.

At each stage a processor communicates with another processor whose information region abuts the information region of the processor. The communication is said to be along the x axis (respectively y axis) if the common border is perpendicular to the x axis (respectively y axis). An example x axis communication is shown in Fig. 7. At any stage the communication will be along one particular axis. If the circuit is divided into $2^k \times 2^{d-k}$ regions, then we have to perform x axis communication in exactly k stages and y axis communications in the remaining

$d-k$ stages. For the case of area based method of partitioning these stages can be done in any order. However, in case of point based method the y axis communications have to completed before the x axis communications are started. In a stage with x axis communication, a processor $G(i,j)$ will communicate with processor $G(k,j)$, where k is equal to

$$i + 2^l(1-2bit_l(i))$$

Here, l is the number of x axis communications done before the current stage, $bit_l(i)$ is the l^{th} bit of i. An identical formula is true for y axis communications. A communication pattern is illustrated in Fig. 8 for area based method of partitioning. This pattern is also valid for point based partitioning.

Having discussed the communication pattern of the algorithm, we now describe the work done in a particular stage by a processor P. Let Q be the processor that it communicates in the stage.

1: Send the border segments of the current information region of the processor, along with $CCOMP$ sets they point to, in a single message to processor Q. Wait for a similar message from Q.

2: Combine the $CCOMP$ sets of the corresponding common border segments of P and Q. The combination is done by an union-find data structure. The label of the union of two $CCOMP$ sets is taken to be the one that minimizes some bijective function f on the labels. The function f is used to make the process of selection random.

3: The information region of P for the next stage is the combination of its current information region and that of Q. The new border segments are obtained by deleting the border segments along the common border, and combining those sent by Q to those at P in order. The $CCOMP$ sets for the new information region are those that are formed in step 2 that touch a border.

4: For each of the active LCS sets, the new set it belongs to after the unions in step 2 is determined. If the new set is a $CCOMP$ set for the new region than the LCS set is retained in the active list. If not, the label of the new set is taken as the label of the GCS set corresponding to the LCS set. The LCS set is thereafter made inactive.

This completes the description of one stage of the Merge algorithm. After d stages each of the LCS sets gets a label denoting the GCS set it belongs. A processor is involved in d messages in this algorithm. In the description of the algorithm we have omitted the portion describing the way we keep track of the number of LCS sets that are part of a *CCOMP* set. This is done by having a count field for each of the *CCOMP* sets. It is initialized to unity before the first stage of the algorithm. Whenever two *CCOMPS* are merged in step 2 above, their numbers are added and assigned to the union set.

The correctness of this algorithm follows from the important observations that were made after the description of steps P4 and P5. The proof is described in [28], and is omitted here.

3.4. Merge Algorithm on a Shared Memory Multiprocessor

This problem can be solved in an easy and interesting way using a parallel union find algorithm and locking. As in the previous section, each processor computes the border segments for the four borders, with their associated *LCS* set labels. Each processor writes this information onto the shared memory. In the merge step, each processor looks at its top and right border segments. For each border segment a, it determines the corresponding border segment b in the processor at the other end of the border. It then does an union operation on the *LCS* sets of a and b. The union operations performed at different processors interfere, and hence locks are used to ensure correctness.

Consider the operation $union(a, b)$, where a and b are two border segments. The data structure used by the algorithm consists of an union find data structure as shown in Fig. 9(b). The first step in the algorithm is to determine the current roots x, y of $LCS(a)$ and $LCS(b)$ respectively. If x is same as y, the union algorithm returns. If $x \neq y$, the processor locks the roots in increasing order of their labels, in order to avoid deadlock. Once the locks are obtained, the processor checks if x and y are still the roots. If x or y is not a root the whole operation is continued from x, y, after releasing the locks. If x and y are still the roots, a directed edge is added from x to y (or y to x) and the locks are released.

4. PARALLEL PARAMETER EXTRACTION

The parameter extraction phase involves device size extraction, parasitic capacitance extraction and resistance extraction. This is the most time consuming part in circuit extraction. There are different models for parameter extraction, with varying accuracy and computational complexity. In the next sub section, we briefly describe two models, namely *MEXTRA* [17], and *HPEX* [22]. In the following sub section, we describe how we interface the two models in *PACE*. The main theme of this section is to look at the general principles for interfacing any parameter model to the output of the netlist extraction phase of *PACE*. Some of these models may require more information from the netlist extraction phase, but the issues will be the same. The main goal is to achieve a considerable speedup in the parameter extraction phase, which is the most time consuming step in extraction. Also, with the increased computational power, we will be able to use more accurate, and hence more time consuming, models.

4.1. Sequential Extraction Models MEXTRA and HPEX

For device size extraction, we illustrate how they are computed for CMOS transistors. Other devices are handled in a similar fashion. The analog characteristics of an CMOS transistor is a function of length and width. For non rectangular transistors an effective length and width can be computed using techniques like those of resistance extraction. However, a simpler alternative that is adopted in most systems including *MEXTRA* and *HPEX* is to take one half of the transistor's perimeter with diffusion as the effective width. The effective length is then calculated as the area of transistor divided by the effective width.

The rectangles of a net have parasitic capacitance with respect to substrate and also other overlapping or neighboring rectangles. The capacitance of a node (a set of electrically connected rectangles) with respect to the substrate can be accurately estimated for each connected region belonging to the same layer by determining its area A, and perimeter P. This is usually determined as

$$C = \alpha A + \beta P ,$$

where α and β are constants depending on the layer. The capacitances between two nodes from different nets caused by the overlap of rectangles are called overlap capacitances. This is not a simple function of area and perimeter, though many extractors report information about the regions of overlaps between rectangles [25]. Some tile based circuit extractors [20] also determine inter nodal capacitances between two long parallel edges close together, usually in the same layer. These are called parallel capacitances. In *MEXTRA* only the substrate capacitance is computed for each net by the above formula. However, in *HPEX* both substrate capacitance and resistance is computed.

The rectangles of a net have resistance, creating an effective resistance between the terminals of the net. The most accurate method to determine resistances is to use the Laplace equation for the region of the net. However, this is very time consuming. The other approximate approach involves identifying straight sub regions of the net, called branches. Simple analytical formulae can be applied for the evaluation of resistance of branches. Once the straight sub regions are removed from a net, we have islands, each of which can be looked up in the library for its resistance, or some approximation can be used. This approach is used by many resistance extractors [19, 22, 29].

We now describe the resistance capacitance model used in *HPEX* [22]. The rectangles of a net are first converted into a horizontally maximal non overlapping form. This is done by using a scan line algorithm. The reason for this step is that this process will produce an unique representation for the net. The next step is to combine horizontally long rectangles in the vertical direction. In this step, two rectangles, that are sufficiently longer in the x direction than the y direction, are combined vertically if they abut on their horizontal edges. This step is accomplished by a similar scan line algorithm as the first step. The above two steps derive a set of rectangles for the net, that gives an indication of the direction of the current. These two steps are illustrated in Fig. 10.

The next step is to take each rectangle, that is sufficiently longer in one direction than the other, and cut it on its larger side, based on the intersections with other rectangles. Two overlapping rectangles from the intersections are merged into a single one. This process is illustrated in Fig. 11. The next step is to find the branches of the net. Two rectangles are said to be electrically connected if they share more than one electrically connected point. A rectangle that connects to more than two rectangles, or to only one rectangle, is defined to be a knot. Also, the rectangles associated with the terminals of the devices are defined as ports. The rest of the rectangles are defined as branch rectangles. The knots and ports are then assigned unique numbers. A branch of a net is defined to be a path from a knot or port to another knot or port using only branch rectangles. The branches can be determined by a pass of the electrical connectivity graph. The above steps are illustrated for the net of Fig. 10 in Fig. 12.

The next step is to perform the resistances calculations for the branches. For this step, analytical formulae [22] are used for various configurations. These include formulae for a long straight rectangle, a cascade of rectangles in the same direction, corners and bends. In the calculation for resistance of a branch, the contributions of knots and ports are carefully approximated.

The calculation for substrate capacitance is done along with the resistance calculation. The capacitance of each of the knots and ports is first determined. Then, the capacitance of each of the branches is computed by adding the contributions of all its branch rectangles. The capacitance of a branch is lumped at its end points equally. Thus the result of parameter extraction is a distributed RC network for each net. If such a network is directly output, it would result in a large simulation time. In order to avoid this, there is a node reduction phase in *HPEX* [30]. This involves finding an approximately equivalent small RC network for each net. This problem cannot be solved by a trivial removal of small resistances and capacitances, as a large number of them could make a lot of difference. A heuristic scheme is used in *HPEX* [30] for this purpose.

4.2. Interfacing MEXTRA and HPEX in PACE.

The issues in parallel parameter extraction are the same for both hypercube multiprocessors and shared memory multiprocessors. For the local devices not connected to a boundary, we have all the information for the device extraction. For the others, we do the following. If a local device is not owned by the processor, then its rectangles, and other necessary information are sent to the processor that owns the global device. For example in CMOS technology, the necessary information consists of the abutting diffusion rectangles. If a local device is owned by the processor, it waits for messages to complete the information for the global device. The processor can then extract the device. A processor can determine when all the information for a device has been received as the number of local devices that comprise the global device is known from the Merge algorithm. All communication between two processors in the above step can be done in a single message, since the information that needs to be sent is completely determined before the step. Thus the communication can be done very efficiently.

