PARALLEL ALGORITHMS FOR CAD WITH APPLICATIONS TO CIRCUIT EXTRACTION

BY

KRISHNA P. BELKHALE

B.Tech., Indian Institute of Technology, 1986
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

CHAPTER 1.

INTRODUCTION

1.1. Parallel Processing for CAD

As the sizes of VLSI circuits increases in the future, the computational requirements for performing various computer-aided design (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. Parallel processing provides an effective way for increasing the capabilities of the CAD tools. It not only reduces the time taken by various CAD tasks, but also increases the sizes of circuits that can be effectively handled by the CAD tools.

There are two approaches to the use of parallel processing. The first approach involves the use of special-purpose hardware accelerators. Special-purpose hardware accelerators have 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.

The second approach to parallel processing involves the use of general-purpose multiprocessors. These machines can also offer the tremendous speed improvements that are needed to solve the complex problems of the future. Many commercial general-purpose multiprocessors have been built and are being widely used for various scientific applications. Example machines include Cray XMP, Alliant FX, Intel Hypercube, and Connection Machine. These machines are 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, 12, 13], and circuit extraction [14, 15, 16]. In this thesis, we will describe parallel algorithms for one particular CAD application, namely VLSI circuit extraction.

General-purpose multiprocessors are broadly classified as Single Instruction Multiple Data (SIMD) or Multiple Instruction Multiple Data (MIMD) multiprocessors. In a SIMD multiprocessor, a central controller broadcasts the same instruction to different processors. Each of processors then executes the instruction on its data. It is possible to mask some processors from the instruction by appropriately setting some mask registers at each processor. In a MIMD multiprocessor, the processors execute different instructions which are obtained from the memory. In other words, there are multiple instruction streams, as opposed to a single instruction stream in the case of SIMD multiprocessors. This is illustrated in Fig. 1.1. For many CAD applications, developing algorithms for MIMD multiprocessors is relatively easier than developing algorithms for SIMD multiprocessors. In

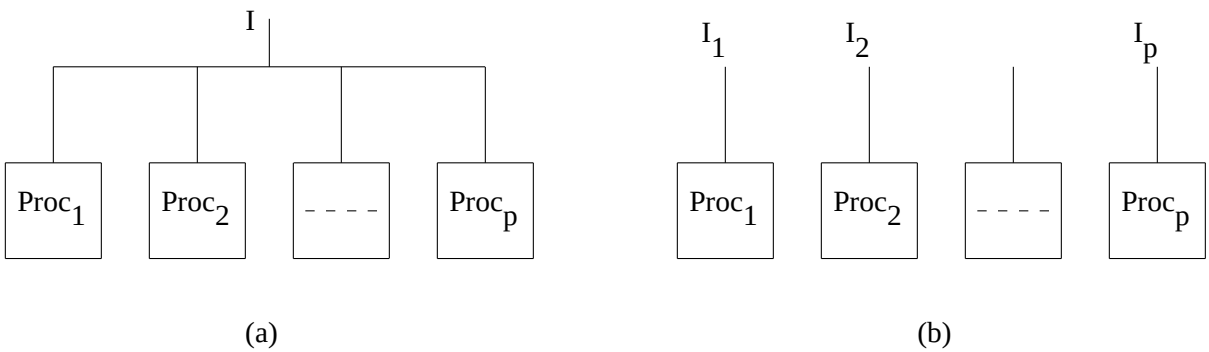


Figure 1.1. (a) SIMD and (b) MIMD general purpose multiprocessors

this thesis, we will be primarily looking at algorithms designed for MIMD multiprocessors.

General-purpose multiprocessors can also be classified as shared memory multiprocessors and distributed memory multiprocessors. This is illustrated in Fig. 1.2. In the case of shared memory multiprocessors, the different processors are connected to a common global memory through an inter-connection network. The interconnection network could be a simple as a single bus. In the case of distributed memory multiprocessors, there is no common global memory. However the processors are connected by a network, which is used for inter processor communication. The use of shared memory multiprocessors is easier, both from the point of view of algorithm design and program development. However, connecting many processors to a single global memory, and yet maintaining a small access time, becomes very expensive for large number of processors (greater than 20). On the other hand, distributed memory multiprocessors are more scalable. However, the algorithm design and program development required by these machines is more complex.

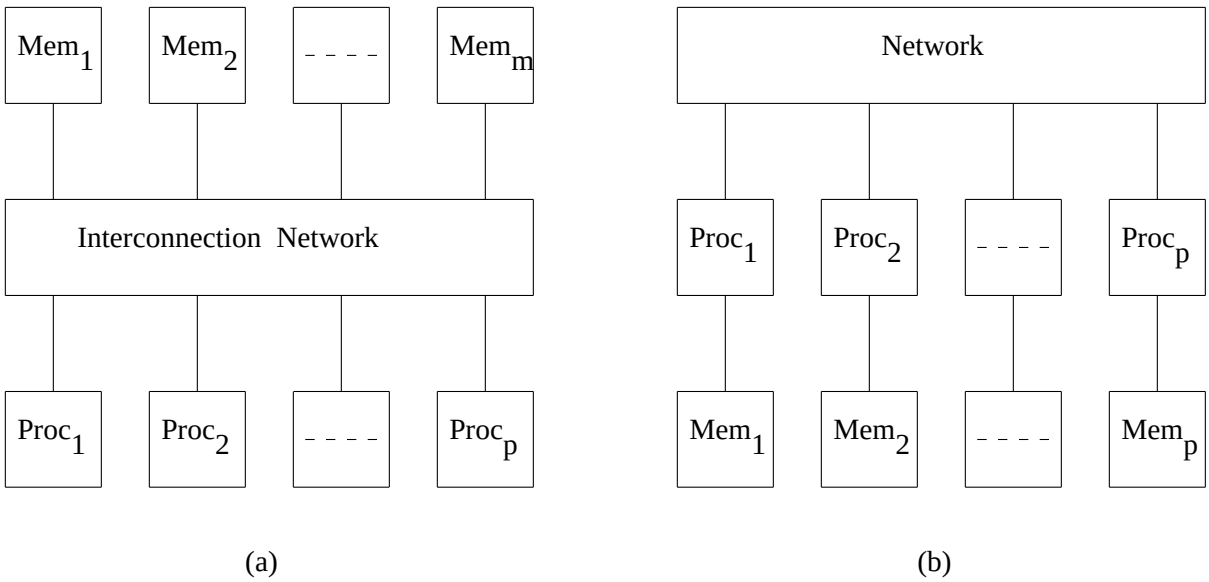


Figure 1.2. (a) Shared Memory and (b) Distributed Memory Multiprocessors

In the HIPERCAD (HIgh PERformance CAD) project, currently ongoing in our group, we are looking at parallel algorithms for CAD problems like Placement, Routing, Design Rule Checking, Extraction, Test generation, Fault simulation, Timing simulation, and Logic synthesis [17]. The goal of this project is to develop an integrated set of efficient tools for different CAD problems on MIMD multiprocessors (both shared memory and distributed memory), providing a very high performance CAD environment. In Fig. 1.3 we provide an overview of the project.

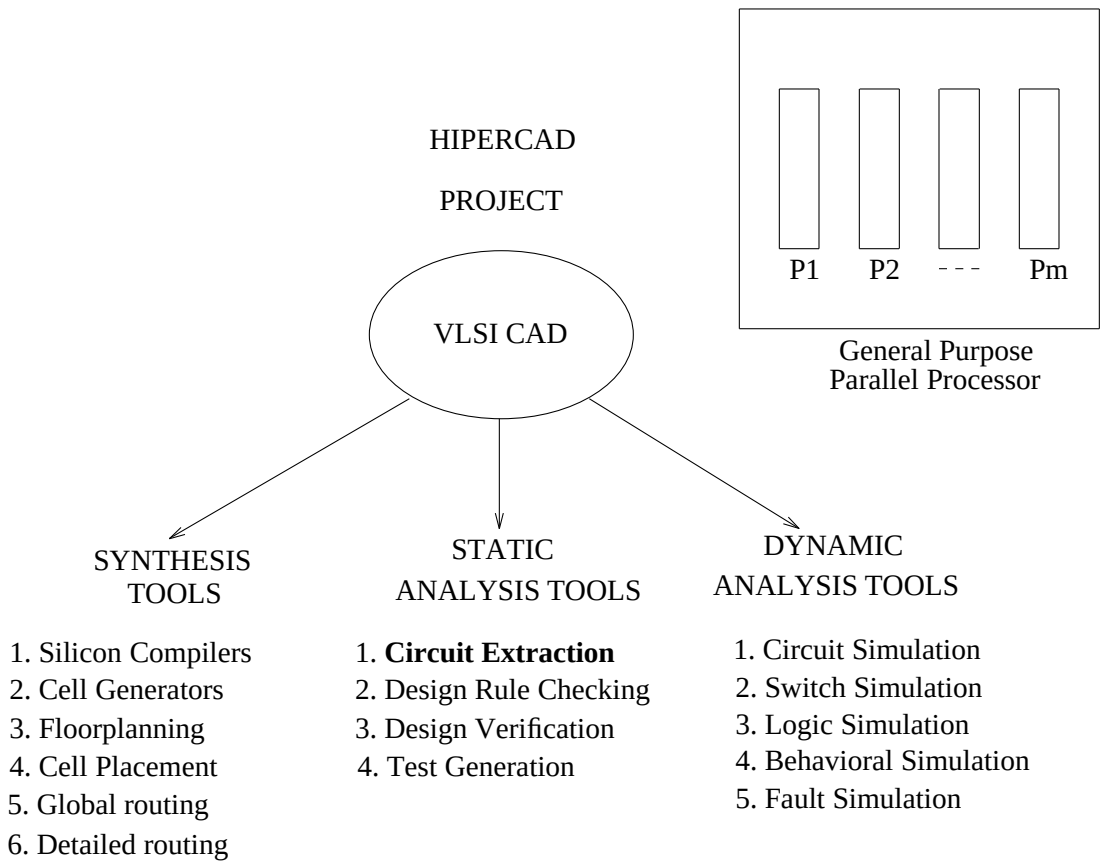


Figure 1.3. HIPERCAD (HIgh PERformance CAD) project

1.2. Circuit Extraction Problem

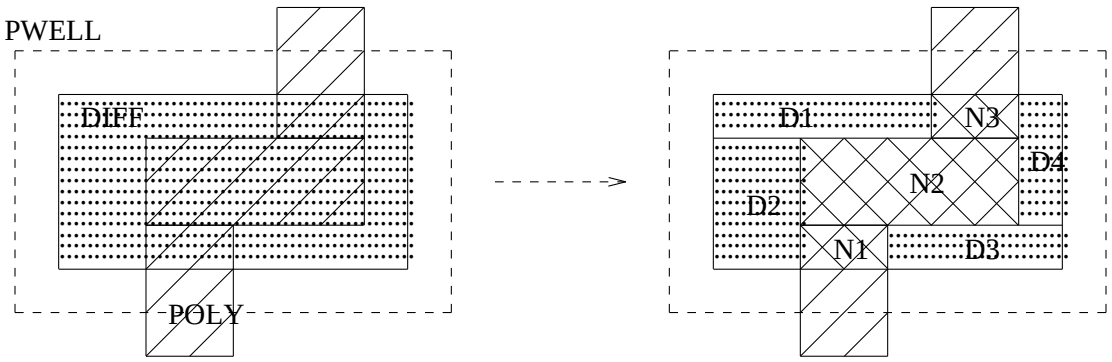
A VLSI layout is typically described as a collection of rectangles in different mask levels. Taking a VLSI layout as input, a circuit extractor determines the circuit connectivity and estimates various electrical parameters such as the resistances of lines, capacitances of nodes, and dimensions of devices. There are two phases in circuit extraction. The first phase involves performing boolean mask manipulations, identification of devices, and 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 [18, 19, 20, 21, 22, 23]. With the increasing size of VLSI layouts, there is a need for researching faster algorithms for performing circuit extraction.

1.2.1. Netlist Extraction

The netlist extraction phase involves performing boolean mask manipulations, determination of devices and determination of electrical connected regions (called nets). Boolean mask manipulations are performed on different layers to derive new layers, as specified in a technology file. The new layers include layers for the different devices. For example, in CMOS technology, N type transistors are obtained by intersecting 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. 1.4.

The rectangles in the device layers are grouped into maximal connected groups, which form the devices. The rectangles in other layers are grouped into maximal electrically connected sets, which form the nets. The electrical connectivity information for the different layers is specified in a a technology file. This gives the layers that electrically connect on overlap. For example, in CMOS technology the metal and contact layers electrically connect on overlap. Similarly, the diffusion and



Here N1, N2, N3 are N type transistor rectangles (P type if Pwell is absent), D1, D2, D3 and D4 are new diffusion rectangles.

Figure 1.4. Identification of device layers in CMOS technology

contact layers electrically connect on overlap. For example, consider a CMOS inverter shown in Fig. 1.5. The device rectangles PT1, PT2, PT3 together form a P type transistor. The device rectangle NT1 forms a N transistor. The rectangles M1, C1, D1 and D2 form a net connected to Vdd. The rectangles M2, C2, and D3 form a net connected to GND. The rectangles P1, P2, P3, and P4 form the input net. The rectangles D4, D5, D6, C3, C4, M3, and M4 form the output net.

Many approaches are known for netlist extraction. A very efficient approach to netlist extraction, used by many extractors [19, 20, 18], uses the "scan line paradigm" of computational geometry. The basic idea in the approach is to sweep an imaginary line in the plane from one end to the other. At any time we are interested only in the geometry that intersects the scan line. At each stop of the scan line we do some problem dependent processing, and advance the scan line. The important aspect of a scan line algorithm is the data structure for storing the geometry intersecting the scan line.

In applying this method to netlist extraction, we first sort the rectangles in a circuit along one dimension, say the x dimension without any loss of generality. The imaginary scan line is then assumed to move from left to right, as shown in Fig. 1.6. At any stop, the rectangles whose left end point touches the scan line are taken from the sorted list, and added to the scan line data structure.

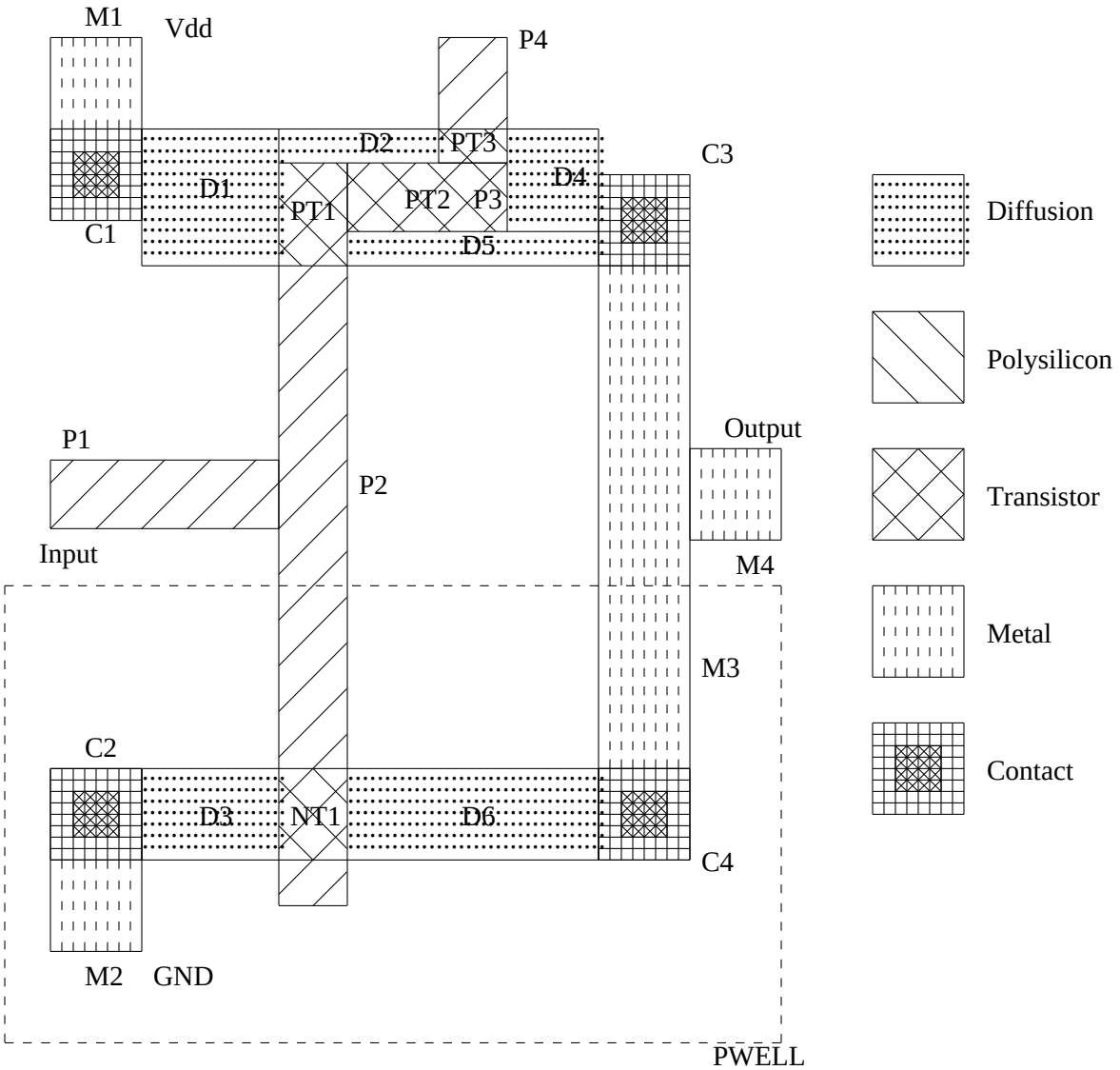


Figure 1.5. Inverter Circuit

For each rectangle *R* added, the rectangles in the scan line data structure that overlap or abut it are determined. This is used to perform the different boolean mask operations. If an abutting or overlapping rectangle *O* is in a layer that electrically connects to *R*'s layer, then *R* and *O* are determined to be in the same net. Similarly, if *R* and *O* together form a device, then the rectangle corresponding to the device is constructed. Connecting device rectangles are grouped into devices in a manner similar to the method used for obtaining nets. When a scan line is advanced, all those rectangles whose right

end point has been past by the scan line are deleted from the scan line data structure. For example, in Fig. 1.6, the rectangles B, G are added to the scan line data structure. The rectangles E, C, D are already in the scan line data structure. When B is added it will be determined to be connected to rectangles E and D. Thus B, D, E will be grouped together in the same net. When the scan line is advanced, the rectangle C will be removed from the scan line data structure. The scan line data structure could be organized as a link list, tree or a collection of bins, the latter being the most efficient in practice.