The next step is the extraction of capacitances and resistances of nets. A general approach similar to device extraction could be followed. In other words, local nets touching the border but not owned by a processor can be sent to its owner which can then perform the detailed extraction of the net. This approach can be expected to distribute these nets equitably among processors, since the Merge algorithm determines the label of a *GCS* set in a random fashion from its constituent *LCS* set labels. The problem, however, is that a net may take different amounts of time for extraction depending on its size causing variation of load among the processors. For example, a processor which owns a power or ground net will have an excessive amount of load. Thus long nets pose a problem for this approach.

An examination of the above problem reveals that parallelism can be achieved only if we can deal with the extraction of long nets in a distributed fashion. A global net with more than K rectangles is designated as a long net (K can be chosen to be around 100). The other global nets are designated as short nets. The Merge algorithm is changed to determine for each GCS set, the number of rectangles in its set, similar to the way it determines the number of LCS sets in a GCS set. The determination of whether a global net is a long or a short net is then made easily. Subsequently, for a short net we could use the approach described above for device extraction. For a long net, we need to design a way to extract it in a distributed fashion, so that it is as close to the original model as possible. This is the general approach that we have taken for the problem. We will now illustrate this approach for the models of *MEXTRA* and *HPEX*.

The *MEXTRA* model only computes the substrate capacitance for the nets. It does not determine any resistances. Now, suppose we have a global net G, which comprises of the local nets L_1, ... L_k. For each of the local nets L_j we can compute its capacitance. This involves converting the rectangles stored in a local net into a non overlapping format, and then finding the contribution to the capacitance from each rectangle. However, the substrate capacitance of G is not just the sum C_{tot} of the substrate capacitances of its local nets, because of the possible over counting of the perimeter component of the capacitance equation. However, this over counting can be resolved precisely by sending all the rectangles of L_j that touch the processor border to the processor owning G. The processor owning G can determine all those instances when a border rectangle B_1 of layer l belonging to L_i touches a border rectangle B_2 belonging to L_j of the same layer, where $i \neq j$. For each such instance, we subtract $2\beta\,dist$ from C_{tot}, where β is the perimeter constant of the layer l and $dist$ is the length of the line of contact between B_1 and B_2. With this correction, we get the correct value of the capacitance for the global net G. Because of the accuracy of the method, we extract every global net, even if it is not a long net, in the above distributed fashion. This step is illustrated in Fig. 13. The communication in this step is done in an efficient manner, as in the previous step for extraction of devices.

The *HPEX* model computes both resistances and capacitances. Let G be a long global net comprising of the local nets $L_{1,} \cdots L_k$. We now describe how it is extracted in a distributed fashion. For each of the local nets L_i the two steps of Fig. 10, and the rectangle decomposition step of Fig. 11 are performed. The next step is to determine the knots and branch rectangles in L_i and assign numbers to the knots. In this step, the rectangles of L_i that touch a border are not classified as knots or branches, though they are counted in the connectivity calculations for other rectangles. The numbers assigned to the knots will contain the processor's address bits concatenated at the end. The above steps are illustrated in Fig. 14(a).

The next step is to consider those branches in L_i, that start with a border rectangle, and end with a knot or port or another border rectangle. These branches represent the incomplete branches of L_i. For example, in processor 0, the branch from knot 24 to the boundary is an incomplete branch. All the rectangles in the incomplete branches, including those at the endpoints, are then deleted from L_i and sent to the processor that owns G. Thus, the knot 24 and all the rectangles in the incomplete branch are deleted from processor 0 and sent to the owner. This step is illustrated in Fig. 14(b).

The rectangles in all the incomplete branches of G are received by the owner of G and combined into a set S. The rectangles in S may include some knots and ports. The set S is then transformed by the two steps of Fig. 10, and the rectangle decomposition step of Fig. 11. In these steps, the rectangles that have been designated as ports or knots are not included. The new knots in S are subsequently determined and numbered. The above steps are illustrated in Fig. 14(c). The new knots determined are 1, 5, and 9. The complete result is shown in Fig. 14(d).

For each processor, that contains a part of the global net G, a portion of S that is sufficient to complete its incomplete branches is sent back by the owner of G. On receiving this, these rectangles are included in the local net L_i. For example, processor 0 gets back the branches 24 to 1 and 2 to 1 of Fig. 14(c). Note that the branch 2 to 1 of Fig. 14(c) is also sent to processor 2 by the owner. Once all the rectangles are received, the branches of L_i are determined. A processor computes the capacitance for each of the knots that it owns. It evaluates the resistance and capacitance of a branch if it owns one of the end points. The capacitance determined is lumped at the end points of the branch owned by the processor appropriately. A processor will report the capacitance of a port or knot only if it owns it. Similarly the resistance of a branch is reported by a processor only if one of the end points is owned by the processor, and the number of the other endpoint is larger than it. This avoids duplication of information in the output.

In Fig. 15, we illustrate the working of the algorithm, with point based rectangular partitioning, on an example circuit, for four processors. The value of K, which gives the minimum number of rectangles required in a net in order to be classified as a long net, was taken as 0. Thus all the global nets are extracted in a distributed fashion for this example. In this figure, we show the whole circuit and the set of rectangles processed by processor 0. During parameter extraction, a processor owning a global net receives only those portions of the net, that are close to the lines of partitioning. For example, the isolated top set of rectangles in Fig. 15(b) is a portion of a global net owned by processor 0, and was sent to it by processors 1 and 3.

The procedure described above guarantees that the quality of the result is not compromised in any way by the parallel algorithm as compared to the sequential algorithm, since the same structure of knots and branches is obtained even though the global net was evaluated in a distributed manner. A simpler approach could have been used, that breaks each of the long global nets at the borders by inserting new knots. However, this approach may create inaccuracies in the direction of the current in the border rectangles. For example, when a long horizontal rectangle is split by a horizontal border line, the direction will be incorrectly assigned to be in the vertical direction.

It is easy to interface a node reduction phase, such as [30], at this stage. We will also be able to do the node reduction in parallel. The node reduction step for short nets or local nets that do not touch a border can proceed as before. However, for long nets node reduction can be done only for nodes that do not connect to a node not owned by the processor. Currently we have not interfaced a node reduction phase, but intend to do it in the future.

5. ANALYSIS

In this section we analyze the time taken by the algorithm. This analysis also determines how the performance scales with the number of processors. This also gives us an idea how more complicated parameter models could be interfaced so that resulting parallel algorithm is efficient. For the analysis, we omit the time for reading in the circuit or printing out of the results.

5.1. Analysis for a Hypercube Multiprocessor

Let $p = 2^d$ be the number of processors. Steps P1-P3 in the parallel algorithm *PACE* are performed by a single scan line pass over the rectangles. This phase is exactly similar to the one in a sequential extractor. The scan line algorithm for netlist extraction takes time $t(n) = c_1 n \log n$, where n is the number of rectangles. Because the number of rectangles assigned to a processor is reduced by a factor of p, where p is the number of processors, the time for local netlist extraction is given by

$$c_1 \frac{n}{p} \log \frac{n}{p}$$

where n is the number of rectangles in the whole circuit.

The steps P4-P6 correspond to the overhead of parallelization of the netlist phase. The Merge algorithm is the most dominant among these steps. The message size for a certain stage in the Merge algorithm depends on the number of border segments along the information region of the processor. This is proportional to the perimeter of the information region, assuming uniform distribution of rectangles. Thus the number of border segments in any stage is expected to be at most $c_2 \sqrt{n}$. Thus the total time taken in the Merge algorithm is given by by $d t_\alpha + d c_2 \sqrt{n}\, t_\beta$, where t_α is the startup time of a message, t_β is the time taken to send a byte. All these messages travel a hop of length 2 or 1. This shows that the Merge algorithm is quite fast.

We now analyze the parameter extraction step. We first analyze the device extraction phase. The time taken by a sequential extractor for device extraction will be $c_3 t$, where t is the total number of devices in the circuit and

c_3 is the time to extract one device. In the parallel algorithm, we observe that the communication in this step is usually restricted to the immediate neighbors of a processor, because it is unlikely to have a device that extends across a full processor region. This implies that a node will send out a maximum of 8 messages in this step as the number of neighbors is expected to be around 8. The total sum of the message lengths is expected to be small as it corresponds to the number of devices that cross a border. The expected value of this is $c_4\sqrt{n/p}$. Hence the time taken for communication in this step can be written as $8t_\alpha + c_4\sqrt{n/p}\,t_\beta$. The total time can hence be written as $c_3\frac{t}{p} + 8t_\alpha + c_4\sqrt{n/p}\,t_\beta$.

The parameter extraction of nets is the most time consuming step of the algorithm. We analyze the parameter extraction of nets according to the model of *HPEX*, the other model being similar. The sequential extraction according to this model takes time $h(n) = c_5 n \log n$, where n is the number of rectangles in the circuit. Here c_5 is a comparatively large constant. In our parallel scheme, the communication in this stage will not be confined to only the neighboring processors. This is because a long global net could extend over a lot of processors. However, the total sum of the message lengths depends on the number of rectangles near the boundary. This is expected to be around $c_6\sqrt{n/p}$. Thus the total time for communication can be written as $n_p t_\alpha + c_6\sqrt{n/p}\,t_\beta$, where $n_p < p$ is the maximum number of processors communicated by any processor. The total time for the parameter extraction step is hence $c_5(\frac{n}{p} + c_6\sqrt{n/p})\log\frac{n}{p} + n_p t_\alpha + c_6\sqrt{n/p}\,t_\beta$. The second term in the bracket takes into account the extra computation resulting from the splitting of nets.

Summarizing, the time for sequential extraction can be written as

$$T_1 = Cn\log n,$$

where C is a constant. The time for parallel extraction on p processors can be written as

$$T_p = C\frac{n}{p}\log n + D\sqrt{n}\log n,$$

where D is a constant. Thus the speedup is

$$S_p = \frac{T_1}{T_p} = \frac{p}{1+(\frac{p\,D}{C})(\frac{1}{\sqrt{n}})}.$$

This shows that we should be able to get more speedup as we increase the size of the circuit.

5.2. Analysis for a Shared Memory Multiprocessor

The analysis for this case essentially follows the analysis in the previous section. Instead of sending messages, the processors communicate using the shared memory. The main difference is in the merge step. If we assume that the contention between processors is very minimal (a reasonable assumption), i.e two processors do not perform an union operation on the same set at the same time, the time taken by the merge step can be given as the number of union operations attempted. Clearly, this is $O(\sqrt{n/p})$. Thus the merge operation is very fast. The final speedup expression derived in this case is the same as in the previous section.

6. RESULTS AND DISCUSSION

The parallel algorithms in *PACE* has been implemented on the iPSC2 hypercube using 8600 lines of C code, and the Encore 510 Multimax shared memory multiprocessor using 6800 lines of C code.

6.1. Results for a Hypercube

In Table 1, the performance of the algorithm *PACE* with the *HPEX* model is presented for a number of VLSI custom layouts. All these chip layouts were obtained by VLSI design projects at the University of Illinois that were subsequently fabricated. These circuits range in various sizes and have varying degrees of regularity. Some of these circuits are extremely regular (e.g multiplier array) while the others have both high density regions and low density regions consisting of free space and routing (e.g placement coprocessor). The table shows the performance of the algorithm for different number of processors and different partitioning strategies, ignoring the time for input of the circuit and the output of the results. In Table 2, similar performance results are presented for the *PACE* algorithm with *MEXTRA* model. Some of the entries in the tables are not shown due to memory problems on the hypercube (each processor node has only 4 Mbytes of memory of which roughly 200K is taken by the operating system and some more by the buffers).

The performance results can be explained as follows. The performance of any parallel algorithm depends on the load balance, computation and communication overhead. The load balance in *PACE* is better in point based methods. This is because each processor gets nearly the same number of rectangles. This is very effective when the circuit has some regions with white space and some less dense regions like the routing areas. The communication and computation overhead is generally less in rectangular partitioning. This is because slice partitioning will in general cut more rectangles than rectangular partitioning creating more nets that touch a border. These two factors jointly influence the performance of the *PACE* algorithm in a way that depends on the circuit. Thus a scheme which addresses both these factors should do uniformly well on all the circuits. Hence, it is not surprising to see that point based rectangular partitioning does very well on all the circuits.

We would like to verify the expressions derived in the analysis section for speedup. The speedup expression suggests that given a value of n, there is a limit on the possible speedup i.e $\frac{C\sqrt{n}}{D}$. However, increasing the value of n will increase the limit on the possible speedup. Thus larger circuits permit the use of more processors effectively. This observation can be verified, for example, in Table 1. For the first circuit, increasing the processors from 8 to 16 decreases the time from 59.6 to 43.5, for point based partitioning. However, for the fifth circuit, increasing the processors from 8 to 16 decreased the time from 220.7 to 101.8.

Though the speedup function derived predicts the general trend, it ignores certain factors that were clear form the experiments. The first factor involves the load imbalance in the parameter extraction phase, even if the circuit is split in a load balanced fashion. This is caused due to the imbalance in the assignment of the LCS sets, that touch a border, to the processors. This imbalance becomes more pronounced when the number of processors is increased for a fixed n, thus decreasing the speedup. The second factor involves the observation that when the number of processors is increased, each processor gets a lesser portion of the circuit, implying a lesser memory requirement at each processor. Thus the memory allocation functions become faster, increasing the speedup. This is especially true if the circuit takes a large portion of the processor's memory.

In Fig. 16, the load balancing property of point based rectangular partitioning is illustrated. Specifically, the performance of area based and point based rectangular partitioning is compared for the fourth circuit (placement coprocessor). This graph shows the amount of CPU time spent by each processor. This time excludes the time spent in sending, receiving, and waiting to receive a message. The graph also shows the total time for extraction (this includes the communication time). This graph shows that the point based partitioning is an effective load balancing strategy. Also, the total time of computation, for both the partitioning methods, is very nearly equal to the CPU time of the processor that has the maximum CPU time. This implies that the communication overhead of the algorithm is small. In Fig. 17, a graph is presented that compares the performance of slice and rectangular point based methods on the fifth circuit (associative cache). This graph is similar to one in Fig. 16. The graph shows that the computation and communication overhead of slice partitioning is greater than that of rectangular partitioning.

6.2. Results on a Shared Memory Multiprocessor

The experimental results on the Encore 510 multimax are nearly the same as for the case of the hypercube. In Table 3, we give the time taken by the algorithm for 1 and 8 processors. The speedup obtained is less than expected, because of memory contention.

6.3. Comparison on an Uniprocessor

In Table 4, the performance of *PACE* (a sequential version) and *MEXTRA* is presented on a uniprocessor. The table shows performance of these algorithms on Sun 3/280 and Sun4/280 file servers. The times reported are only the CPU times, and exclude the time for input and output. It should be noted that both the sequential implementation of *PACE* and *MEXTRA* could not run for some of these circuits on Sun 3/280 due to memory limitations. The original *HPEX* program is not included in this table as it took extremely large times for the big circuits. Also, it should be noted that the sequential implementation of *PACE* with the *MEXTRA* model is faster than the original *MEXTRA* program, which shows that our sequential implementation is very efficient. Hence our parallel speedup numbers are very realistic. These results show that *PACE* has achieved very good speedups for the big circuits.

7. CONCLUSIONS

In this paper, we looked at the problem of designing a parallel algorithm for circuit extraction. We have designed a parallel algorithm *PACE* and have implemented it on an Intel hypercube and an Encore shared memory multiprocessor. We have presented the results of our implementation which shows that significant amounts of speedups can be achieved for this problem. We have described different partitioning strategies and compared their performance.

The main conclusion of this research is that with parallel processing we can effectively use more accurate and time consuming extraction methods. This research has provided a general outline of how one would parallelize any such method. Future research for parallel circuit extraction should concentrate on how different extraction models can be interfaced using this general principle. These include models that compute cross coupling and side wall capacitances. The computation of these across processors will be a tough problem. Also, future research should look at how node reduction can be done in parallel.

Combining the benefits of hierarchy and parallelism is also an important area for future research. As it was mentioned before, most hierarchical extractors first transform a circuit in which there are overlapping cells into one that does not have any overlapping cells. This transformation step can be done in parallel reducing the overhead. The next step is to assign non overlapping cells to processors for extraction. This could be done using well known approximate processor scheduling techniques [31] so that the maximum load on a processor is minimized. This approach can be expected to give good results if there are a large number of cells and if no cell has an extraordinarily large number of rectangles. However, if there are a few very large cells then it might be necessary to divide the extraction of a cell among processors, using the techniques described in this paper. The way the workload has to be distributed among processors for optimal performance is an interesting problem and needs to be looked into.

Though we have assumed Manhattan geometry in our discussion, the parallel algorithm described in this paper can be easily extended for the non Manhattan case. For example, let us suppose the circuit is described as a set of trapezoids. The steps P1-P3 that are executed locally at a processor need to be changed so that they can handle trapezoids. The definition of maximal border segments and the Merge algorithm remain as before. Finally, the parameter extraction modules have to be changed so that they can handle trapezoids. Thus the only changes that need to be done are in the serial steps of the algorithm at each of the processors. The parallel algorithm remains the same as before.

The point based partitioning described in the paper may also have applications in designing parallel algorithms for other layout related VLSI CAD problems. It could also be looked upon as a distributed data structure for storing rectangles. This might be useful for some interactive VLSI CAD applications.

REFERENCES

[1] T. Blank, ‘‘A Survey of Hardware Accelerators Used in Computer-Aided Design,’’ *IEEE Design and Test*, pp. 21-39, Aug. 1984.

[2] P. Agrawal, W. J. Dally, A. K. Ezzat, W. C. Fischer, H. V. Jagdish, and A. S. Krishnakumar, ‘‘Architecture and Design of the MARS Hardware Accelerator,’’ *Proc. 24th Design Automation Conf.*, pp. 108-113, Jun. 1987.

[3] F. Darema and G. F. Pfister, ‘‘Multipurpose Parallelism for VLSI CAD on the RP3,’’ *IEEE Design and Test of Computers*, vol. 4, pp. 19-27, Oct. 1987.

[4] R. J. Smith, ‘‘Fundamentals of Parallel Logic Simulation,’’ *Proc. 23rd Design Automation Conf.*, pp. 2-12, Jun. 1986.

[5] R. E. Bryant, ‘‘Data Parallel Switch level Simulation,’’ *Proc. Int. Conf. on Computer-Aided Design*, pp. 354-357, Nov. 1988.

[6] J. T. Deutsch and A. R. Newton, ‘‘MSPLICE: A Multiprocessor-based Circuit Simulator,’’ *Proc. Int. Conf. on Parallel Processing*, pp. 207-214, Aug. 1984.