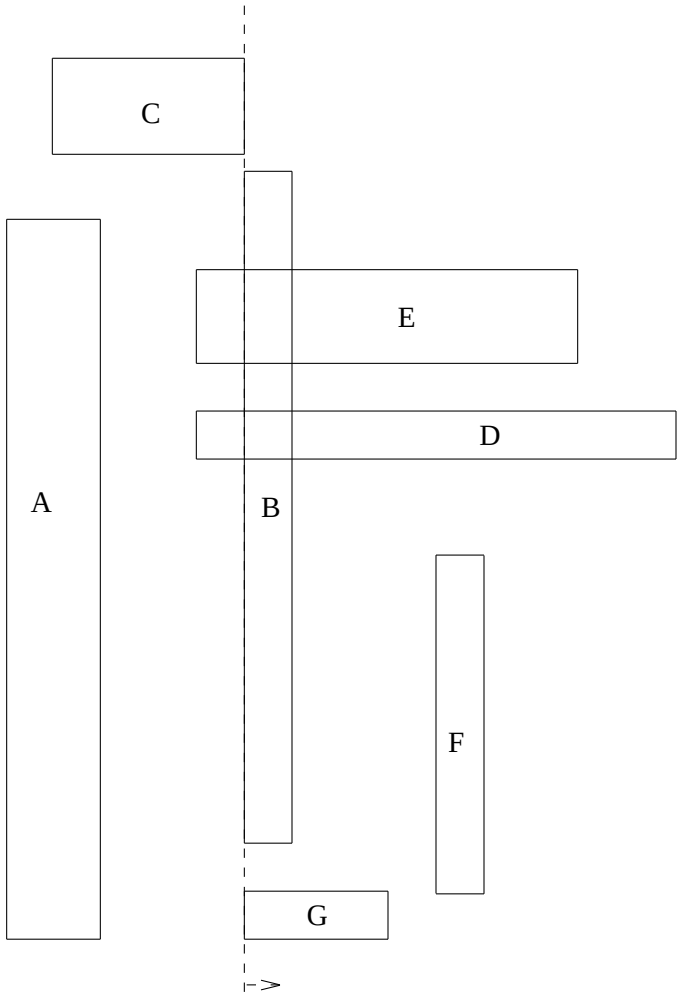


Figure 1.6. The scan line moving from left to right

The output of the netlist extraction phase for a circuit is a list of devices and a list of nets. A device is described as a collection of device rectangles. For each device, the output contains information about the nets connecting to the different terminals of the device. A net is described as a collection of rectangles. For each net, the output contains information about the devices that connect to the net.

1.2.2. Parameter Extraction

The parameter extraction phase involves device size extraction, parasitic capacitance extraction and resistance extraction of nets. This is the most time consuming part in circuit extraction. There are different models for parameter extraction with varying accuracy and computational requirements. The accurate models require more computational time. In this section, we briefly outline the steps in parameter extraction, giving particular reference to two models, namely *MEXTRA* [18] and *HPEX* [23].

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. This step is illustrated in Fig. 1.7.

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

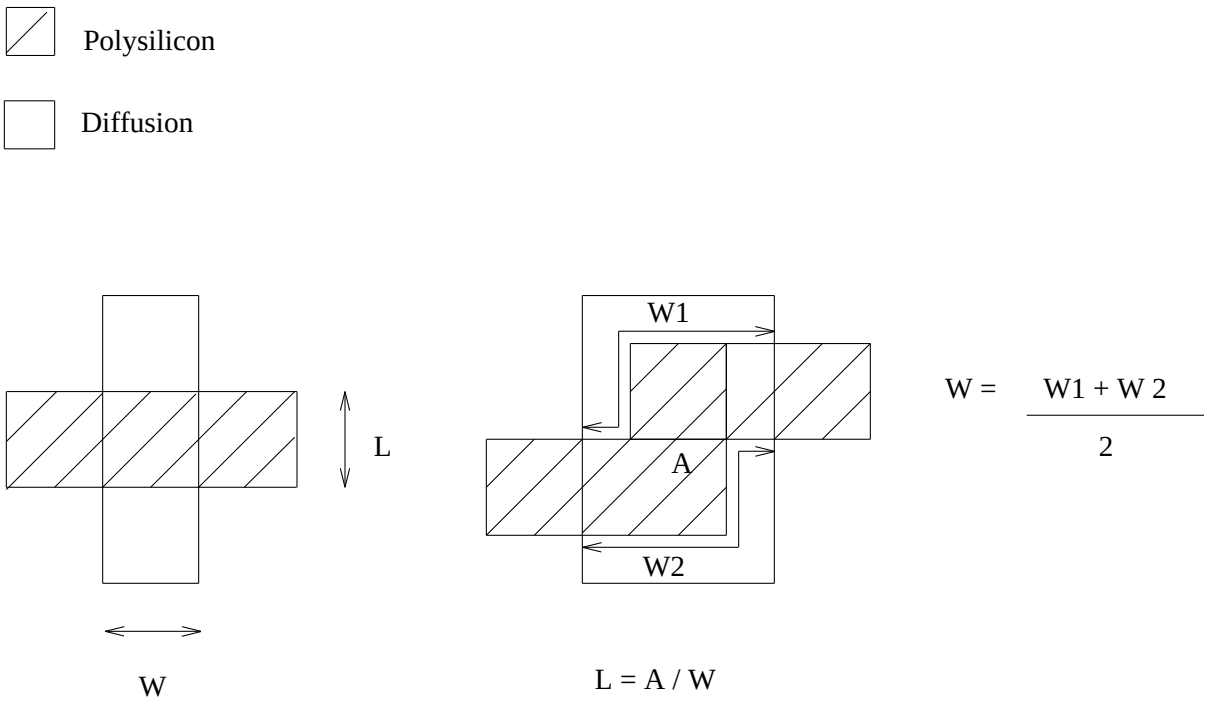


Figure 1.7. Device size extraction for CMOS technology

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 [24]. Some tile based circuit extractors [21] also determine inter nodal capacitances between two long parallel edges close together, usually in the same layer. These are called parallel capacitances. The different types of capacitances are illustrated in Fig. 1.8. In *MEXTRA* only the substrate capacitance is computed for each net by the above formula. However, in *HPEX* both substrate capacitance and resistance are 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

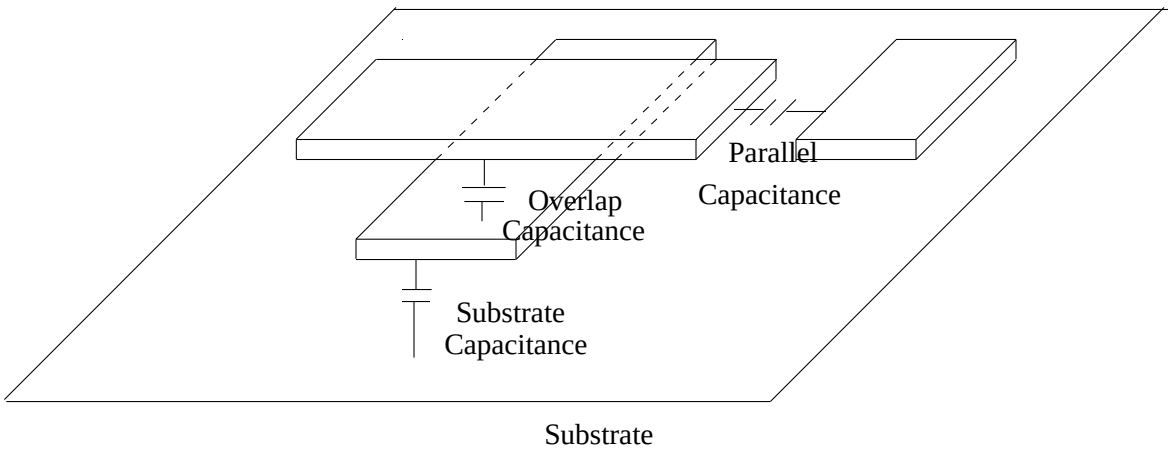


Figure 1.8. Different types of capacitances

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 [20, 23, 25].

We now describe the resistance capacitance model used in *HPEX* [23]. 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 a 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. 1.9.

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. 1.10. 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

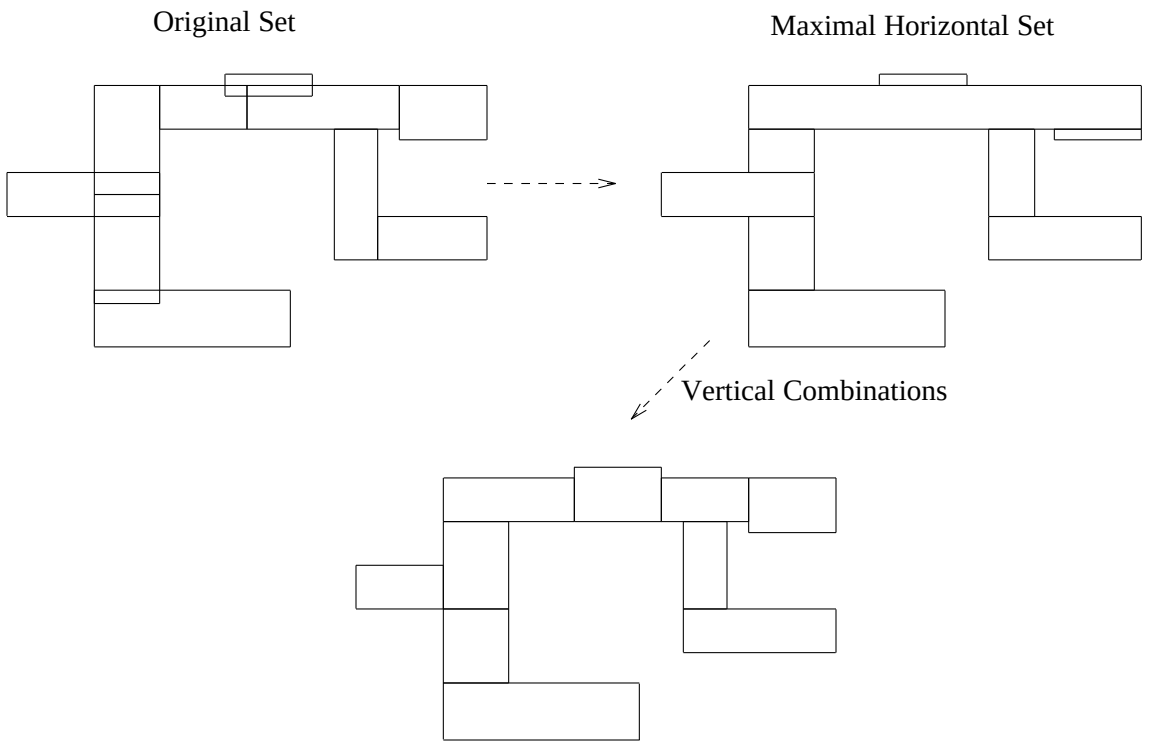


Figure 1.9. The first two steps in resistance extraction

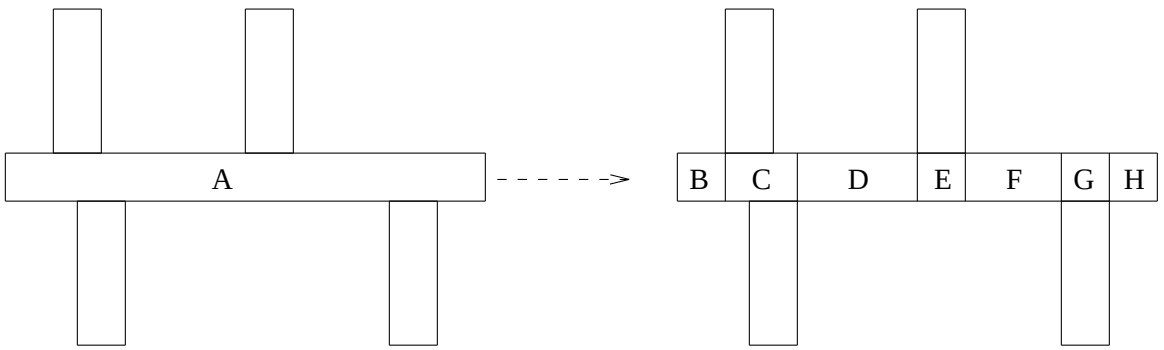


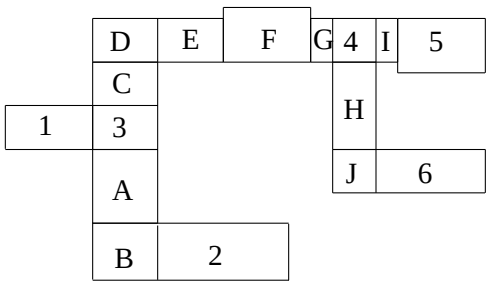
Figure 1.10. The rectangle decomposition of *A*

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. 1.9 in Fig. 1.11.

The next step is to perform the resistances calculations for the branches. For this step, analytical formulae [23] 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. An example RC network is shown in Fig. 1.12. 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* [26]. 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* [26] for



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	

Figure 1.11. Rectangle decomposition and branch creation

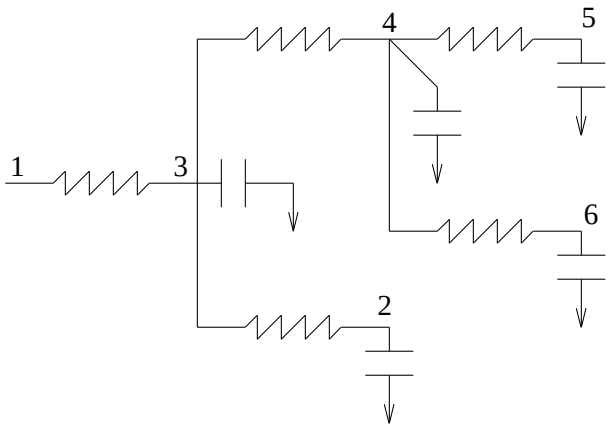


Figure 1.12. An example RC network

this purpose.