[7] S. Patil and P. Banerjee, ‘‘A Parallel Branch and Bound Algorithm for Test generation,’’ *Proc. 26th Design Automation Conf.*, pp. 339-343, Jun. 1989.

[8] A. Casotto, F. Romeo, and A. S. Vincentelli, ‘‘A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells,’’ *IEEE Trans. Computer-Aided Design*, pp. 838-847, Sep. 1987.

[9] S. A. Kravitz and R. A. Rutenbar, ‘‘Placement by Simulated Annealing on a Multiprocessor,’’ *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 534-549, Jun. 1987.

[10] J. Rose, ‘‘Locusroute: A Parallel Global Router for Standard Cells,’’ *Proc. Design Automation Conf.*, pp. 189-195, Jun. 1988.

[11] G. E. Bier and A. R. Pleszkun, ‘‘An Algorithm for Design Rule Checking on a Multiprocessor,’’ *Proc. 22nd Design Automation Conf.*, pp. 299-303, Jun. 1985.

[12] F. Gregoretti and Z. Segall, ‘‘Analysis and Evaluation of VLSI Design Rule Checking Implementation in a Multiprocessor,’’ *Proc. Int. Conf. Parallel Processing*, pp. 7-14, Aug. 1984.

[13] E.C. Carlson and R.A. Rutenbar, ‘‘Mask Verification on the Connection Machine,’’ *Proc. 25th Design Automation Conf.*, pp. 134-140, Jun. 1988.

[14] S. Levitin, *A Multiprocessing Approach to Circuit Extraction*. MIT, 1986, Master’s Thesis.

[15] K. P. Belkhale and P. Banerjee, ‘‘PACE: A Parallel VLSI Circuit Extractor on the Intel Hypercube Multiprocessor,’’ *Int. Conf. on Computer-Aided Design*, Nov. 1988.

[16] K. P. Belkhale and P. Banerjee, ‘‘PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction,’’ *Proc. Int. Conf. on Computer-Aided Design*, pp. 526-529, Nov. 1989.

[17] D. T. Fitzpatrick, ‘‘MEXTRA: A Manhattan Circuit Extractor,’’ Electronics Research Lab Memo M82/42, UC Berkeley, Jan. 1982.

[18] A. Gupta, ‘‘ACE: A Circuit Extractor,’’ *Proc. 20th Deisgn Automation Conf.*, pp. 721-725, Jun. 1983.

[19] S. P. McCormick, ‘‘EXCL: A Circuit Extractor of IC Designs,’’ *Proc. 21st Design Automation Conf.*, pp. 624-628, June 1984.

[20] W. S. Scott and J. K. Ousterhout, ‘‘Magic’s Circuit Extractor,’’ *IEEE Design and Test*, pp. 24-34, Feb. 1986.

[21] R. Hon and A. Gupta, *HEXT: A Hierchical Circuit Extractor*. Computer Science Press, 1983. pp. 18-34.

[22] Shun-Lin Su, Vasant B. Rao, and Timothy N. Trick, ‘‘HPEX: A Hierarchical Parasitic Circuit Extractor,’’ *Proc. 24th Design Automation Conf.*, pp. 566-569, Jun. 1987.

[23] G.M. Tarolli and W.J. Herman, ‘‘Hierarchical circuit extraction with detailed parasitic capacitance,’’ *Proc. 20th Design Automation Conf*, pp. 337-345, 1983.

[24] M. E. Newell and D. T. Fitzpatrick, ‘‘Exploiting structure in integrated circuit design analysis,’’ *Conf. on Advanced Research in VLSI*, pp. 84-92, 1982.

[25] T.G. Szymanski and C. J. Van Wyk, ‘‘Goalie: a space efficient system for VLSI artwork analysis,’’ *IEEE Design and Test of Computers*, vol. 2, no 3, pp. 64-72, 1985.

[26] Y. Saad and M. H. Schultz, ‘‘Topological Properties of Hypercubes,’’ *IEEE Transactions on Computers*, vol. 37, No. 7, pp. 867-872, Jul. 1988.

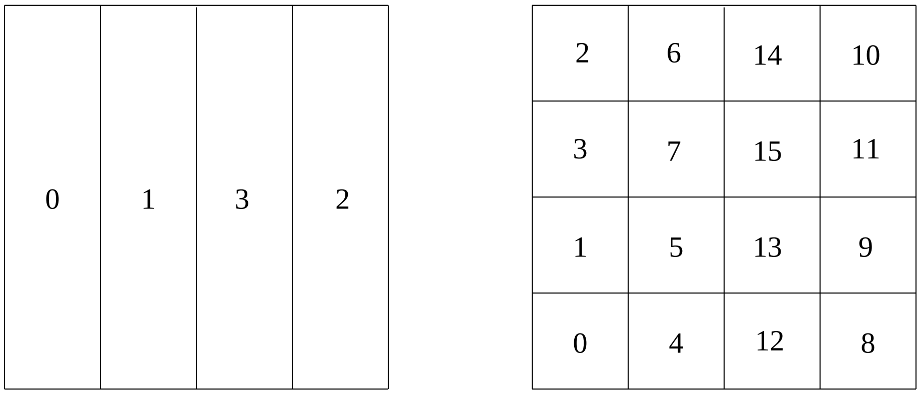
[27] K. P. Belkhale and P. Banerjee, ‘‘Recursive Partitions on Multiprocessors,’’ *Proc. 5th Distributed Memory Computing Conf.*, pp. 930-938, Apr. 1990.

[28] K.P. Belkhale and P. Banerjee, ‘‘Parallel Algorithms for Geometric Connected Component Labeling on a Hypercube Multiprocessor,’’ *IEEE Transactions on Computers*, vol. (submitted).

[29] K.W. Chiang, ‘‘Resistance Extraction and Resistance Calculation in GOALIE2,’’ *Proc. Design Automation Conf.*, pp. 682-685, Jun. 1989.

[30] S. L. Su, V. B. Rao, and T. N. Trick, ‘‘A Simple and Accurate Node Reduction Technique for Interconnect Modeling in Circuit Extraction,’’ *Proc. Int. Conf. on Computer-Aided Design*, pp. 270-273, Nov. 1986.

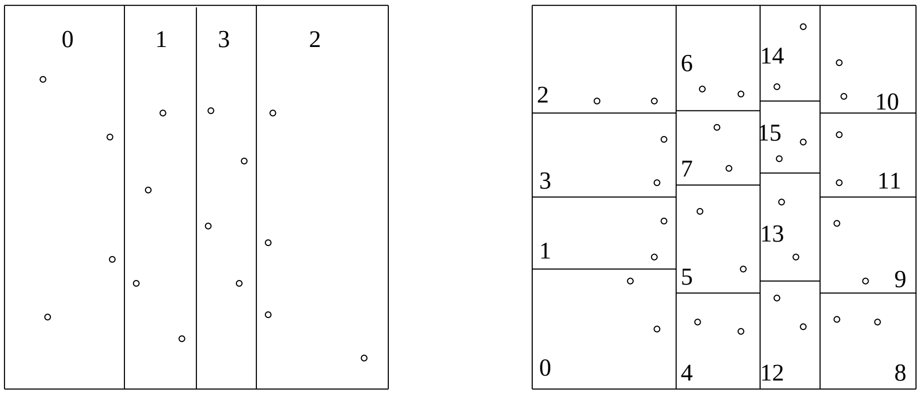
[31] E. Horowitz and S. Sahni, in *Fundamentals of Computer Algorithms*. Rockville, Maryland: Computer Science Press, 1978.



(a) (b)

NOTE: The numbers show the processor assignment used by the *PACE* algorithm on a hypercube.

Fig. 1. Area based (a) Slice Partitioning (b) Rectangular Partitioning



(a) (b)

NOTE: The numbers show the processor assignment used by the *PACE* algorithm on a hypercube.