1.3. Thesis Overview

The organization of the thesis is as follows. In Chapter 2, we describe parallel algorithms for circuit extraction on distributed and shared memory MIMD multiprocessors. Towards this goal, we formulate two general problems in parallel processing. In Chapter 3, we describe the first problem, called the geometric connected components labeling problem, and present different algorithms for the problem. In Chapter 4, we look at the second problem, which consists of maintaining a recursively partitioned data structure on a multiprocessor, and present different algorithms for the problem. In Chapter 5, we take a general look at the use of hierarchical methods for speeding various problems in computer-aided design, and approaches to combine the benefits of hierarchy and parallelism. Towards this goal, we formulate two general problems in processor scheduling. We specifically illustrate the approach for circuit extraction. In the following two chapters, we present approximate algorithms for the two scheduling problems. Finally, in Chapter 8 we give the conclusions.

CHAPTER 2.

PARALLEL ALGORITHMS FOR VLSI CIRCUIT EXTRACTION

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. The issues in parallel circuit extraction, therefore, are to design efficient algorithms that can be used in the partitioning and combining steps. In this chapter, we discuss parallel algorithms for circuit extraction, on shared and distributed memory MIMD multiprocessors, that operate on a flattened description of a Manhattan layout. These algorithms will be referred by a common name *PACE*. For the case of distributed memory MIMD multiprocessors, we describe the issues involved by illustrating it for a hypercube. 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. The processors may in addition be connected to a single processor, called the host, which is used for I/O operations.

The organization of this chapter is as follows. In Section 2.1, we discuss the different partitioning strategies used in *PACE*. In Section 2.2, we will describe the netlist extraction phase in *PACE*. In Section 2.3, we will describe the parameter extraction phase in *PACE*. In Section 2.4, we analyze the time complexity of *PACE*. In Section 2.5, we describe the implementation of *PACE* on an Intel iPSC2 hypercube, and on an Encore 510 multimax, which is a shared memory multiprocessor.

2.1. Circuit Partitioning and Mapping

2.1.1. Circuit Partitioning

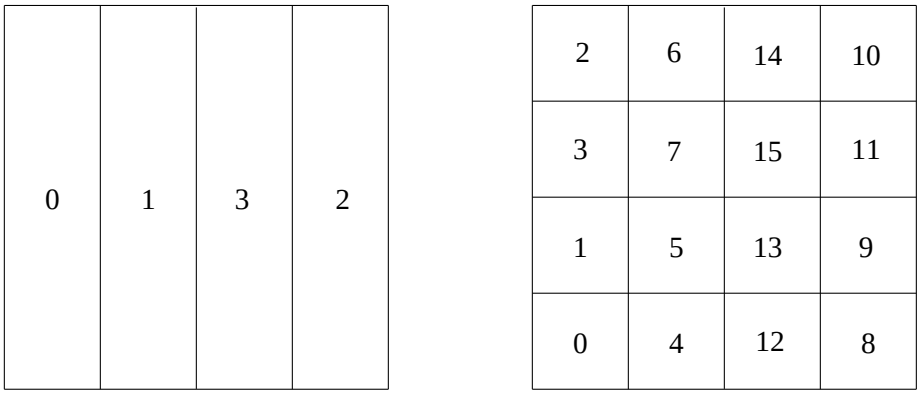
There are four partitioning strategies implemented in the *PACE* algorithm. These are classified as area based and point based methods. We discuss these strategies assuming there are $P = 2^d$ processors,

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. 2.1.

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



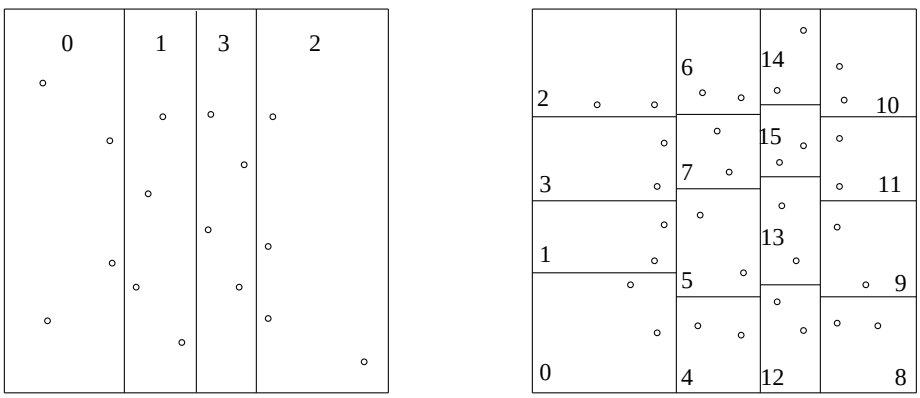
(a) (b)

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

Figure 2.1. Area based (a) Slice Partitioning (b) Rectangular Partitioning

determined as before. The two point based strategies are shown in Fig. 2.2.

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



(a) (b)

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

Figure 2.2. Point based (a) Slice Partitioning (b) Rectangular Partitioning

$j=0$ 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 $R'(t)$ to be the region obtained by expanding $R(t)$ 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].

2.1.2. Implementation on a Hypercube Multiprocessor

We now discuss how the circuit partitioning strategies are implemented on a hypercube multiprocessor. 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 [27]. The region assignment is illustrated in Fig. 2.1 and Fig. 2.2. 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 processor 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 recursively partitioned data structure. This is described in Chapter 4. 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.

2.1.3. 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. 2.3.

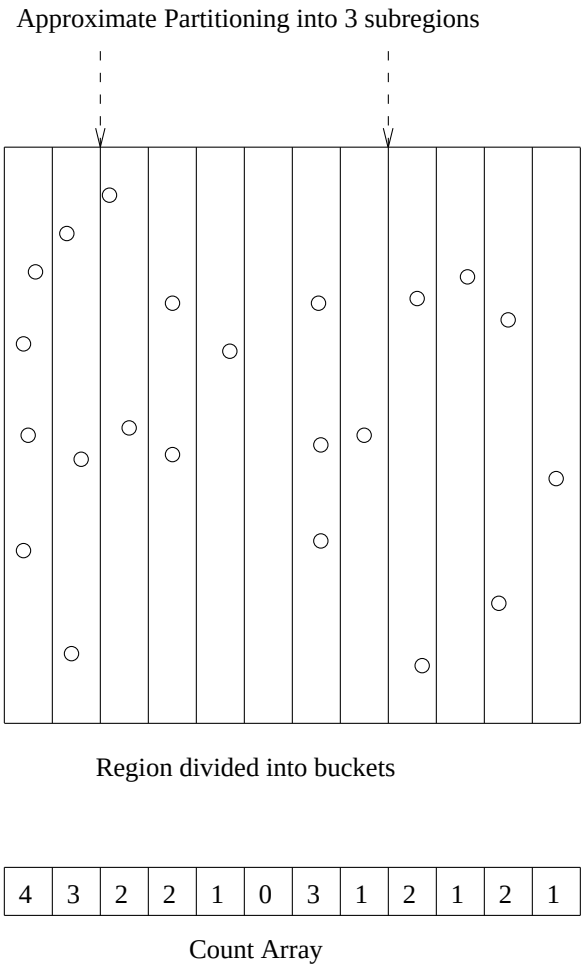


Figure 2.3. Implementing approximate point based partitioning

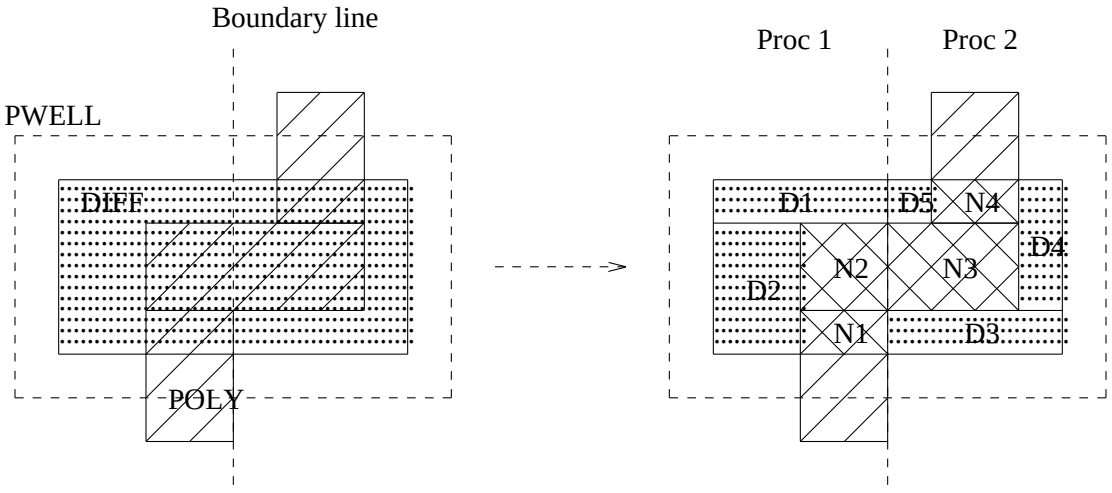
The circuit region is divided into different bins in the x dimension. The number of points that fall in each bin is determined and stored in an array. The entries in the array can then be used to determine the partitioning lines along the x dimension. The process can then be repeated in each of the resulting sub regions, by the appropriate processors, to determine the partitioning lines along the y dimension.

2.2. Parallel Netlist Extraction

In both the methods of partitioning, each processor t gets all those rectangles that touch or overlap $R(t)$. Before starting the net list extraction, these rectangles are cut so that the processor retains only the portion within $R'(t)$ (the expanded version of $R(t)$). It should be noted that the regions $R'(t)$ for different processors t 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 t.

P1: Boolean mask manipulations are performed locally on the different layers to derive new layers, including the device layers. This step is illustrated in Fig. 2.4, 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



After step P1, N1-N2 are device rectangles in processor 1, N3-N4 are device rectangles in processor 2, D1-D2 are diffusion rectangles in processor 1 and D3-D5 are diffusion rectangles in processor 2.

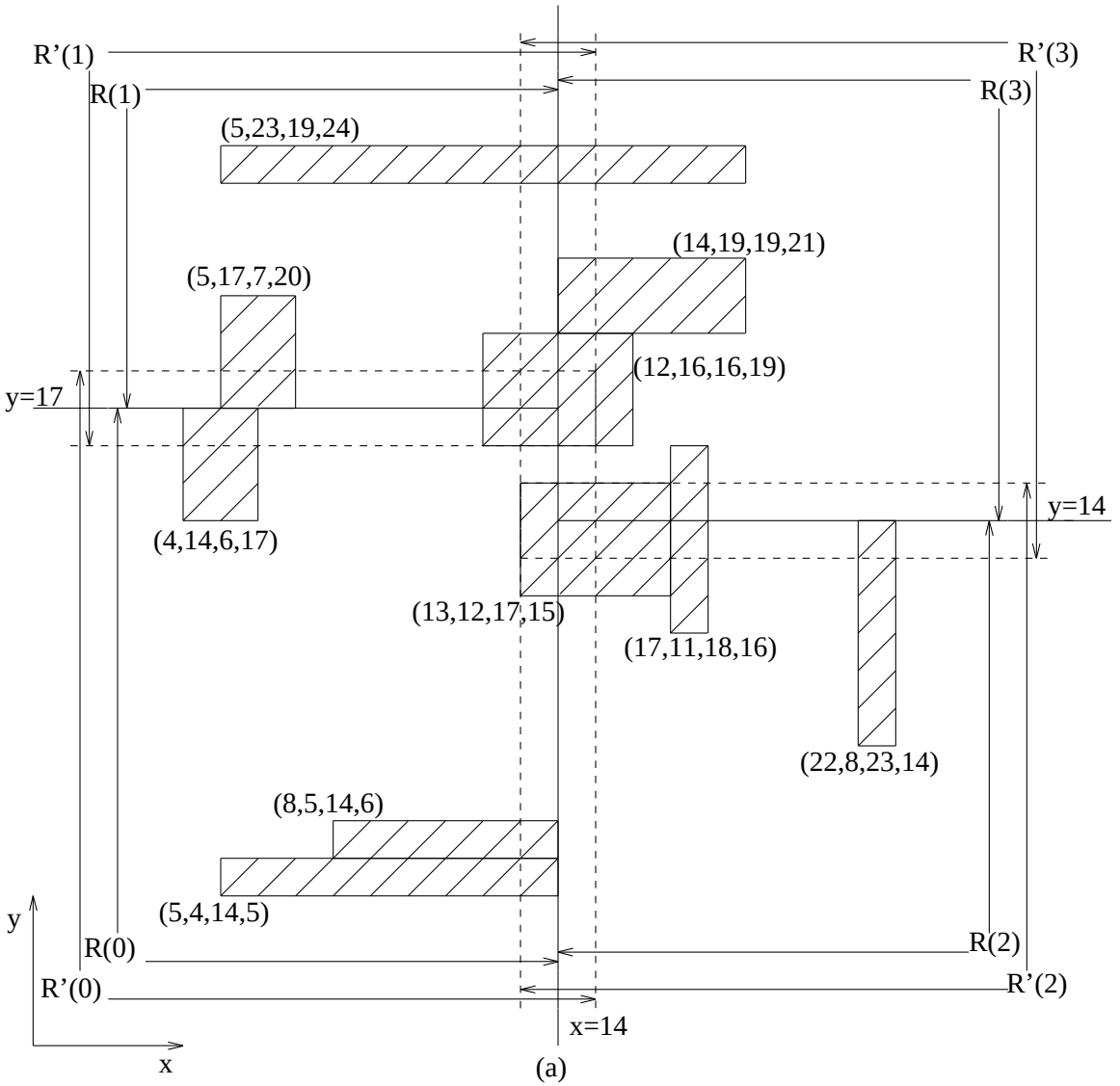
Figure 2.4. Parallel device identification for CMOS technology

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 $R(t)$ 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. 2.5, 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 $R(t)$ 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. 2.5. 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. 2.5 that the maximal border segments at processor 0 for the top



	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)

Figure 2.5. (a) Set of rectangles (b) Maximal border segments after step P4

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 $R(t)=(x_{\min},y_{\min},x_{\max},y_{\max})$. In case of point based partitioning, for each of the x borders the maximal border segments (a,b) that satisfy $a \leq y_{\min}$ are deleted. The other segments are left in the original order. For example in Fig. 2.5, 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 $G(i,m)$, $0 \leq m < 2^{d-k}$. Referring to Fig. 2.5, $rightlist(0)$ = {(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. 2.5, $L(1)=R(0)$ = {(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. 2.5, $rightlist(0)$ 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 $0 < i < 2^k$.

P6: The lists constructed in P4, P5, for each of the borders, and the *LCS* sets that touch a border are passed to a geometric connected components algorithm. This algorithm determines for each of the *LCS* sets a unique label which identifies its global connected component (*GCS*). The unique label of a *GCS* set is chosen by the algorithm from the labels assigned to its *LCS* components in step P2. In next chapter, we describe different algorithms for the geometric connected com-

ponents problem. 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.