Fig. 2. Point based (a) Slice Partitioning (b) Rectangular Partitioning

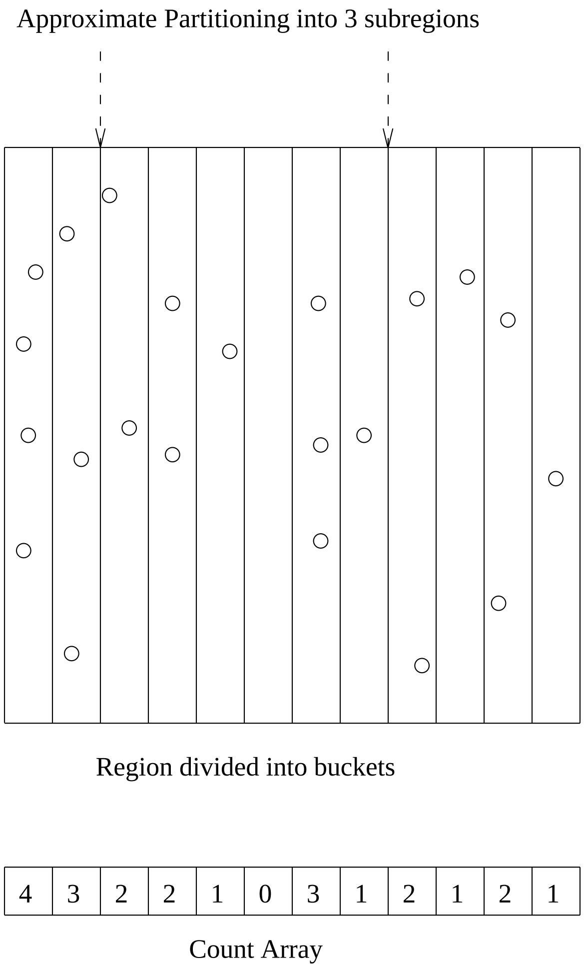
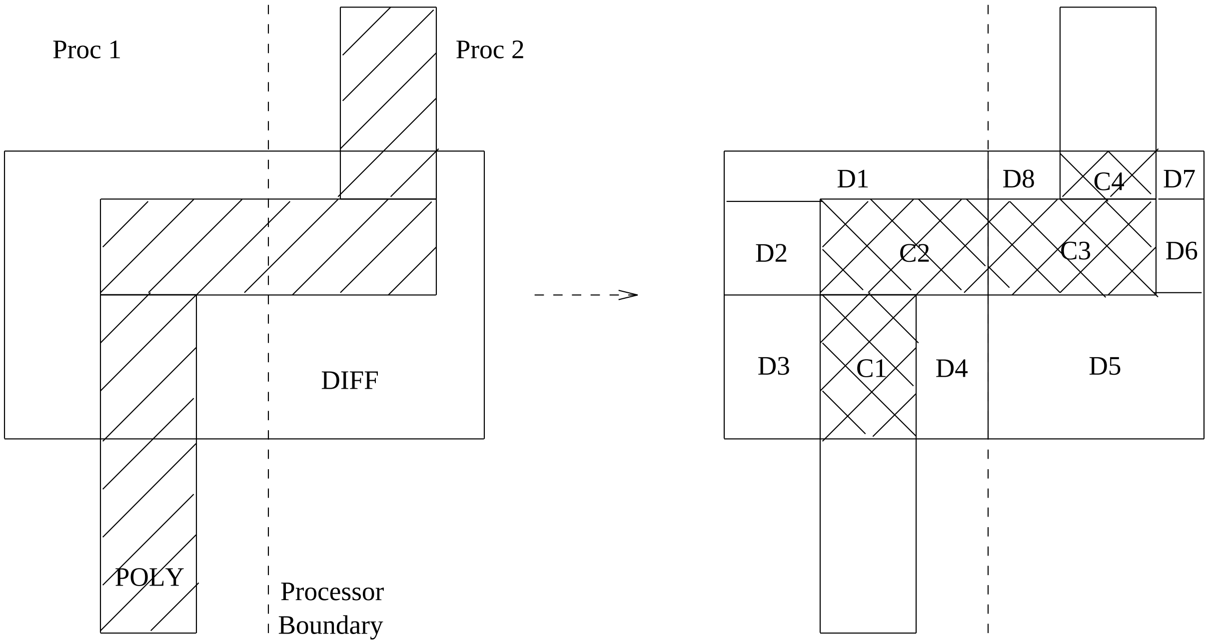
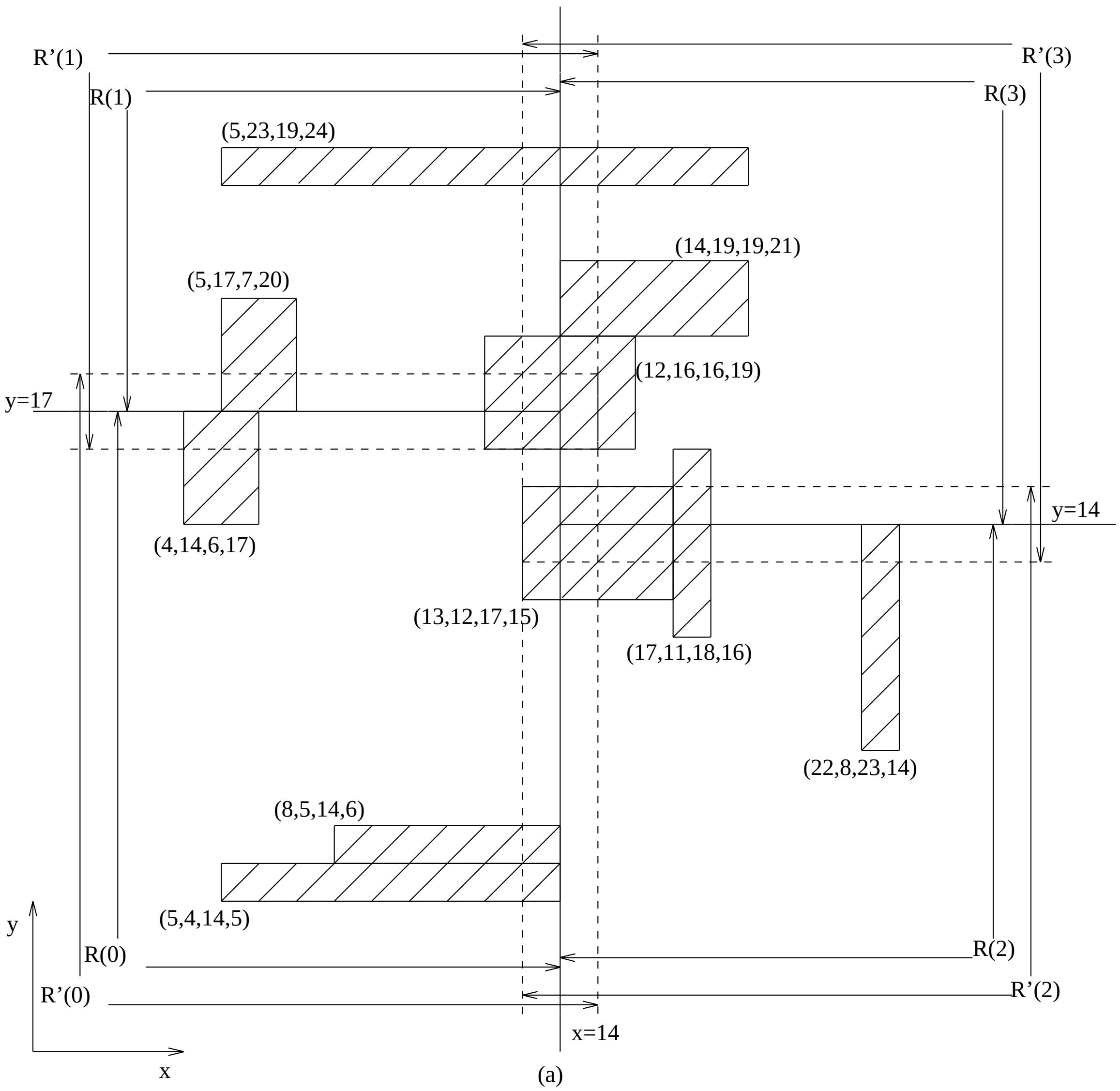


Fig. 3. Implementing approximate point based partitioning.



NOTE: This example illustrates the work involved in determining the device rectangles and splitting of diffusion rectangles. After step P1, C1-C2 are device rectangles in processor 1, C3-C4 are device rectangles in processor 2, D1-D4 are diffusion rectangles in processor 1 and D5-D7 are diffusion rectangles in processor 2.

Fig. 4. Device identification in CMOS technology



(a)

	Proc. 0
Left x	_______
Bot y	_______
Right x	(4,6)(12,15) (16,18)
Top y	(4,7)(12,15)

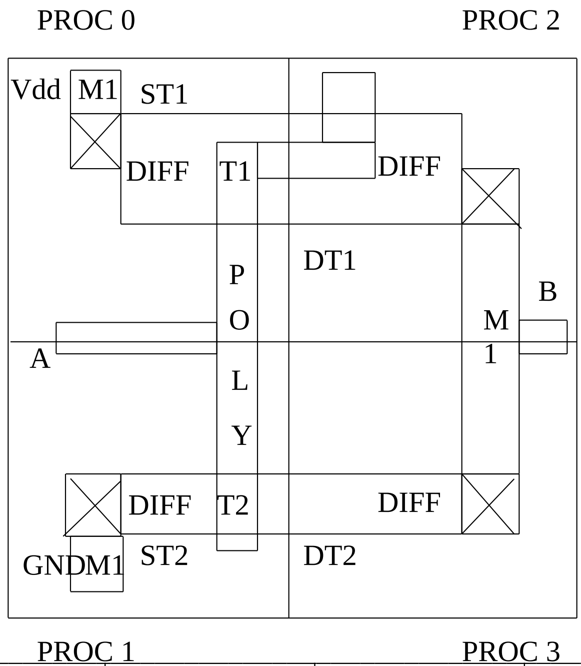
	Proc. 1
Left x	_______
Bot y	(4,7)(12,15)
Right x	(16,21)(23,24)
Top y	_______

	Proc. 2
Left x	(4,6)(12,15)
Bot y	_______
Right x	_______
Top y	(13,18)(22,23)

	Proc. 3
Left x	(13,15)(16,21) (23,24)
Bot y	(13,18)(22,23)
Right x	_______
Top y	_______

(b)

Fig. 5. (a) Set of rectangles in same layer (b) The maximal border segments after step P4.



Proc 0	Proc 1	Proc 2	Proc 3
Lnet 1 (VDD)	Lnet 1 (A)	Lnet 1 (VDD)	Lnet 1 (B)
Lnet 2 (A)	Lnet 2 (GND)	Lnet 2 (A)	_______
Lnet 3 (B)	Lnet 3 (B)	Lnet 3 (B)	_______
Ltran 1 (T1)	Ltran 1 (T2)	Ltran 1 (T1)	______

Fig. 6. Determination of global nets, transistors

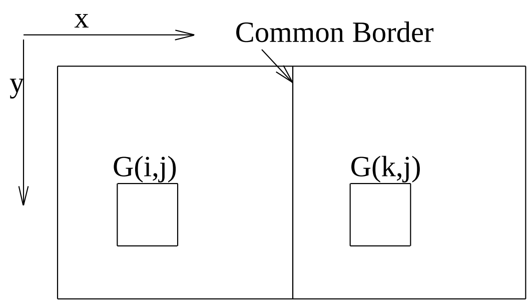


Fig. 7. Proc $G(i,j)$ in an x axis communication

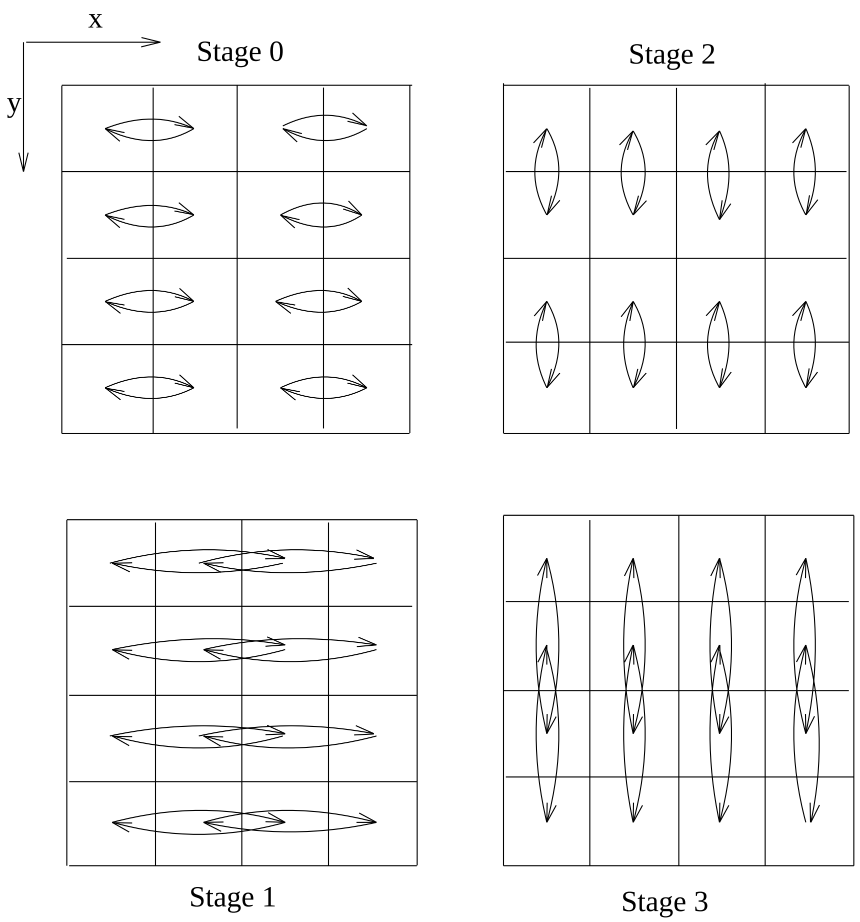
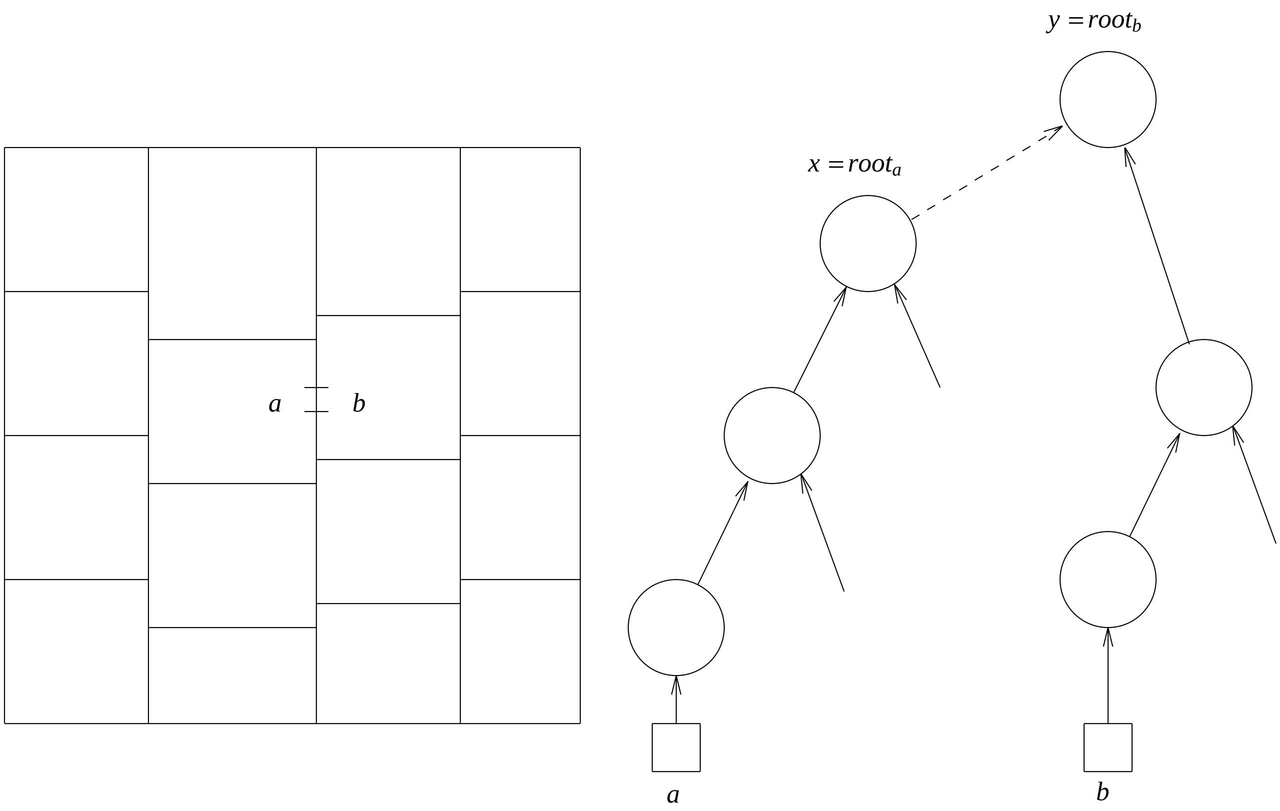


Fig. 8. A communication pattern of merge algorithm for area based partitioning



(a) (b)