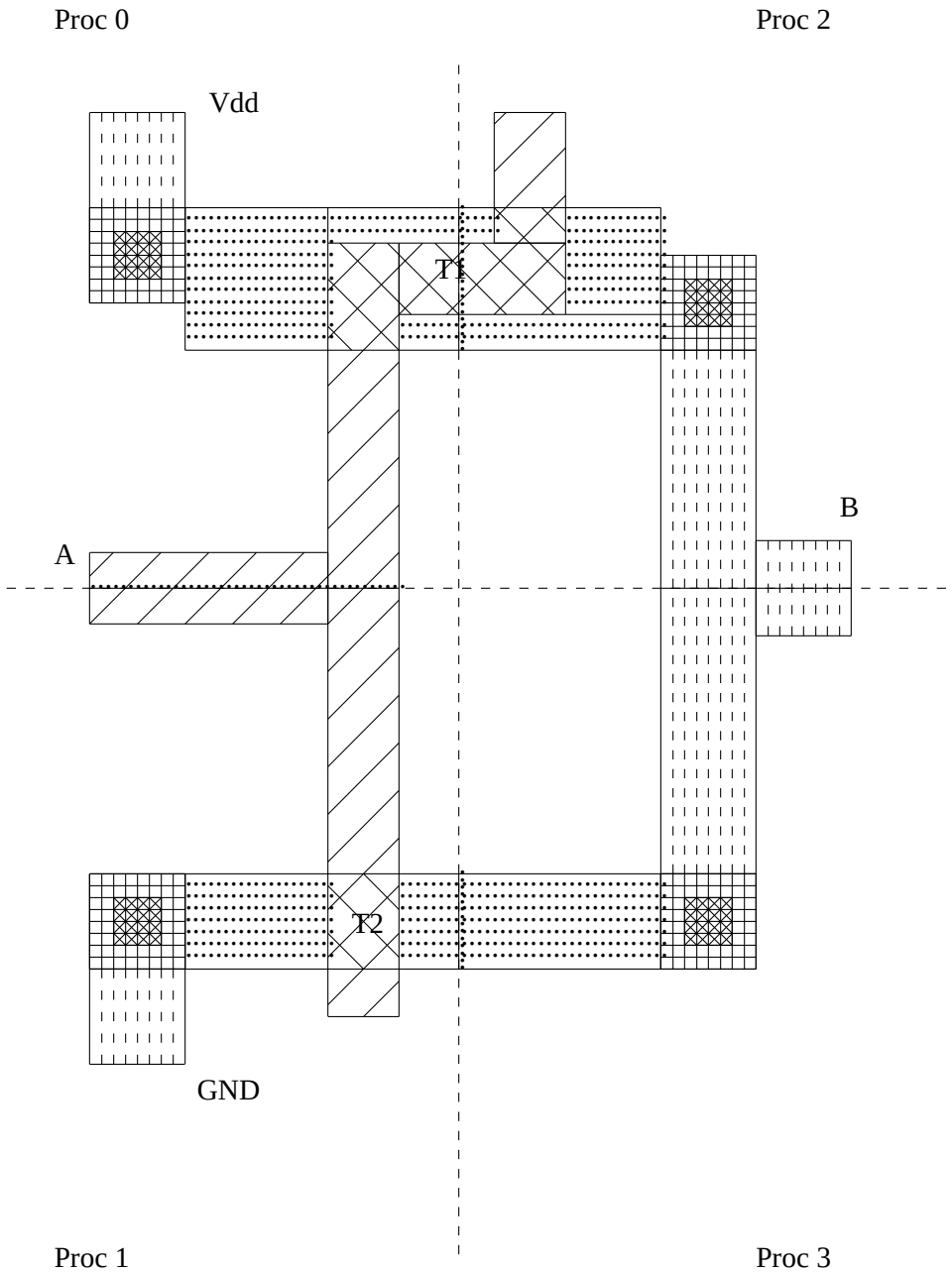
The local nets, devices for a four processor CMOS example is shown in Fig. 2.6. Their final labels are shown in brackets in a table.

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. 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.

2.3. Parallel Parameter Extraction

In this section, we briefly describe how the two models, namely *MEXTRA* [18] and *HPEX* [23], are interfaced 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 time consuming models.

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



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)	_____

Figure 2.6. Determination of global nets, devices

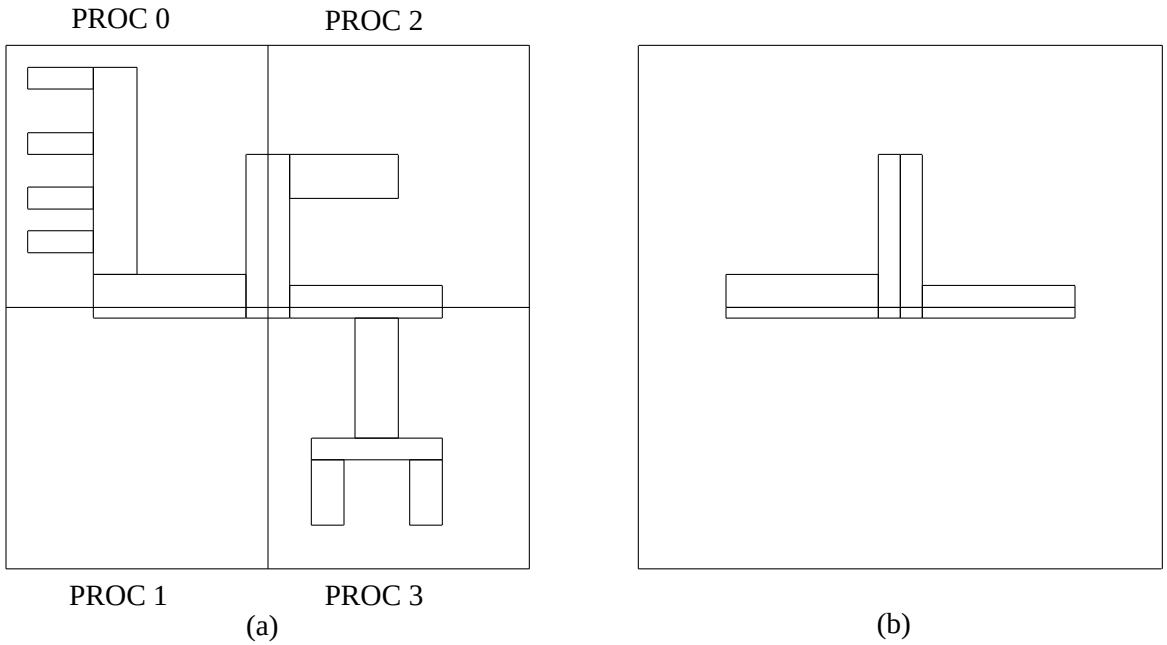
complete the information for the global device. The processor can then extract the device. All communication between two processors in the above step can be done in a single message, since the information that needs to be sent is completely determined before the step. Thus the communication can be done 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 geometric connected components 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. 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. 2.7. The communication in this step is done in an efficient manner, as in the previous step for extraction of devices.



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

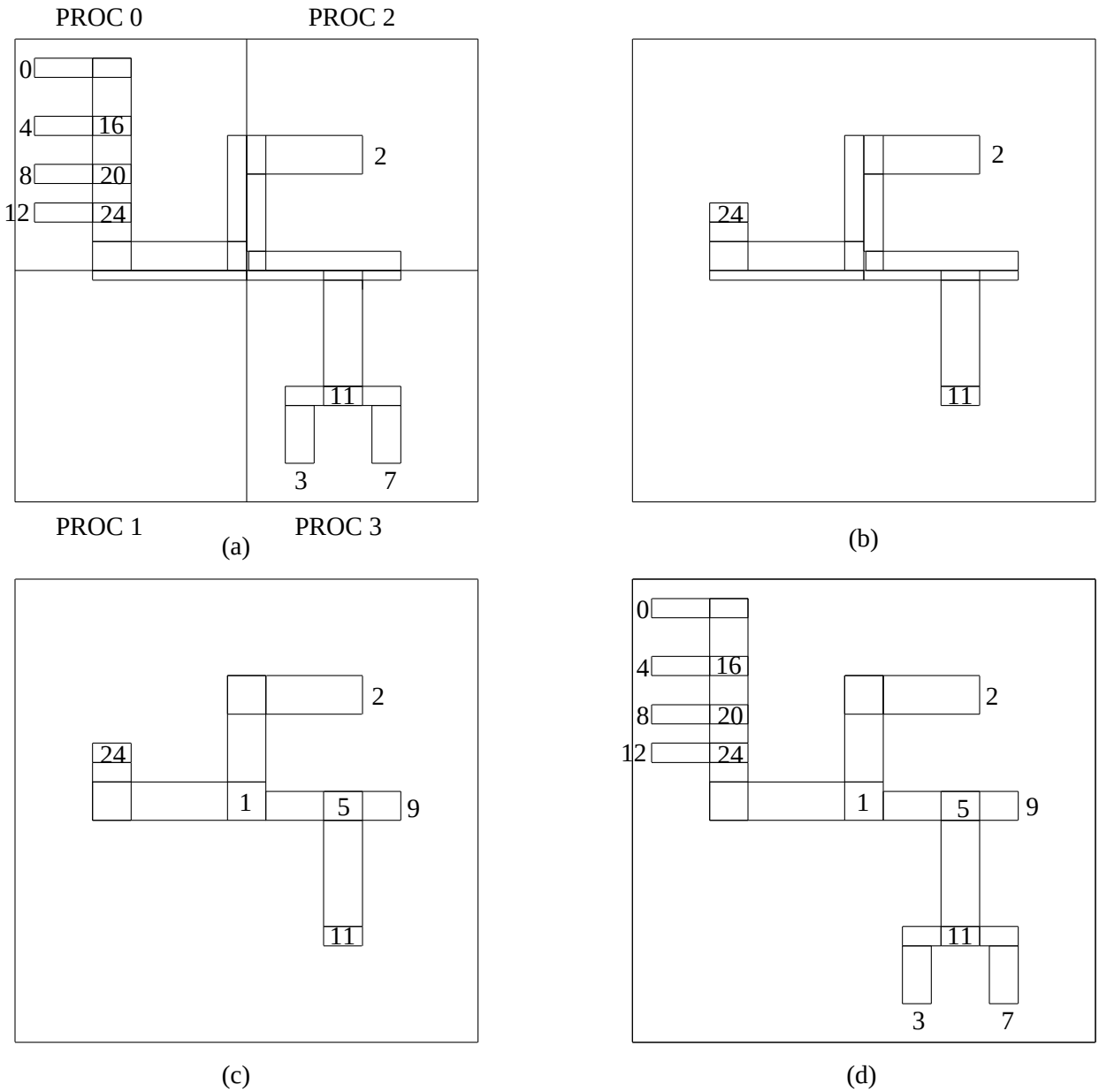
Figure 2.7. Interfacing *MEXTRA* model for extraction

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. 1.9, and the rectangle decomposition step of Fig. 1.10, 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. 2.8(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 end points, 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. 2.8(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. 1.9, and the rectangle decomposition step of Fig. 1.10. 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. 2.8(c). The new knots determined are 1, 5, and 9. The complete result is shown in Fig. 2.8(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. 2.8(c). Note that the branch 2 to 1 of Fig. 2.8(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 capaci-



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

Figure 2.8. Interfacing *HPEX* model for extraction

tance 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. 2.9, 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. 2.9(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 [26], 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.

(a)

(b)

Figure 2.9. (a) The example circuit (b) Rectangles processed by processor 0

2.4. 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.

2.4.1. 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 parallelizing of the netlist phase. The geometric connected component algorithm is the most dominant among these steps. The time taken by the simplest of the geometric connected components algorithms, to be described in the next chapter, is bounded by $dt_\alpha + dc_2\sqrt{n}\,t_\beta$, where t_α is the startup time of a message, t_β is the time taken to send a byte.

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

$$S_P = \frac{T_1}{T_P} = \frac{P}{1+(\frac{P}{C}\frac{D}{})(\frac{1}{\sqrt{n}})}.$$

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

2.4.2. 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 geometric connected components algorithm. This is expected to take $O(\sqrt{n/P}\log n)$. The final speedup expression derived in this case is the same as in the previous section.

2.5. 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.

2.5.1. Results for a Hypercube

In Table 2.1, 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 2.2, 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

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

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 2.1. 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.

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

In Fig. 2.10, 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. 2.11, 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 the one in Fig. 2.10. The graph shows that the computation and communication overhead of slice partitioning is greater than that of rectangular partitioning.

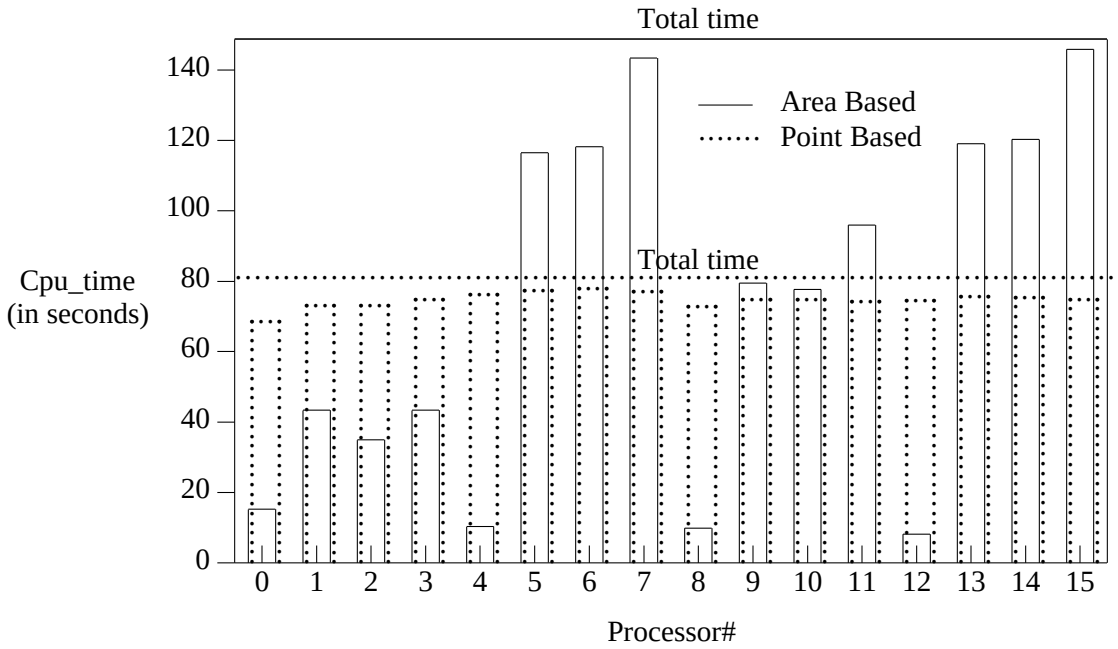


Figure 2.10. *PACE* (*HPEX* model) with rectangular partitioning (area versus point)

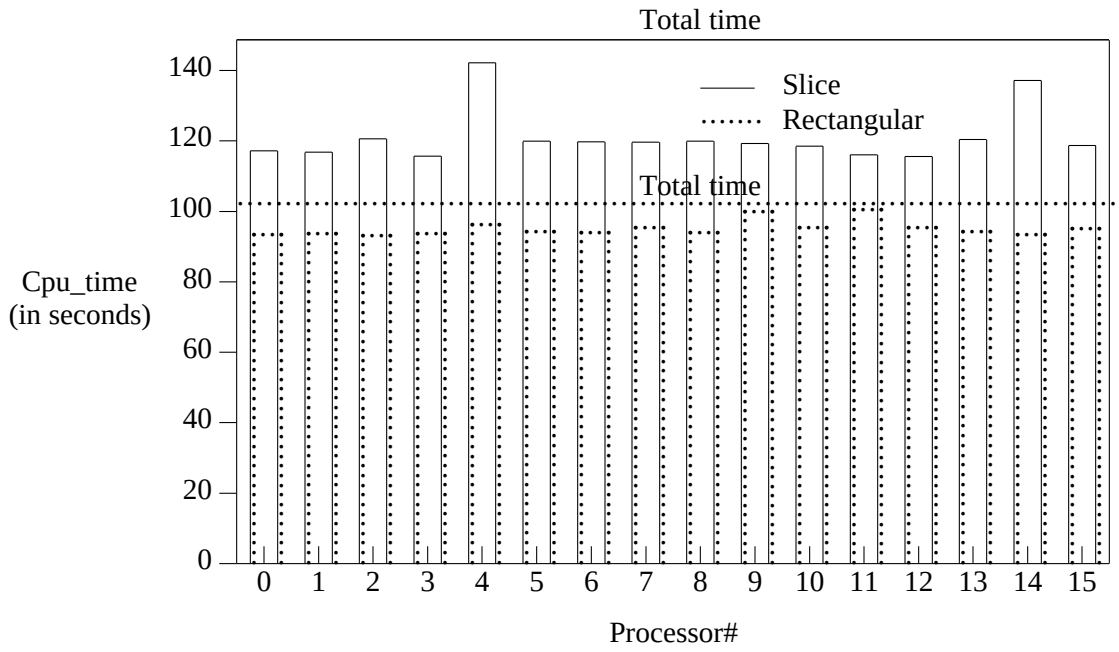


Figure 2.11. *PACE* (*HPEX* model) with point based partitioning (slice versus rectangular)

2.5.2. 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 2.3, we give the time taken by the algorithm for 1 and 8 processors. The speedup obtained is less than expected, because of memory contention.

2.5.3. Comparison on an Uniprocessor

In Table 2.4, 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

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

Table 2.3. Performance of *PACE* with *HPEX* model on Encore 510 multimax

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.

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

Table 2.4. Performance of *PACE* sequential implementation on Sun 3/280 and Sun 4/280

2.6. Summary

In this chapter, 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.

CHAPTER 3.

GEOMETRIC CONNECTED COMPONENT LABELING

In this chapter, we will describe parallel algorithms for the geometric connected component problem, that arises as a sub problem in the design of a parallel extractor. In Section 3.1, we formally define the problem. In Section 3.2, we review the previous work done on related problems. In Section 3.3, we present new algorithms for the problem on a hypercube and analyze their performance. In Section 3.4, we present an algorithm for the problem on a shared memory multiprocessor.

3.1. Geometric Connected Component Labeling Problem

The Geometric Connected Component Labeling (*GCCL*) problem was described in the previous chapter. The inputs to the *GCCL* problem at a processor t consists of:

(1) The locally connected sets at the processor, which are referred to as *LCS* sets. Of all the *LCS* sets at the processor t, we are only interested in those that touch a border of $R(t)$. The *LCS* sets at the processors have been assigned labels that are unique across processors.

(2) A list of border segments for each of the borders. Each border segment is associated with an *LCS* set that touches the border. For area based partitioning, the list of border segments for a border at the processor will correspond exactly to the list of border segments for the same border at the processor at the other end of the border. For point based partitioning, this is true only for y borders. However, for an x border the lists $leftlist$ and $rightlist$ of border segments, as defined in the previous chapter, will correspond to each other.

The *GCCL* problem involves determining, for each of the *LCS* sets, a label that denotes its globally connected set. The globally connected sets are referred to as *GCS* sets. All *LCS* sets at different

processors that are part of a *GCS* set must get the same label by the *GCCL* algorithm. It must be observed that solution to the *GCCL* problem is possible without the knowledge about the elements in the *LCS* sets. In other words, it suffices to represent the *LCS* sets by their labels alone for the *GCCL* problem.

3.2. Previous Work

A problem, closely related to the *GCCL* problem, is the image component labeling problem. The input to the image component labeling problem consists of a $P^{1/2} \times P^{1/2}$ grid of binary pixels. Two 1 pixels are said to be adjacent if they share a common horizontal or vertical edge. Two 1 pixels are said to be connected if there is a path of adjacent 1 pixels that connects one to another. The image component labeling problem consists of assigning labels to the 1 valued pixels so that two 1 valued pixels get the same label if and only if they are connected to each other. An example image grid is shown in Fig. 3.1. The different components of the image are marked in the figure.

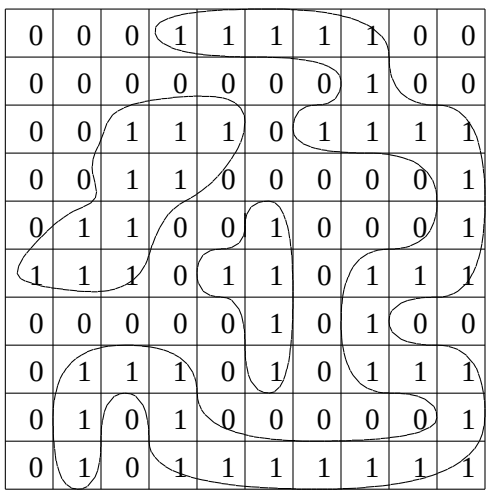


Figure 3.1. Example grid

The image component labeling problem is a special case of the graph connected component labeling problem. Given a graph, the graph connected component labeling problem involves labeling each vertex of the graph so that two vertices of the graph get the same label if and only if they are connected by a path in the graph. Given an input I to the image component labeling problem, one can construct a graph G as follows. The vertices of G correspond to the 1 valued pixels of the grid. Two vertices in G are connected by an edge if their corresponding 1 valued pixels are adjacent. The connected components of G correspond exactly to the image connected components of I.

In [28], Shiloach and Vishkin present an algorithm for the graph connected components labeling problem on a CRCW PRAM. A PRAM is a parallel processing model, in which processors are connected to a common memory. In a time cycle, each of the processors reads from a memory location, performs an operation, and writes back to a memory location. A CRCW PRAM is the most powerful PRAM model, as it allows simultaneous reads and writes to the same memory location. In case many processors write into the same memory location, one of them will succeed.

A brief outline of the Shiloach and Vishkin's algorithm follows. During the course of the algorithm, for each vertex i a value $D(i)$ (initially set to i) is maintained. The $(i,D(i))$ pairs can be viewed as directed edges. At any step, this defines a set of rooted trees, with self loops at the root of the trees. All vertices in a rooted tree necessarily belong to the same connected component of the graph. A "short-cut" operation replaces the value $D(i)$ by $D(D(i))$. This operation reduces the height of all the rooted trees. A "hooking" operation hooks the root of a star tree (a rooted tree of height 1) to another vertex, not in its tree, but connected to it in the graph. This is done by looking at edges that connect a member of the star tree to another vertex that is not a member of the same star tree. The algorithm presented by Shiloach and Vishkin uses $v+2e$ processors, where v and e are the number of vertices and edges in the graph respectively. Using $O(\log v)$ "short-cut" and "hooking" operations, it derives a set of rooted stars, one corresponding to each component of the input graph. Since these operations take $O(1)$ time on a CRCW PRAM, the total time taken is $O(\log v)$.

Though a CRCW PRAM is an impractical machine, algorithms designed on it can be translated to algorithms for real machines. For example, a single operation on a CRCW PRAM of size P can be simulated by a P processor hypercube or shuffle-exchange network in time $O(\log^2 P)$ [29]. The simulation algorithm consists of an $O(\log^2 P)$ bitonic sort followed by a number of $O(\log P)$ subroutines. By replacing the bitonic sort step by a radix sort step, it has been shown [29] that an operation of a P processor CRCW PRAM can be simulated by a $P^{1+1/K}$ processor hypercube, for a constant K, in time $O(\log P)$. Similarly, an operation of a P processor CRCW PRAM can be simulated by a $P^{1/2} \times P^{1/2}$ mesh in time $O(P^{1/2})$ [29]. Again, the simulation algorithm consists of a sorting step taking $O(P^{1/2})$ time followed by a number of routines each taking $O(P^{1/2})$ time.

Using the Shiloach and Vishkin's algorithm and the relationship between image and graph connected component labeling problems, one can design algorithms for the image component labeling problem on realistic machines by simulating a CRCW PRAM. Thus one could obtain an $O(\log^3 P)$ algorithm for the problem on a P processor hypercube or shuffle-exchange network, and an $O(P^{1/2}\log P)$ algorithm for the problem on a $P^{1/2} \times P^{1/2}$ mesh.

The above results can be improved by exploiting the special properties of the image component labeling problem. Nassimi and Sahni [30] have presented an $O(P^{1/2})$ algorithm for the problem on a two dimensional $P^{1/2} \times P^{1/2}$ mesh of processors. Cypher, Sanz and Snyder [31] have presented an $O(\log^2 P)$ algorithm on a P processor hypercube or shuffle-exchange network.

The essential idea in the new algorithms is to use the divide and conquer technique. To solve the problem for a $P^{1/2} \times P^{1/2}$ grid, we first recursively solve the problem for $P^{1/2}$ sub grids of size $P^{1/4} \times P^{1/4}$, in parallel. This is illustrated in Fig. 3.2. Subsequently, the connecting image components in different sub grids have to be identified and assigned the same label. This can be formulated as a graph connected components problem with $O(P^{3/4})$ vertices and edges. The reduction in the size of input to the graph connected components sub problem is the key idea in the improved algorithms. In case of a hypercube, this problem can be solved using P processors in time $O(\log^2 P)$,

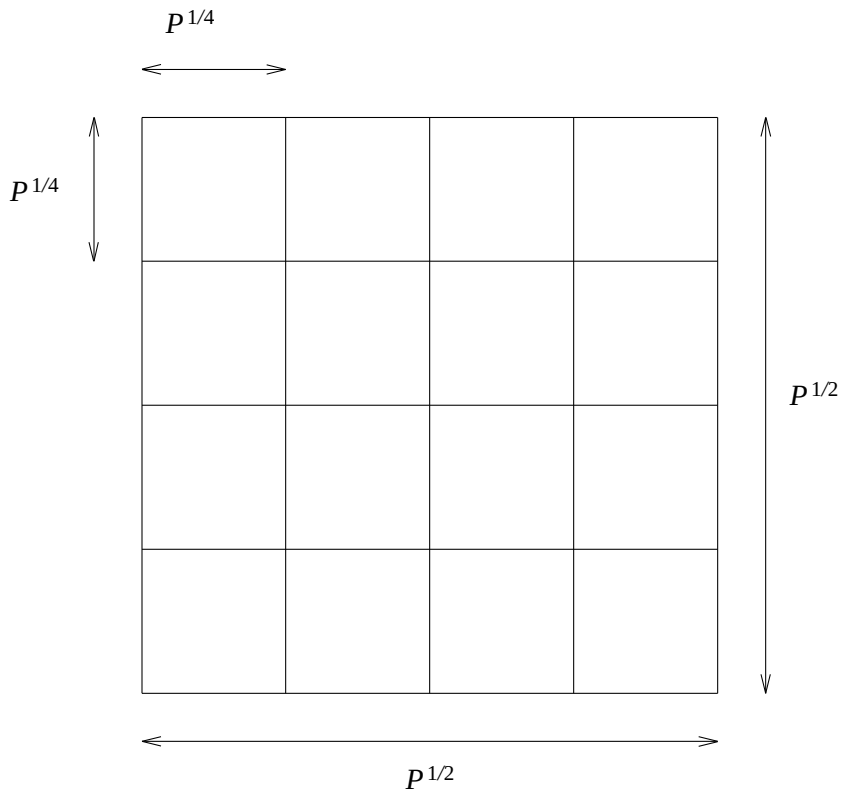


Figure 3.2. Applying divide and conquer

using the radix-sort based simulation of a CRCW PRAM. In case of a mesh, this can be solved in time $O(P^{1/2}+P^{3/4}\log P)$. The first term is due to overhead of setting up the graph connected components sub problem. The analysis of the new algorithms can be obtained by solving the appropriate recurrence equation.

Algorithms for the image component labeling problem, that were described above, can be adapted to solve the *GCCL* problem. However, these algorithms are quite complicated for a practical implementation. Even if these algorithms are implemented, it may not be efficient due to the large constant multiplier, that is ignored by the big-oh expression. In the following sections, we present some practical algorithms for the *GCCL* problem on hypercubes and shared memory multiprocessors, that were implemented as a part of the circuit extractor *PACE*.

3.3. Algorithms for a Hypercube Multiprocessor

In this section, we describe algorithms for the *GCCL* problem on a hypercube multiprocessor. Let the dimension of the hypercube be d. We assume that the circuit domain is divided into 2^{kx} strips in the the x dimension, and 2^{d-kx} strips in the y dimension. The partitioning method could either be area based or point based. The mapping of regions to processors is as discussed in the Chapter 2.

3.3.1. Quad Algorithm

In this algorithm, there are d stages of message passing, where d is the dimension of the hypercube. 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 domain. Initially, a processor knows only about its region. After d stages a processor has knowledge about what it needs to know in the entire domain.

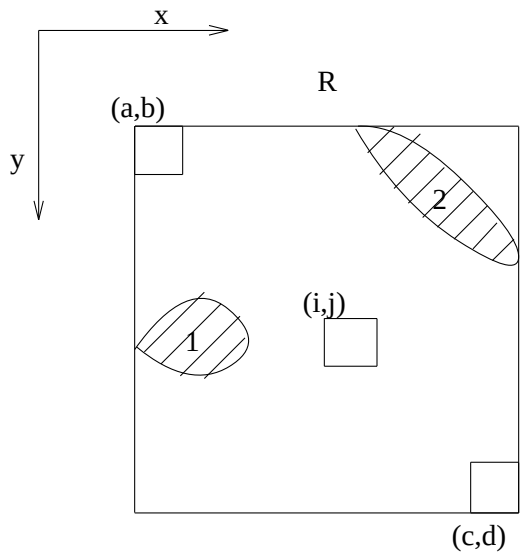
Before each stage of the algorithm, a processor is associated with a rectangular region R called the information region. The information before the k^{th} stage, $1 \le k \le d$, of processor $G(i,j)$, is shown in Fig. 3.3. The information region is bounded by cells (a,b) and (c,d) of the rectangular partition, where the values of a,b,c,d are as shown in the figure. We now define a *CCOMP* set of an information region R.

Definition 3.1: A *CCOMP* set of an information region R is a maximal connected set confined to R that touches a border of R.

Example 3.1: In Fig. 3.3, connected sets 1, 2 are *CCOMP* sets as they touch a border of R.

3.3.1.1. Outline of Quad Algorithm

As described above, the algorithm consists of d stages. The algorithm maintains the following invariant information at a processor before every stage.



NOTE: Here $a = \lfloor i/2^l \rfloor * 2^l$, $c = a + 2^l - 1$, $b = \lfloor i/2^m \rfloor * 2^m$, and $d = b + 2^m - 1$, where l, m are the number of x and y communications done before the stage ($l + m = k - 1$).

Figure 3.3. Information region of $G(i, j)$ before k^{th} stage

(1) A list of all *CCOMP* sets for the information region R of the processor before the stage. The *CCOMP* sets are specified by labels. The label of a *CCOMP* set is the minimum of the labels of all the *LCS* sets (at different processors) it contains in R.

(2) List of border segments for the borders of the information region R. Each border segment is associated with a *CCOMP* set that it connects to.

(3) List of *LCS* sets at the processor that connect to at least a border of the information region R. Each of them is associated with a *CCOMP* set that contains it. These *LCS* sets are called active *LCS* sets.

The information region before the first stage is nothing but the processor's region assigned by the partitioning and region assignment discussed in the previous chapter. So before the first stage, the *CCOMP* sets are precisely the *LCS* sets that are input to the *GCCL* algorithm.

At each stage a processor $P = G(i,j)$ communicates (sends and receives a message) with another processor Q, whose current information region abuts the current information region of P. The communication is said to be along the x axis (respectively y axis) if the common border of the information regions is perpendicular to the x axis (respectively y axis). For an x axis communication, $Q = G(r,j)$, where $r = i \oplus 2^l$ and l is the number of x axis communications done before the stage. This is illustrated in Fig. 3.4. An identical situation is true for a y axis communication. At any stage of the algorithm, the communication will be along one particular axis at all the processors. We have to perform x axis communication in exactly kx stages and y axis communications in the remaining $d-kx$ stages. For area based partitioning, these can be done in an order. However, it is better if they alternate in the beginning until the communications remaining are all done along one axis. For point based partitioning, the x axis communications can be done only after all the y axis communications are completed.

The work done by P in the stage is described below.

1: A message is sent to Q containing all the border segments of the borders of current information region, with their associated *CCOMP* set labels. The message structure is as shown in Fig. 3.5. A similar message is received from Q.

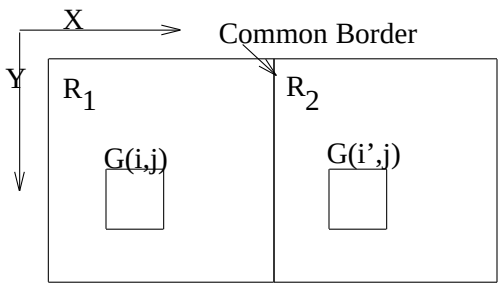
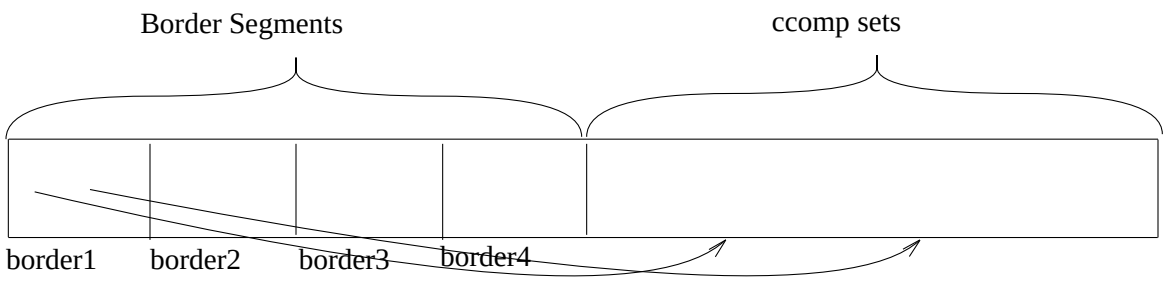


Figure 3.4. Proc $P = G(i,j)$ in an x axis communication with $Q = G(r,j)$



NOTE: The figure shows the border segments for the four borders pointing to their associated *CCOMP* sets in the message.

Figure 3.5. Message structure for the Quad algorithm

2: The *CCOMP* sets associated with border segments along the common information region border of P and Q are combined. 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 which is the minimum of the two.

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. The *CCOMP* sets for the new information region are taken to be those that are formed in Step 2 that touch a border of the new information region.

4: For each of the active *LCS* sets, the new set X it belongs to after the unions in Step 2 is determined. If X is a *CCOMP* set for the new region then the *LCS* set is retained in the active list. If not, the label of X 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 a stage of the algorithm. After d stages each of the *LCS* sets get a label denoting the *GCS* set it belongs.