Fig. 9. (a) Example partitioning (b) Illustration for merge algorithm

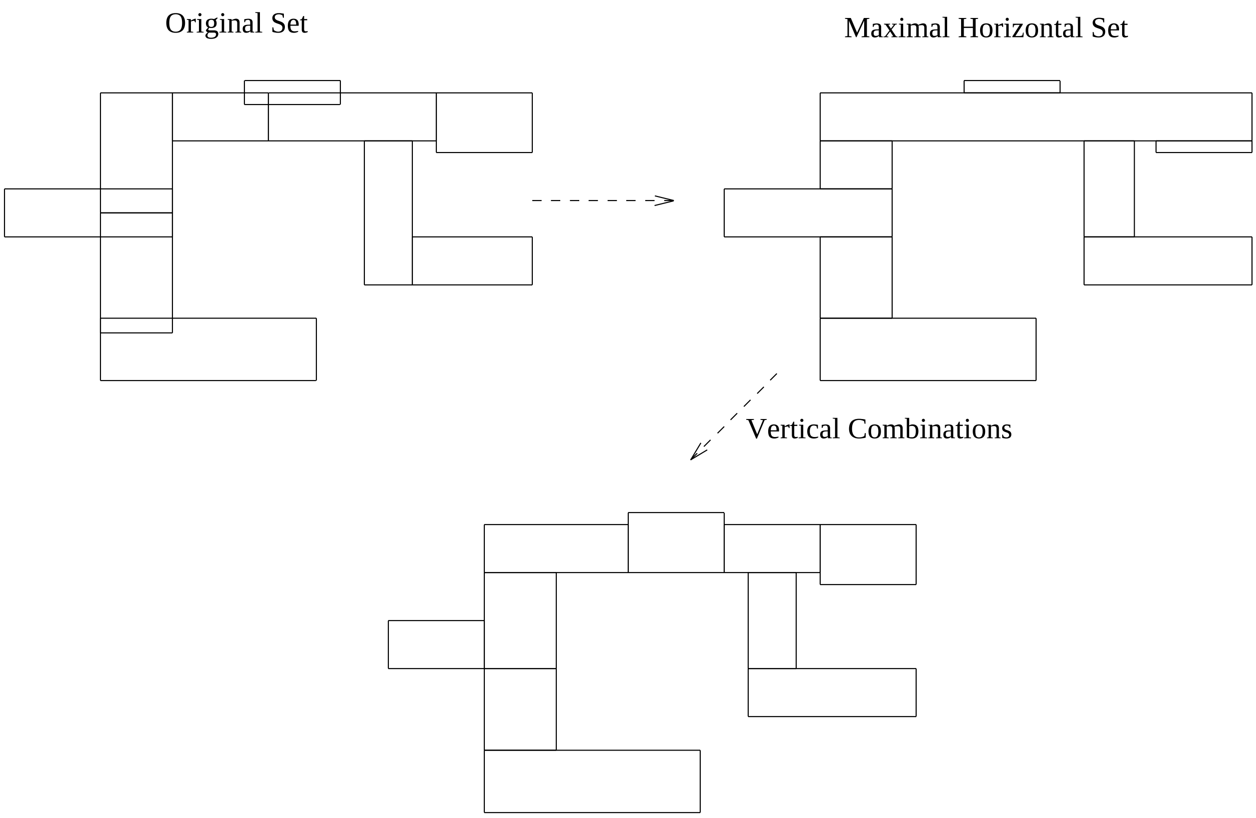


Fig. 10. The first two steps in resistance extraction

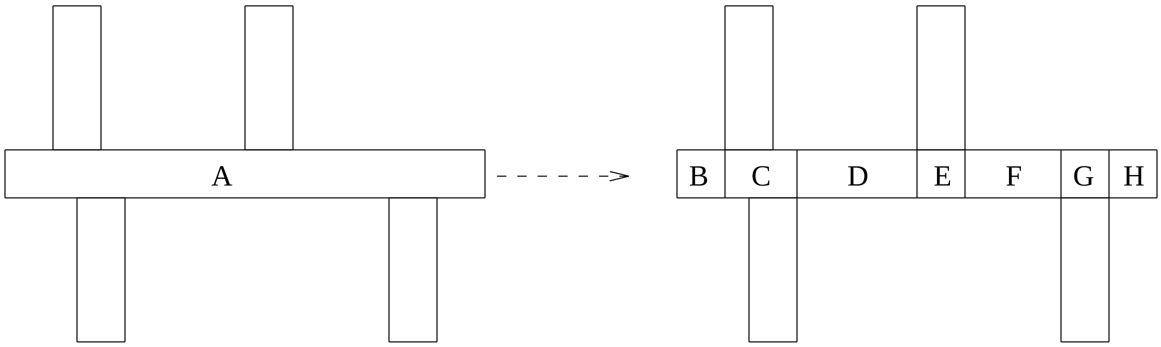
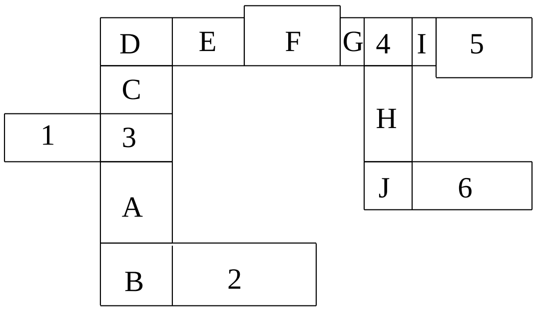


Fig. 11. The rectangle decomposition of ***A***

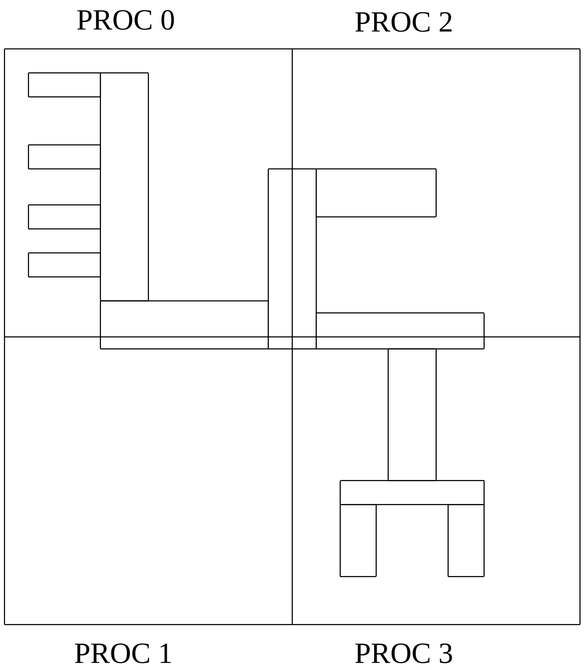


1 2 6 5 - Ports or Knots
3 4 - Knots
A B C D E F G H I J - Branch Rects

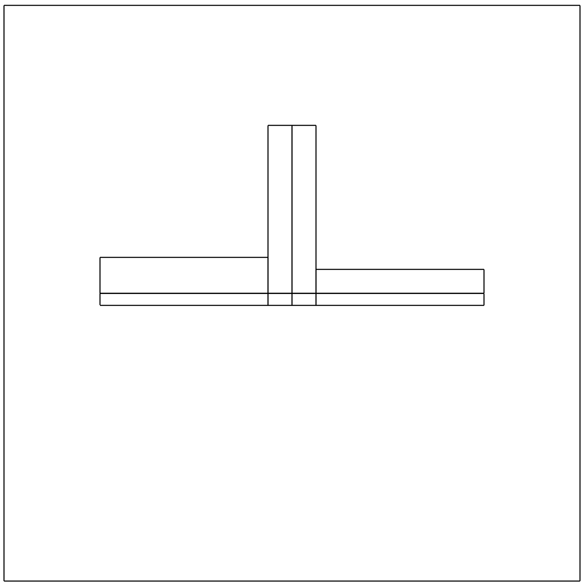
Branches

1 3	4 H J 6	3 C D E F G 4
3 A B 2	4 I 5	

Fig. 12. Rectangle decomposition and branch creation



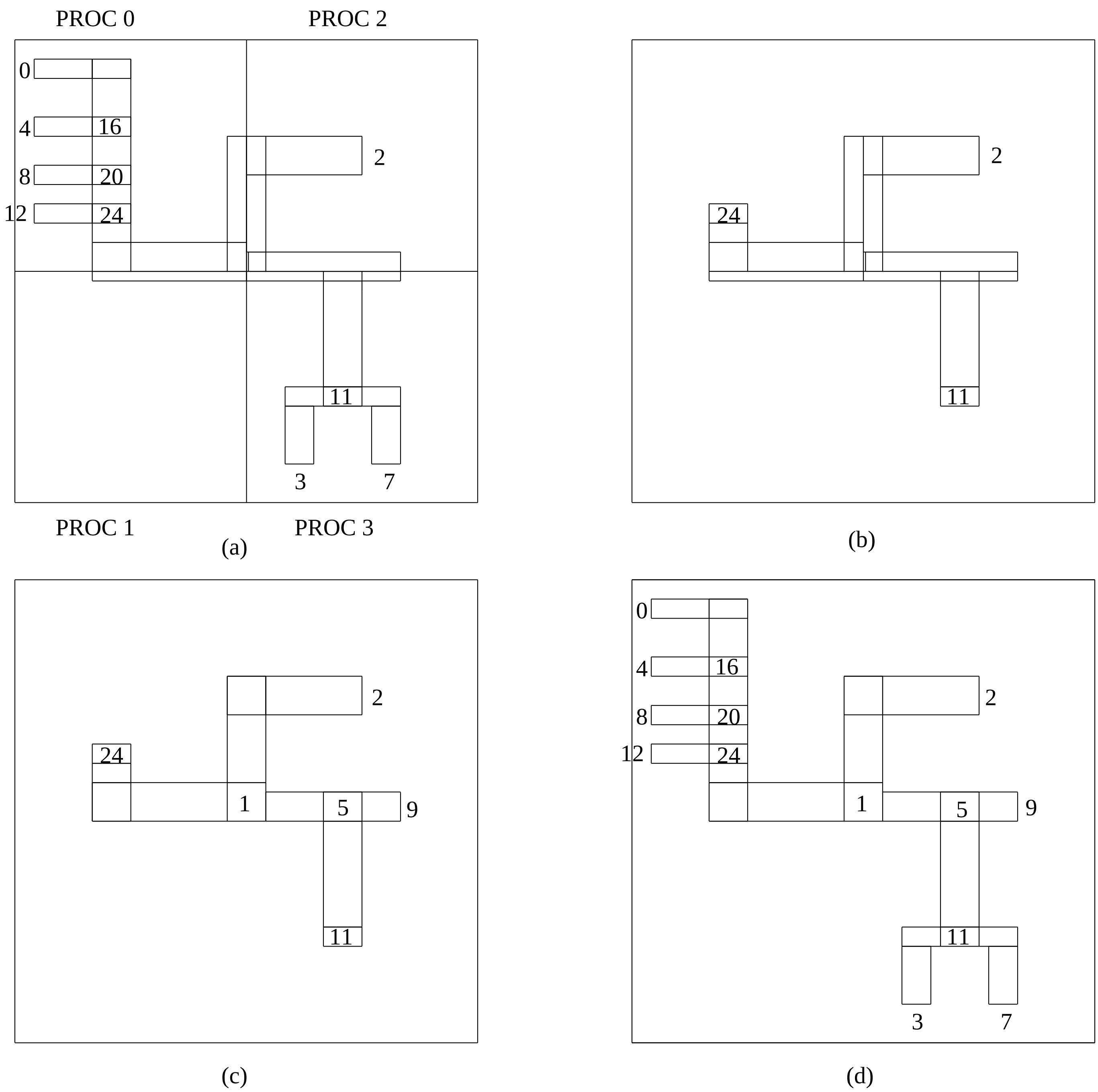
(a)



(b)

(a) A long net (b) Border rectangles received by owner

Fig. 13. Interfacing *MEXTRA* model for extraction



(a) Local processing, (b) Rectangles sent to owner (say proc 1), (c) Processing by owner, (d) Final result.