3.3.1.2. Proof of Correctness of Quad Algorithm

To prove the correctness of the Quad algorithm, we first state a lemma. Consider any processor P. Let Q be the processor it communicates in the k^{th} stage. Also, let R_1, R_2 be the information regions of P and Q before the k^{th} stage. Let R be the union of the regions R_1 and R_2. The situation is as shown in Fig. 3.4.

Lemma 3.1: Let the invariant of the algorithm, as stated before, be true for the k^{th} stage at all the processors. After the unions in Step 2 at processor P, a *CCOMP* set S of region R_1 or R_2 is grouped under a set that represents the maximal connected set in R containing S. Each of the maximal connected sets gets the label that is the minimum of all its *LCS* sets.

Proof: Consider the *CCOMP* set S. It is available at P after Step 1 (either locally or sent by Q) because of the invariant before the k^{th} stage.

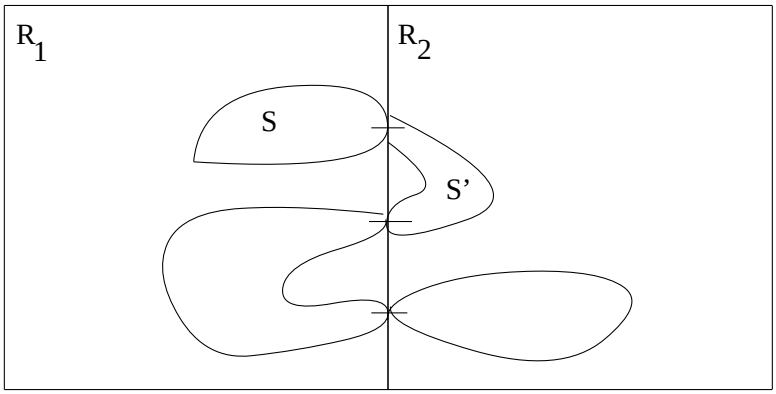
We first consider the case when S does not touch the common border of R_1 and R_2. In this case, it does not take part in the combinations in Step 2. Hence the statement of the lemma is true as S is a maximal set by itself.

Now, consider the case when S touches the common border. Assume S is a *CCOMP* set of R_1 without loss of generality. We can follow the connections of the set S in R. Initially we are in the region R_1. When we cross to R_2, we will be in a *CCOMP* set S' that is connected to S by a common border segment. Hence the two sets S and S' will be combined in Step 2. Thus the situation is as shown in Fig. 3.6. By continuing this argument, we see that S is grouped under a set that is the maximal connected set containing S in R. The label of the maximal set is the minimum of all the *CCOMP* sets in the set which, by the invariant before the k^{th} stage, is the minimum of all the constituent *LCS* sets.

□

Lemma 3.2: The invariant of the algorithm is true before every stage i.

Proof: The proof is done by induction on the stage i. It is trivially true for $i = 1$. Assuming it is true for $i = k$, we show that it is true for $i = k+1$. Let us consider any processor P at the k^{th} stage.



NOTE: The curved regions represent *CCOMP* sets connected by border segments.

Figure 3.6. Example illustration for Lemma 3.1

Let Q be the processor it communicates with in that stage. Also, let R_1, R_2 be the information regions of P, Q before the k^{th} stage. Then the information region R of P for the next stage is the union of the two regions R_1 and R_2. We can easily prove that the region R satisfies the invariant description shown in Fig. 3.3 for the $k+1^{th}$ stage.

Now consider any *CCOMP* set C of R. It is easy to see that there should at least be one *CCOMP* set S' of either R_1 or R_2 that is a part of C. Applying Lemma 3.1 for S', we find that after Step 2, S' is grouped under a maximal set C', which represents the *CCOMP* set C. The set will be identified to be a *CCOMP* set for the new region in Step 3. From Lemma 3.1, the label of C' is the minimum of all its constituent *LCS* sets. Thus all *CCOMP* sets of R are determined correctly after Step 3. Also, since all the sets after Step 2 are maximal connected sets the new *CCOMP* sets determined in Step 3 are *CCOMP* sets of R.

Consider any *LCS* set L that was active before the k^{th} stage. It can be verified that L connects to a border of R if and only if its maximal connected set S, after unions in Step 2, is a *CCOMP* set of R. Thus the active *LCS* sets for the new region R are correctly computed in Step 4. Also, the new border segments for the new region R can be easily verified to be computed correctly in Step 3.

Thus, the invariant is true at P before the $k+1^{th}$ stage. This proves the lemma.

□

Theorem 3.1: The Quad Algorithm solves the *GCCL* problem.

Proof: From Lemma 3.2, an *LCS* set L at a processor becomes inactive in Step 4 if its set S is not a new *CCOMP* set (i.e does not touch a border). Since S is a maximal set, and it does not touch a border, it is the *GCS* set containing L. From Lemma 3.1, the label of the set S is the minimum of the labels of its *LCS* sets. Hence, it is an unique label for the *GCS* set. The proof of the algorithm is complete once we note that each *LCS* set will become inactive at some stage as there can be no *CCOMP* sets after the final stage.

□

3.3.1.3. Analysis of Quad Algorithm

For purpose of analysis, we assume that the partitioning used is area based, and that x and y axis communications in the algorithm are done alternately. We also assume that the aspect ratio of the circuit is 1, and that $kx = d/2$. The analysis for other cases can be worked out in a similar fashion.

We assume that the border segments are distributed uniformly. Let B be the number of border segments along a cross section of a domain. The number of border segments in the initial information region of a processor is $\frac{4B}{\sqrt{P}}$, where $P = 2^d$ is the number of processors. It can be seen that the number of border segments in the current information region doubles every two stages. The number of border segments in the k^{th} stage is at most $b(k) = \frac{4B}{\sqrt{P}} 2^{\lfloor k/2 \rfloor}$.

The message size at the k^{th} stage is directly proportional to the number of border segments at the stage. A message travels to a processor that is either one or two hops away. Thus the communication time for the k^{th} stage is $O(t_\alpha + t_\beta b(k))$, where t_α and t_β are the startup and per byte time for communication. The number of union-find operations done at the stage is proportional to the number of border segments, i.e $b(k)$. If one uses the path compression algorithms (though this is not used in our implementation), these operations can be done in time $O(b(k)\alpha(b(k)))$, where α is the inverse of the Ackermann function, and is nearly a constant [32]. Thus the total time for the k^{th} stage is

$O(t_\alpha + t_\beta b(k) + b(k)\alpha(b(k)))$. Summing over the d stages, we can see that the total time taken by the algorithm is $O(dt_\alpha + t_\beta B + B\alpha(B))$.

The Quad algorithm is an efficient algorithm for the *GCCL* problem. The only disadvantage with the algorithm is that the message sizes and the time taken are proportional to B, the number of border segments along a cross section of the domain. Hence, in situations wherein there are large number of processors, and each processor is assigned only a small portion of the domain, this could be unacceptable because of both time and memory considerations. Except for this deficiency, the algorithm is efficient and could be used when there are small number of processors. In the following subsections, we describe algorithms that utilize the locality of connected sets to decrease the message sizes and the total time.

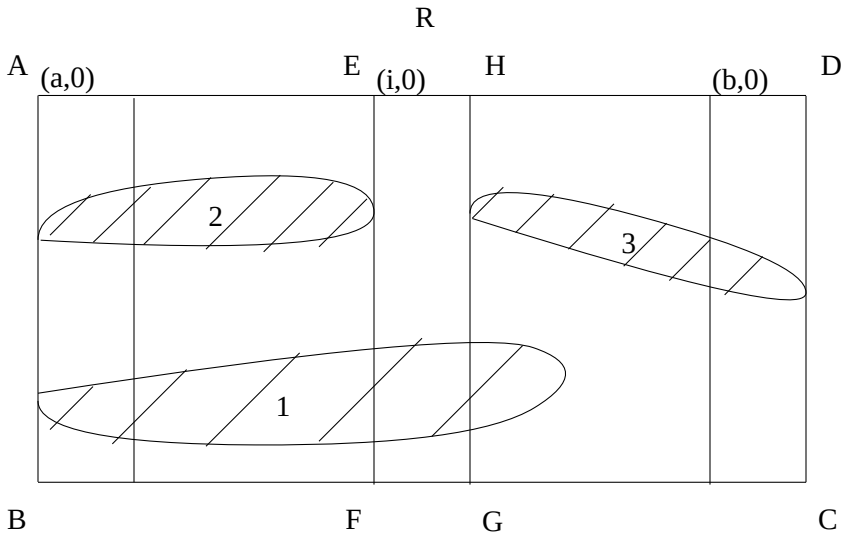
3.3.2. Overlap Slice Algorithm

The algorithm in this section solves the *GCCL* problem for a slice partition, which has $kx = d$. In the next section, this is generalized for the case of a general partition. The algorithm consists of d stages. In each stage a processor communicates (sends and receives a message) with at most two processors. The basic idea in this algorithm is that at each stage a processor increases its knowledge in the domain, similar to the Quad algorithm.

As in the Quad algorithm, for a stage a processor is associated with a rectangular region R called the information region. The information region before the k^{th} stage, $1 \le k \le d$, is shown in Fig. 3.7. The information region has effectively two borders as the other two coincide with a domain border. We now have the following definitions.

Definition 3.2: A *CCOMP* set of an information region R, as shown in Fig. 3.7, is a maximal connected set confined to R that touches either the borders AB and EF or the borders GH and CD or both.

Definition 3.3: A left(right) *ACTCOMP* set of an information region R, as shown in Fig. 3.7, is a



NOTE: Here $a = MAX(i - 2^{k-1} + 1, 0)$ and $b = MIN(i + 2^{k-1} - 1, 2^d - 1)$

Figure 3.7. Information region of $G(i,0)$ before the k^{th} stage

maximal connected set confined to ABFE(CDHG) that touches the borders AB and EF (GH and CD).

Example 3.2: In Fig. 3.7, the connected set 1 is a *CCOMP* set, and the connected sets 2, 3 are left and right *ACTCOMP* sets respectively.

3.3.2.1. Outline of Overlap Slice Algorithm

The algorithm maintains the following invariant information before every stage.

(1) A list of all *CCOMP* sets (specified by labels) for the information region R of the processor before the stage. The label of a *CCOMP* set is the minimum of the labels of all the *LCS* sets (at different processors) it contains in R.

(2) List of left and right *ACTCOMP* sets for the information region R. Each of them is associated with a *CCOMP* set that contains it. From the definitions, it can be verified that the maximal connected set in R containing an *ACTCOMP* set is a *CCOMP* set in R.

(3) List of *LCS* sets at the processor that connects to at least a border of the information region. Each of them is associated with a *CCOMP* set that contains it. These *LCS* sets are called active *LCS* sets. From the definitions, it can be verified that if an *LCS* set connects to a border of R it must be a part of a *CCOMP* set of R.

The information region before the first stage is the processor's region assigned by the partitioning and region assignment. So from the definitions, the *CCOMP* sets before the first stage are precisely the *LCS* sets that are sent to the *GCCL* algorithm. The border segments of the two borders of the processor's region are taken as the *ACTCOMP* sets.

At a stage k, a processor P communicates (sends and receives a message) with two processors $O = G(r,0)$ and $Q = G(s,0)$, where $r = i - 2^{k-1}$ and $s = i + 2^{k-1}$. The information regions of the two processors overlap the information region of P. It should be noted that one of the processors may not be defined, in which case P communicates with only one processor. This is illustrated in Fig. 3.8. The work done by P in the k^{th} stage is as follows.

1: A message is sent to O consisting of the left *ACTCOMP* sets of P with their associated *CCOMP* set labels. The message structure is shown in Fig. 3.9. Similarly, a message is sent to Q consisting of the right *ACTCOMP* sets of P with their associated *CCOMP* set labels. Similar

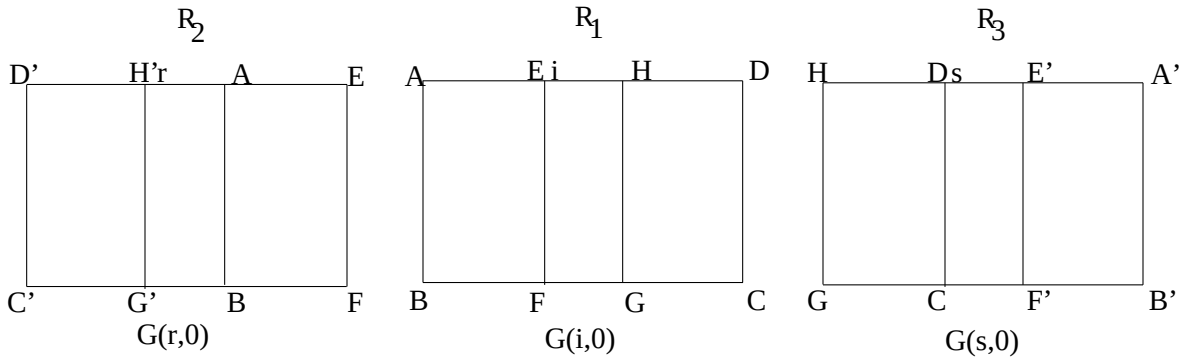
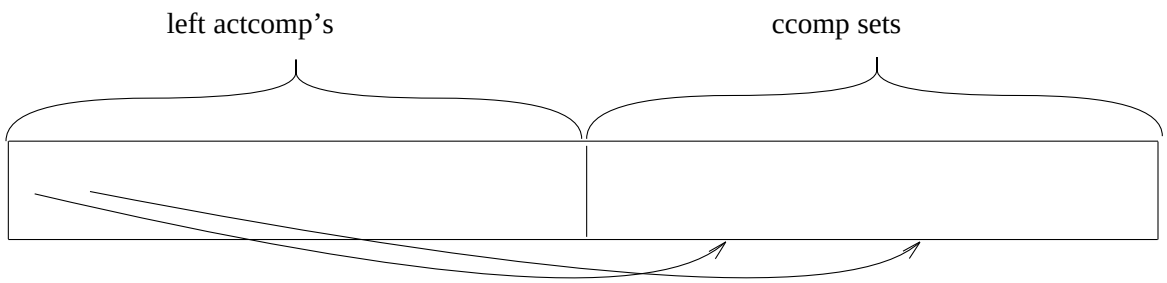


Figure 3.8. $P = G(i,0)$ communicating with $O = G(r,0)$, $Q = G(s,0)$ for the k^{th} stage



NOTE: The *ACTCOMP* sets point to their associated *CCOMP*s in the message.

Figure 3.9. Message structure of Overlap Slice algorithm

messages are received from O, Q.

2: The *CCOMP* set associated with a left *ACTCOMP* set of P is combined with the *CCOMP* set associated with the corresponding right *ACTCOMP* of O (received in Step 1). A similar combination is done between the *CCOMP* sets of P and Q.

3: The information region of P for the next stage is the combination of its current information region and those of O and Q. The *CCOMP* sets for the new information region are taken to be those that are formed in Step 2 that satisfy the definition of a *CCOMP* set (Definition 3.2). The left (right) *ACTCOMP* sets for the new information region are taken to be the *CCOMP* sets sent by $O(Q)$ that satisfy the definition of a *ACTCOMP* set (Definition 3.3). A new *ACTCOMP* set A derived above from a *CCOMP* set S is now associated with the new *CCOMP* set containing S after the unions in Step 2.

4: For each of the active *LCS* sets, the new set X it belongs to after the unions in Step 2 is determined. If X is a *CCOMP* set for the new region, then the *LCS* set is retained in the active list. If not, the label of X 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 a stage of the algorithm. After d stages each of the *LCS* sets gets a label denoting the *GCS* set it belongs.

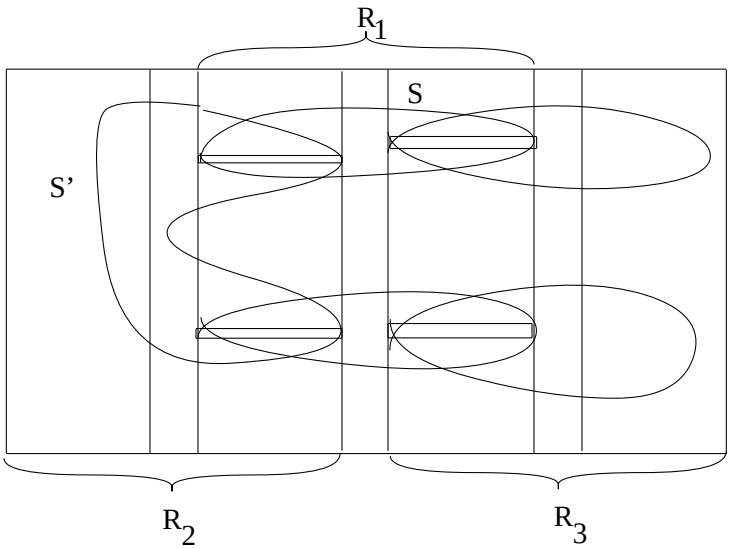
3.3.2.2. Proof of Correctness of Overlap Slice Algorithm

To prove the correctness of the algorithm, we first state a lemma. Consider any processor P. Let O, Q be the processors it communicates in the k^{th} stage. We denote the information regions of O, P, Q as R_2,R_1,R_3 respectively. Let R be the union of the regions R_1, R_2, R_3. The situation is shown in Fig. 3.8.

Lemma 3.3: Let the invariant of the algorithm, as stated before, be true before the k^{th} stage at all the processors. After the unions in Step 2 at processor P, a *CCOMP* set S available in P after Step 1 is grouped under a set that represents the maximal connected set in R containing S. The label of the maximal set is the minimum of all its *LCS* sets.

Proof: Consider the *CCOMP* set S. Either it was available locally at P before Step 1, or it was sent by O or Q in Step 1. We only prove the lemma for the case when S already was available in P as the other cases are similar.

We can follow the connections of the set S in R. Initially we are in the region R_1. When we cross to another region, say R_2, we are in a *CCOMP* set S' in R_2 that has a right *ACTCOMP* set whose corresponding left *ACTCOMP* set is associated with S. This follows from the definition of *CCOMP* and *ACTCOMP* sets. Hence, the two sets S and S' will be combined in Step 2. The above argument implies that each time we shift from one region to other we go through an *ACTCOMP* set. Also, we are always in a *CCOMP* set of the current region. Hence the situation will be as shown in Fig. 3.10. Thus we see that, after the unions in Step 2, S is grouped under a set that represents the maximal set in R containing S. Moreover, all the *CCOMP* sets connecting to S in R will also be grouped under the same set. The label of the maximal set is the minimum of all the *CCOMP* sets which, by the invariant before the k^{th} stage, is the minimum of all the constituent *LCS* sets. This completes the proof of the lemma.



NOTE: The curved regions represent the *CCOMP* sets connected by common *ACTCOMPs* represented as rectangles.

Figure 3.10. Example illustration for Lemma 3.3

□

Lemma 3.4: The invariant of the algorithm is true before every stage i.

Proof: The proof is done by induction on the stage i. Assuming the statement is true for $i=k$, we show that it is true for $i=k+1$. Let us consider any processor P at the k^{th} stage. Let O, Q be the processors it communicates with in that stage. We denote the information regions of O, P, Q as R_2, R_1, R_3 respectively. The information region R of P for the next stage is the union of R_1, R_2, R_3. The region R can be easily seen to satisfy the invariant description given in Fig. 3.7 for the $k+1^{th}$ stage.

Now consider any *CCOMP* set C of R. From the definitions it can be verified that it should contain at least one *CCOMP* set S of either R_1, or of R_2 that is sent by O, or of R_3 that is sent by Q. Applying Lemma 3.3 for S, we find that S is grouped under a maximal set C', that represents C. This set will be identified to be a *CCOMP* set in Step 3, with the label satisfying the conditions of Lemma 3.4. Thus all *CCOMP* sets of R are determined correctly after Step 3. Also, since the sets

after Step 2 are all maximal connected sets, the new *CCOMP* sets determined in Step 3 are *CCOMP* sets of R.

From the definitions, a left (right) *ACTCOMP* set of R corresponds to a *CCOMP* set in R_2 (R_3) that connects to both the borders. Such a set will be sent to P in Step 1 and hence the *ACTCOMP* set is correctly identified in Step 3, with the correct associations to its new *CCOMP* set.

It can be easily verified that any *LCS* set L connects to a border of R if and only if its maximal connected set S, after unions in Step 2, is a *CCOMP* set of R. Thus the active *LCS* sets for the new region R are correctly computed in Step 4. Thus, the invariant is true at P before the $k+1^{th}$ stage. This proves the lemma.

□

Theorem 3.2 : The Overlap Slice algorithm solves the *GCCL* problem.

Proof: The proof follows from Lemmas 3.3, 3.4, in a manner similar to the proof of Theorem 3.1.

□

The overlap slice algorithm could be modified so that a processor stops communication to the left (right) once the number of left (right) *ACTCOMP*s becomes 0. The correctness of such an algorithm can be easily verified.

3.3.2.3. Analysis of Overlap Slice Algorithm

The communication time and computation time of a stage depends on the number of *ACTCOMP* sets. In the worst case, this could be $2B$, where B is the number of border segments along a cross section. Thus the total time taken by d stages is $O(dt_\alpha + dBt_\beta + dB\alpha(B))$, which is the same as that of the Quad algorithm, applied to a slice partition.

The worst case analysis given above is pessimistic, as it ignores the fact that *ACTCOMP* sets span nearly 1/2 the current information region. Hence, their number is expected to decrease with the increase in stage number. If there are not many connected sets in the domain that span more than a processor's region, the number of *ACTCOMP* sets will be $2B$ only in the first stage, and close to 0 in the succeeding stages. Thus the total time taken would be $O(dt_\alpha + Bt_\beta + B\alpha(B))$.

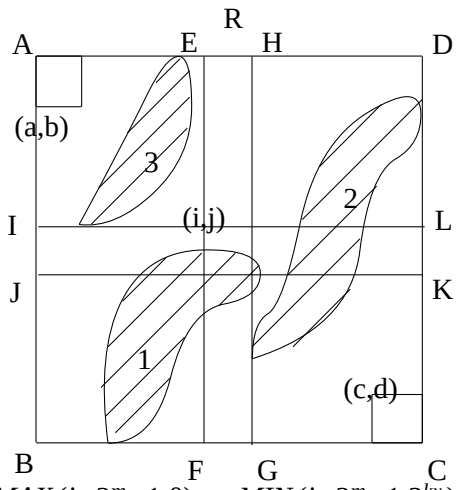
3.3.3. Overlap Quad Algorithm

In this section, we generalize the algorithm given in the previous section for the case of general rectangular partitioning. This algorithm, however, applies only for area based partitioning. The algorithm has d stages, and the information region for the k^{th} stage, $1 \le k \le d$ is shown in Fig. 3.11. We now have the following definitions. They refer to the information region R shown in Fig. 3.11.

Definition 3.4: A *CCOMP* set of R, is a maximal connected set confined to R that touches either the borders AB and EF or CD and HG or AD and IL or JK and BC.

Definition 3.5: In the case AB (CD) does not coincide with a domain border, a left (right) *XACTCOMP* set of R is a maximal connected set confined to ABFE (CDHG) that connects to borders AB and FE or AD and IL or JK and BC (HG and CD or AD and IL or JK and BC). In the other case, R does not have any left (right) *XACTCOMP* sets. An identical definition can be given for top and bottom *YACTCOMP* sets.

Definition 3.6: A collection of connected sets, all of which are not necessarily maximal, in the region ABFE(CDHG) is called a left(right) *XACT* collection of R, if for each left(right) *XACTCOMP*



NOTE: $a = MAX(i - 2^l + 1, 0), b = MAX(i - 2^m + 1, 0), c = MIN(i + 2^m - 1, 2^{kx}), d = MIN(j + 2^m - 1, 2^{d-kx})$, where l, m are the number of x and y axis communications done before the stage and $l + m = k - 1$.

Figure 3.11. The information region of $G(i, j)$ before the k^{th} stage

set X there exists at least one set in the collection which is a subset of X, provided that the border AB(CD) does not coincide with a domain border. In other words a *XACT* collection covers the collection of *XACTCOMP* sets. A similar definition can be given for top and bottom *YACT* collections.

From the definition, the list of *XACTCOMP* sets of the information region form a *XACT* collection. However, many other collections satisfy the definition of a *XACT* collection. The purpose of defining these sets, which may look mysterious now, will be clear as we describe the algorithm.

Example 3.3: In Fig. 3.11, the connected set 1 is a *CCOMP* set, the connected set 2 is a right *XACTCOMP* set, and the connected set 3 is both a left *XACTCOMP* and a top *YACTCOMP* set.

3.3.3.1. Outline of Overlap Quad Algorithm

The invariant information maintained at a processor $P = G(i,j)$ for the k^{th} stage depends on the type of communication done in the stage. We describe it for the case when the communication is an x axis communication. An analogous invariant is true for the case when the communication is a y axis communication. The processor P communicates with two processors $Q = G(r,j)$ and $O = G(s,j)$ (one of them may not be defined), where $r = i - 2^l$, $s = i + 2^l$, and l is the number of x axis communications done before the stage. The information regions of the two processors overlap the information region of P as shown in Fig. 3.12. Let R be the information region of P before the stage. The invariant information at P consists of

(1) A list of *CCOMP* sets of R. The *CCOMP* sets are specified by their labels. The label of a *CCOMP* set is the minimum of the labels of all the *LCS* sets (at different processors) it contains in R.

(2) A list of top and bottom *YACTCOMP* sets of R. Each of them is associated with a *CCOMP* set it is part of. From the definitions, it can be verified that the maximal connected set in R containing a *YACTCOMP* set is a *CCOMP* set of R.

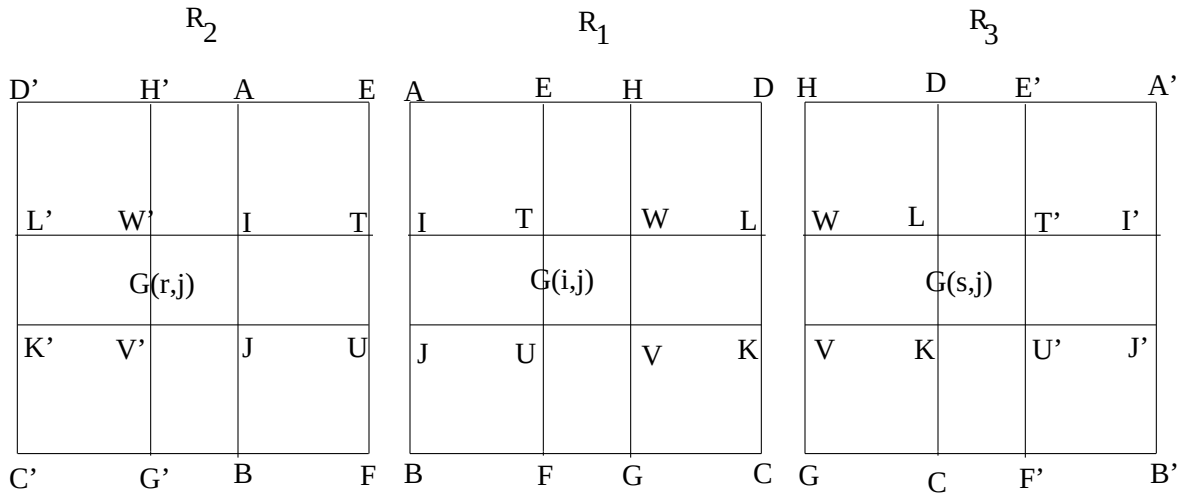


Figure 3.12. $P=G(i,j)$ in a x axis communication with $O=G(r,j)$ and $Q=G(s,j)$

(3) Left and right *XACT* collections of R. Each of the sets in the collections is associated with a *CCOMP* set of R that contains it, if such a set exists. The existence is not necessary because a set in a *XACT* collection need not be be a part of any *CCOMP* set in R. The left (right) *XACT* collection of P corresponds exactly to the right (left) *XACT* collection at processors O (Q).

(4) List of *LCS* sets at the processor that connects to at least a border of the information region. Each of them is associated with a *CCOMP* set that contains it. These *LCS* sets are called active *LCS* sets. From the definitions, it can be verified that if an *LCS* set connects to a border of R it must be a part of a *CCOMP* set of R.

Before the first stage, the *CCOMP* sets are given precisely by the *LCS* sets at the processor. The border segments along the four borders give rise to the *XACTCOMP* (left and right border) and *YACTCOMP* (top and bottom) sets for the region. Each of these sets is associated with the *CCOMP* set corresponding to their border segment.

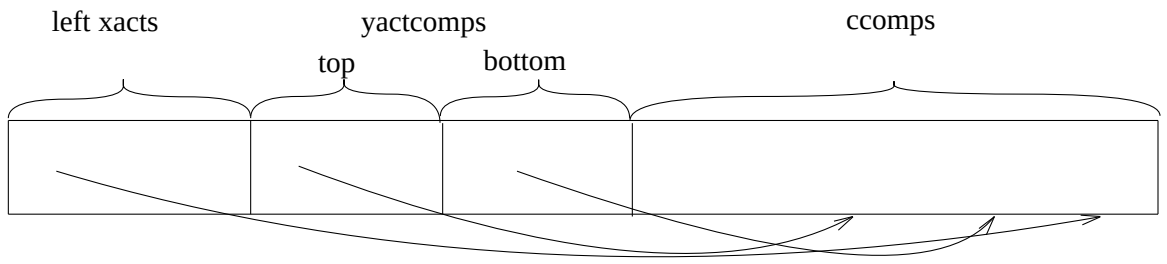
At any stage of the algorithm, the communication will be along one particular axis at all the processors. As in the Quad algorithm, there are kx stages with x axis communication, and $d-kx$ stages with y axis communication. These are done alternately, as in the Quad algorithm.

Now we describe the work by processor P in the k^{th} stage. Let O, Q be the processors it communicates with in the stage. We assume the communication is an x axis communication. The description for the work in a stage with y axis communication is analogous.

1: A message is sent to O consisting of the left *XACT* collection of P, with the *CCOMP* set (if any) associated with each set in the collection. In the same message, the top and bottom *YACTCOMP* sets of P, with their associated *CCOMP* sets, are sent. The *CCOMP* sets sent in the message are not duplicated. The message structure is shown in Fig. 3.13. A similar message is sent to processor Q. Similar messages are received from O and Q.

2: The *CCOMP* sets associated with a left *XACT* set of P and its corresponding right *XACT* set of O (sent in Step 1) are combined using a union operation. A similar combination is done between the *CCOMP* sets of P and Q.

3: The information region for the next stage is the combination of its current information region and those of O and Q. The new *CCOMP* sets are taken to be those that are formed in Step 2 that satisfy the definition of a *CCOMP* set (Definition 3.4). The new left(right) *XACTCOMP* sets are chosen from the *CCOMP* sets sent by O (Q) that satisfy the definition of a *XACTCOMP* set (Definition 3.5). A new *XACTCOMP* set X, derived from a *CCOMP* set S, is associated with



NOTE: The *XACT*s and the *YACTCOMP*s point to their *CCOMP* sets in the message.

Figure 3.13. The message structure of the Overlap slice algorithm

the new *CCOMP* set containing S after the unions in Step 2.

4: A top (bottom) collection of *YACT* sets for the new information region is obtained by simply collecting the top (bottom) *YACTCOMP* sets sent by O, Q, to those that were in P before the stage. For a new *YACT* set Y, the *CCOMP* set C associated with the *YACTCOMP* set Y is first identified. The maximal set S, containing C after the unions in Step 2, is then obtained. The new *YACT* set Y is associated with C, if C is a new *CCOMP* set.

5: For each of the *LCS* sets, the new set it belongs 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.