Fig. 14. Interfacing *HPEX* model for extraction

(a)

(b)

Fig. 15. (a) The example circuit (b) Rectangles processed by processor 0

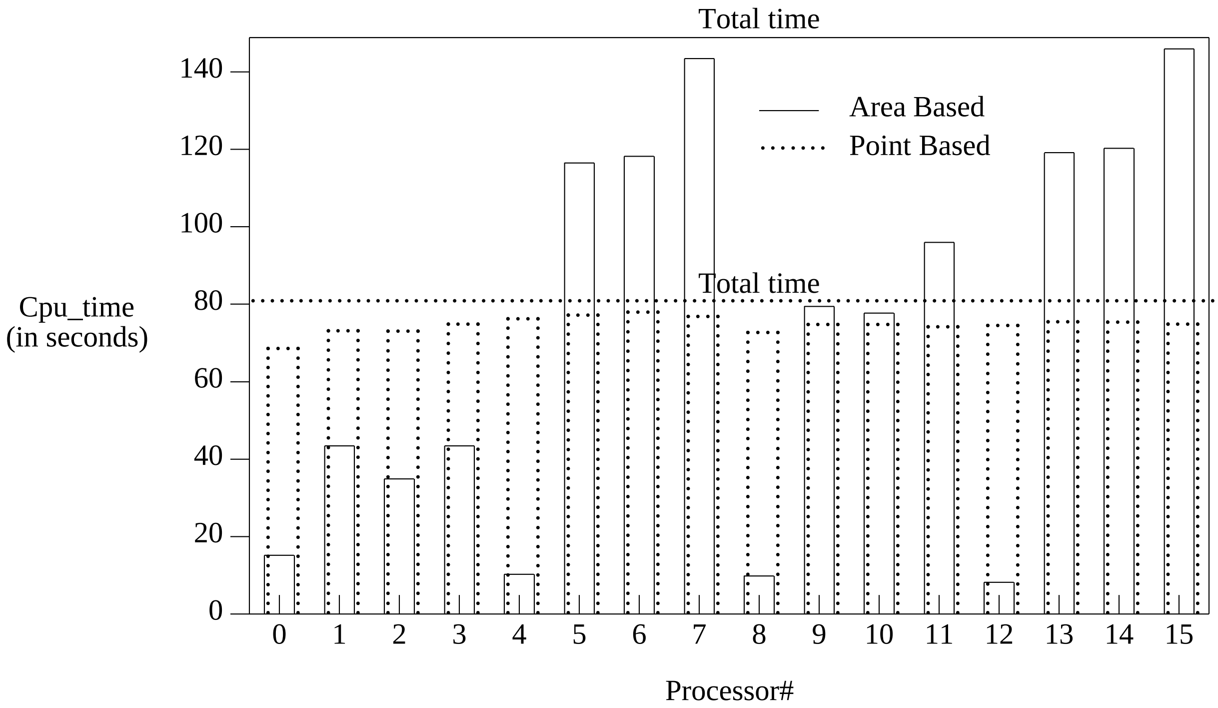


Fig. 16. ***PACE*** **using** ***HPEX*** **model with rectangular partitioning (area versus point)**

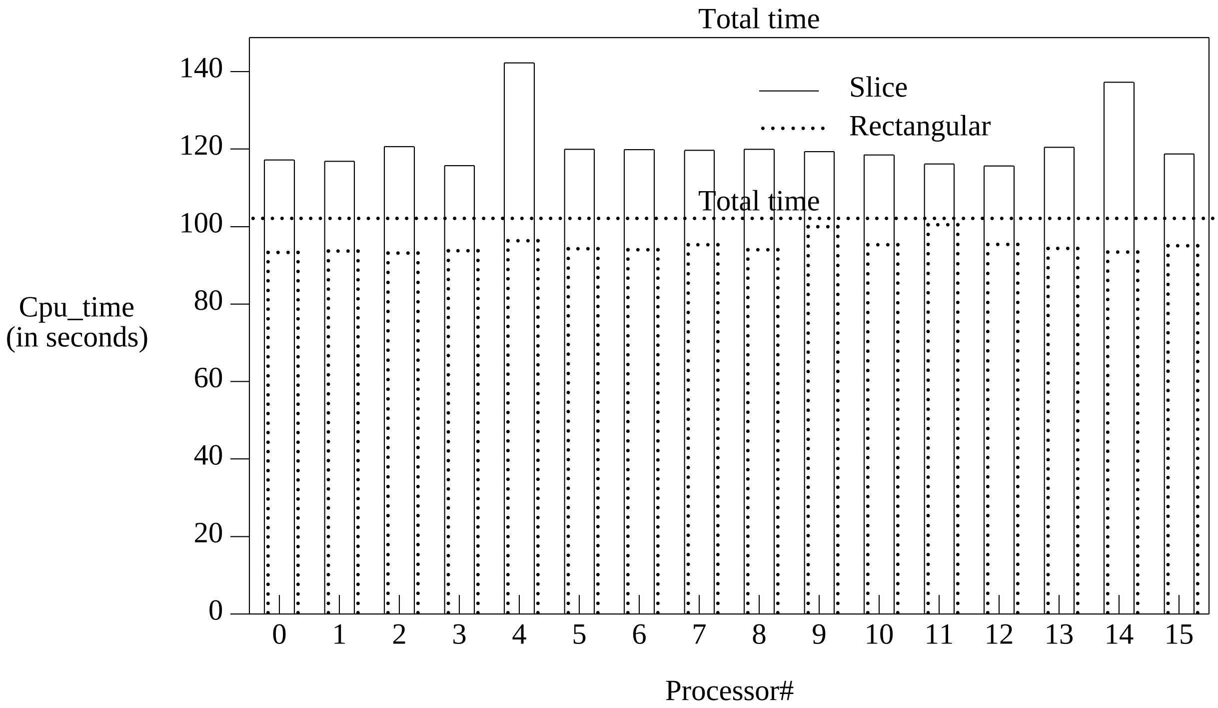


Fig. 17. ***PACE*** **using** ***HPEX*** **model with point based partitioning (slice versus rectangular)**

Table 1. Performance of *PACE* with *HPEX* model on Intel iPSC2/D4-MX

Circuit	Rectangles	Devices	Processors	Area Based		Point Based	
				slice	rectangular	slice	rectangular
				sec	sec	sec	sec
Hypercube Router Chip	54800	3500	1	373.3	373.3	373.3	373.3
			2	203.7	221.2	197.6	207.6
			4	112.6	119.0	103.1	107.6
			8	61.7	62.9	67.9	59.6
			16	56.5	51.6	65.6	43.5
Multiplier Array	66000	8400	2	275.3	275.3	275.5	275.5
			4	112.6	105.5	112.7	106.4
			8	56.4	60.0	64.0	60.3
			16	40.3	32.4	58.3	41.4
Static Ram	130700	14300	4	192.1	188.7	154.0	156.8
			8	93.4	97.7	83.2	78.1
			16	53.0	64.2	47.5	46.3
Placement Coprocessor	258600	28500	8			164.8	163.2
			16	112.7	148.7	90.5	80.5
Associative Cache	258700	38300	8			248.4	220.7
			16	159.3	140.0	148.7	101.8

Table 2. Performance of *PACE* with *MEXTRA* model on Intel iPSC2/D4-MX

Circuit	Rectangles	Devices	Processors	Area Based		Point Based	
				slice	rectangular	slice	rectangular
				sec	sec	sec	sec
Hypercube Router Chip	54800	3500	1	154.1	154.1	154.1	154.1
			2	95.9	89.5	81.4	86.8
			4	43.9	48.8	39.9	43.3
			8	22.2	23.4	22.1	21.4
			16	18.0	16.9	16.4	16.7
Multiplier Array	66000	8400	2	103.2	103.3	104.7	104.7
			4	45.8	44.7	45.6	43.6
			8	22.3	23.6	23.6	23.6
			16	12.6	13.8	15.2	14.8
Static Ram	130700	14300	4	84.6	79.5	68.7	67.7
			8	41.0	43.1	34.9	34.8
			16	21.8	28.2	20.3	22.7
Placement Coprocessor	258600	28500	8			76.5	69.3
			16	50.5	63.8	45.0	39.2
Associative Cache	258700	38300	8	87.3	83.0	83.4	79.4
			16	45.8	53.8	43.9	40.6

Table 3: Performance of *PACE* with *HPEX* model on Encore 510 multimax.

Circuit	Flat Sequential	Flat Parallel (8 processors)
Hypercube Router Chip (54800)	196	43
Multiplier Array (66000)	221	41
Static Ram (130700)	305	63
Placement Coprocessor (258600)	691	137
Associative Cache (258700)	985	190

Table 4: Performance of *PACE* sequential implementation on Sun 3/280 and Sun 4/280

Circuit	Sun 3/280 (8 Mbytes)			Sun 4/280 (32 Mbytes)		
	MEXTRA sec	PACE (MEXTRA) sec	PACE (HPEX) sec	MEXTRA sec	PACE (MEXTRA) sec	PACE (HPEX) sec
Hypercube Router	281.4	204.3	497.6	142.8	71.7	157.6
Multiplier Array	211.3	243.2	620.8	105.0	85.8	195.9
Static Ram	367.9	373.1	967.0	157.5	137.4	332.4
Placement Coprocessor				318.6	282.8	685.3
Associative Cache				453.4	328.9	825.3