3.3.3.2. Proof of Correctness of Overlap Quad Algorithm

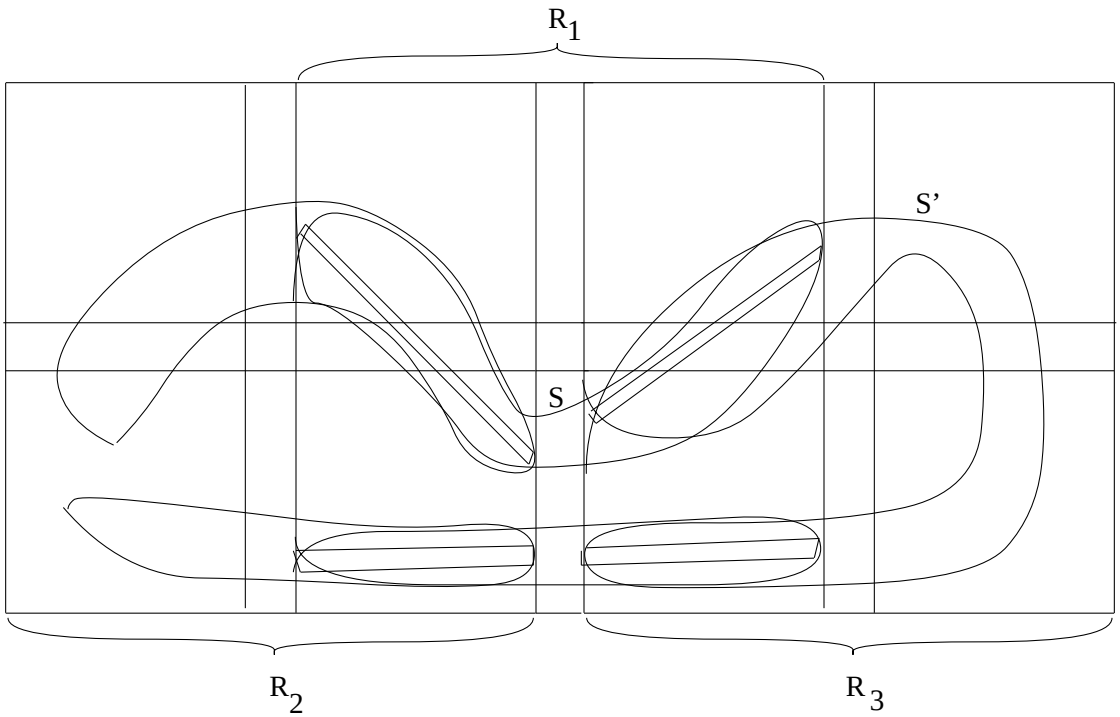
To prove the correctness of the algorithm, we first state a lemma. Consider any processor P. Let O, Q be the processors it communicates in the k^{th} stage. Let the information regions of O, P, Q be R_2, R_1, R_3 respectively. Also, let R be the union of the regions R_1, R_2, R_3. The configuration is as shown in Fig. 3.12.

Lemma 3.5: Let the invariant of the algorithm, as stated before, be true for the k^{th} stage at all the processors. After the unions in Step 2 at processor P, a *CCOMP* set S, available in P after Step 1, is grouped under a set that represents the maximal connected set in R containing S. The label of the maximal set is the minimum of all its *LCS* sets.

Proof: We prove the lemma for the case when the communication in the k^{th} stage is an x axis communication. The proof for the other case is similar.

Consider the *CCOMP* set S. Either it was locally available at P before Step 1 or was sent by O or Q in Step 1. We only prove the lemma for the case when S was already available in P as the other cases are similar.

We can follow the connections of the set S. Initially we are in the region R_1. When we cross to another region, say R_2, we will be in a *CCOMP* set in R_2, S', that has a right *XACTCOMP* set whose corresponding left *XACTCOMP* set is associated with S. This can be easily verified by considering the various combinations. However, from the invariant, the information about *XACTCOMP* sets is not maintained in the stage. The *XACT* collection that is maintained is very useful since for each left *XACTCOMP* set of R_1 there exists at least one set in the left *XACT* collection that is associated with the same *CCOMP* set. That set will have a corresponding set in the right *XACT* collection of R_2. Hence the the sets S, S' will be combined in Step 2. The above argument argument implies that each time we shift from one region to another we go through a *XACT* set. Also, we are always in a *CCOMP* set of the current region. Hence the situation is as shown in Fig. 3.14. Thus after the unions



NOTE: The curved shaped regions represent the *CCOMP* connect together by *XACTCOMP*s shown as quadrilaterals.

Figure 3.14. Example illustration for Lemma 3.5

in Step 2, S is grouped under a set that represents the maximal set in R containing S. Moreover, all the *CCOMP* sets connecting to S in R will be grouped under the same set. The label of the maximal set is the minimum of all the *CCOMP* sets, which by induction is the minimum of all the constituent *LCS* sets. This completes the proof of the lemma.

□

Lemma 3.6: The invariant of the algorithm is true before every stage i.

Proof: The proof is done by induction on the stage i. The statement is trivially true for $i = 1$. Now we prove that if it is true for $i = k$, it is true for $i = k+1$.

We assume that the communication of the k^{th} stage is a x axis communication, the other case being analogous. Let us consider a processor P at the k^{th} stage. Let O, Q be the processors it communicates in that stage. We denote the information regions of O, P, Q as R_2, R_1, R_3 respectively. The information region R of P for the next stage is the union of R_1, R_2, R_3. The region R can be easily verified to satisfy the invariant description given in Fig. 3.11.

Now consider any *CCOMP* set C of the combined region R. From the definitions, C should contain at least one *CCOMP* set S of either region R_1, or of R_2 that is sent by O, or of R_3 that is sent by Q as a subset. The proof of this statement consists of going through all the possible combinations. Applying Lemma 3.5 for S, we find that S is grouped under a maximal set C', that represents C. Thus all the *CCOMP* sets of R are determined correctly after Step 3. Also, since the sets after the unions in Step 2 are maximal sets, the new *CCOMP* sets determined in Step 3 are *CCOMP* sets of R.

From the definitions, the left (right) *XACTCOMP* sets of R correspond to *CCOMP* sets of R_2,R_3, that satisfy the definition of a *XACTCOMP* set. These sets are sent in Step 1 and hence the *XACTCOMP* sets are correctly identified in Step 3, with the correct associations to its new *CCOMP* set.

The collection obtained by combining the top (bottom) *YACTCOMP* sets of R_1, R_2 and R_3 can be shown to be a top (bottom) *YACT* collection for R. This consists of showing that for each

YACTCOMP set Y of R there exists at least a set of either region R_1,R_2,R_3 that is a subset of Y. Also, it can be seen that the *YACT* collections at P will correspond with the ones in processors that P communicates in the next stage, if the communication in the next stage is a y axis communication.

It can be easily verified that any *LCS* set L connects to a border of R if and only if its maximal connected set S, after unions in Step 2, is a *CCOMP* set of R. Thus the active *LCS* sets for the new region R are correctly computed in Step 5.

Thus the invariant at P will be true if the next stage involves y axis communication. However, if the next axis communication is an x axis communication it implies that there are no more y axis communications. In this case, the collection of *YACT* sets computed above will be empty. The only information available are the *CCOMP* sets and the *XACTCOMP* sets. This corresponds to the invariant for a stage with x axis communication, as the information region in this case does not have any *YACTCOMP* sets. Thus the invariant will be true at P even in this case for the next stage. This completes the proof of the lemma.

□

Theorem 3.3: The Overlap Quad algorithm solves the *GCCL* problem.

Proof: The proof follows from Lemmas 3.5, 3.6, in a manner similar to Theorem 3.1.

□

3.3.3.3. Analysis of Overlap Quad Algorithm

For sake of analysis, we make the same assumptions as for the case of the Quad algorithm. The computation and communication time of a stage depends on the number of *X(Y)ACTCOMP* and *Y(X)ACT* sets. In the worst case, these are proportional to the number of border segments of the current information region. Thus applying an analysis, similar to the Quad algorithm, we can derive the same complexity, which is $O(dt_\alpha + Bt_\beta + B\alpha(B))$. However, the constants in this case are larger.

It can be observed that the elements in a *XACT* or *YACT* collection are respectively *XACTCOMP* and *YACTCOMP* sets of the previous stage. The *XACTCOMP*, *YACTCOMP*, *CCOMP* sets are sets that span a distance which is nearly half the minimum of the x and y spans of the infor-

mation region. Thus the number of $X(Y)ACTCOMP$ and $Y(X)ACT$ sets can be expected to small. If there are not many connected sets that span more than a processor's region, the total complexity of the algorithm would be $O(dt_\alpha + \frac{B}{\sqrt{P}} t_\beta + \frac{B}{\sqrt{P}} \alpha(\frac{B}{\sqrt{P}}))$, accounting only for the computation and communication required in the initial stages.

3.3.4. Implementation

The *GCCL* algorithms have been implemented in the C programming language, using 1400 lines. The algorithms run on an Intel iPSC2/D4/MX hypercube multiprocessor. In Tables 3.1, 3.2, 3.3 we present some results for the above three algorithms, and a fourth algorithm called the Slice algorithm. The Slice algorithm is same as the Quad algorithm except that it is restricted to slice partitioning. The test cases are practical VLSI circuits involving adders, PLA's and ALU's. They consist of 55000, 140000, and 220000 rectangles respectively.

In Table 3.1, we give performance of the algorithms on a 4-dimensional hypercube. We present both the run time and the sum of lengths of messages sent by a processor, averaged over all the processors. Since the comparison of the algorithms is interesting only at higher dimensions, we simulated hypercubes of dimension 7 and 8 by running multiple processes in each of the nodes. In Table 3.2 and 3.3 we present the results comparing the four algorithms on 7 and 8 dimension hypercubes respectively.

Data	Boxes	Quad		Ovlp Quad		Slice		Ovlp Slice	
		msg (Kb)	time (sec)	msg (Kb)	time (sec)	msg (Kb)	time (sec)	msg (Kb)	time (sec)
test1	55000	4.8	0.7	5.3	0.7	5.2	0.8	4.8	0.7
test2	140000	7.4	1.1	5.9	0.9	8.3	1.2	5.0	0.8
test3	228000	13.4	1.8	9.3	1.8	16.0	2.8	9.0	1.5

Table 3.1. Performance on a 4 dimensional hypercube

Data	Boxes	Quad msg (Kb)	Ovlp Quad msg (Kb)	Slice msg (Kb)	Ovlp Slice msg (Kb)
test1	55000	7.7	6.8	12.2	8.2
test2	140000	12.0	7.3	15.9	8.3
test3	228000	18.0	9.6	30.7	14.9

Table 3.2. Performance on a 7-dimensional hypercube

Data	Boxes	Quad msg (Kb)	Ovlp Quad msg (Kb)	Slice msg (Kb)	Ovlp Slice msg (Kb)
test1	55000	8.4	6.9	14.7	9.7
test2	140000	12.3	6.9	18.4	10.0
test3	228000	19.2	8.5	35.4	18.0

Table 3.3. Performance on a 8-dimensional hypercube

The results show that for VLSI circuits the Overlap Quad is more efficient then the Quad algorithm. Also, the Overlap Slice algorithm is more efficient then the Slice algorithm. Thus the locality available in the VLSI circuits is successfully utilized by the Overlap algorithms.

3.4. Algorithm for a Shared Memory Multiprocessor

3.4.1. Parunion Algorithm

The Parunion algorithm solves the *GCCL* problem in an easy and interesting way using parallel union find and locking. Each processor, after obtaining the border segments for the four borders, with their associated *LCS* set labels, writes the information onto the shared memory. It, then, 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. 3.15. 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, y are still 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), depending on whether the tree rooted at x has lesser number of elements than the tree rooted at y (or otherwise). This is done for the sake of efficiency, as this results in trees of lesser height. The complete algorithm is shown in Fig. 3.16.

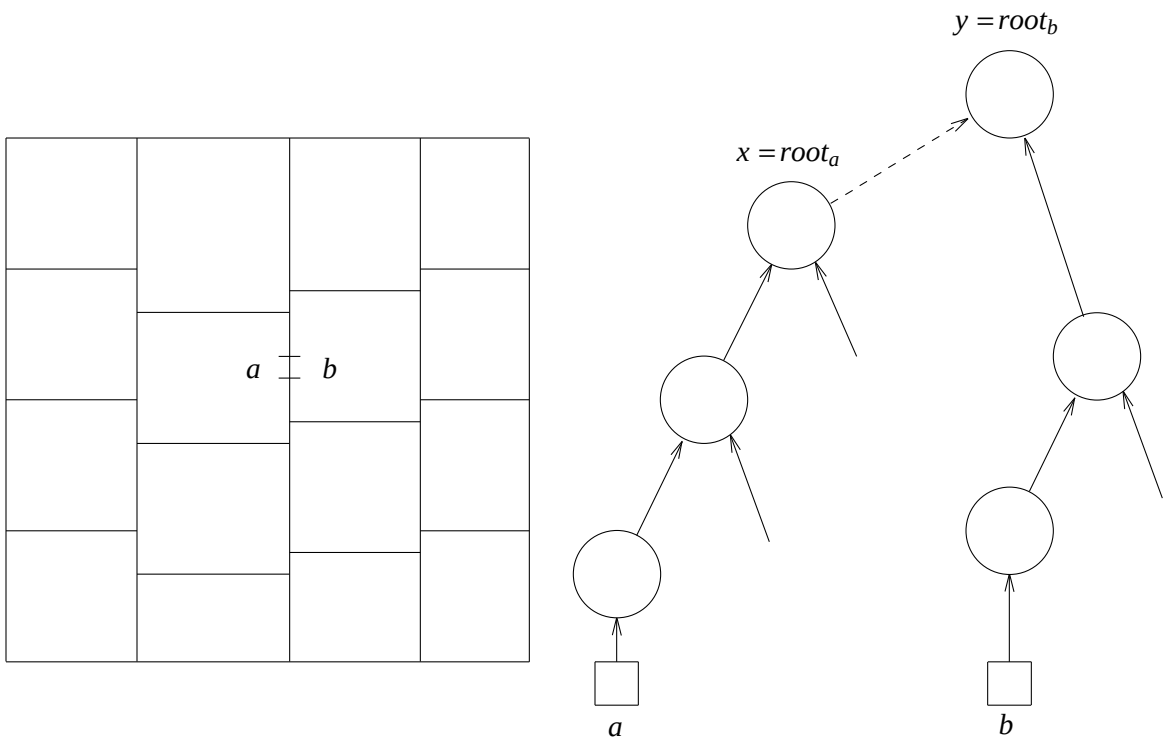


Figure 3.15. Example partitioning (b) Illustration for Parunion algorithm

```
union(a, b)
    x ← LCS(a), y ← LCS(b), fin ← 1
    while ( fin ≠ 1 ) do
        x ← findroot(x), y ← findroot(y)
        if ( x = y ) then return;
        lock(x), lock(y)  /* In increasing order of labels  */
        if ( x and y are still roots ) {
            if ( x.count > y.count ) /* Count gives no of nodes; and is initially 1*/
              interchange(x, y);
            x.father ← y, fin ← 1
            y.count = y.count + x.count /* Add Count fields */
          }
        unlock(x), unlock(y)
    end while
end union
```

Figure 3.16. The union algorithm

Once all the *union* operations are performed, for each *LCS* set L the label of the root of the tree it belongs to is determined. This is taken as the label of the *GCS* set corresponding to L. The correctness of the algorithm can be easily proved.

3.4.2. Analysis of Parunion Algorithm

We now analyze the algorithm, making the same assumptions as for the Quad algorithm. The time taken by the algorithm depends on memory contention. If we assume that the parallel unions performed at different processors do not interfere in a large way, the time taken by the algorithm depends on the number of union operations performed at each processor. This is same as the number of border segments at the processor, which is $O(\frac{B}{\sqrt{P}})$. Thus the time taken can be estimated as $O(\frac{B}{\sqrt{P}}u)$, where u is the time for an union operation. Since small trees are hooked onto large trees by the union algorithm, the height of a tree is at most $O(\log n)$, where n is the number of nodes in the

tree. Thus $u = O(\log(BP))$, as the maximum number of nodes in a tree is $B\sqrt{P}$, the total number of border segments at all the processors.

3.5. Summary

In this chapter, we have described practical algorithms for the geometric connected component labeling problem. For a hypercube, we have described two different algorithms for the problem and compared their performance. These algorithms have different stages of message passing. At each stage, a processor increases its information region in the circuit. The more sophisticated algorithm uses overlap of information regions to reduce the message and computational requirements. For a shared memory multiprocessor, we have presented a simple but efficient algorithm for the problem. This algorithm uses a parallel union-find algorithm.

CHAPTER 4.

RECURSIVELY PARTITIONED DISTRIBUTED DATA STRUCTURES

In Chapter 2, we showed that point based partitioning leads to a good balance of load among the processors. In implementing the partitioning strategy on a distributed memory multiprocessor, we naturally came across the problem of maintaining a partitioned distributed data structure of points. The distributed data structure had to be maintained under the operation of addition of sets of points. In this chapter, we look at the problem of maintaining a general recursively partitioned data structure of points on a multiprocessor, under the operations of addition and deletion of sets of points. This problem also arises in a class of applications involving particle evolution, such as vortex methods for fluid dynamics problems, n-body problems and multiple target tracking [33].

A recursively partitioned (*RP*) data structure of points is specified in terms of levels. At each level, the region is split into a specified number of parts based on a specified coordinate. The objective is to have an approximately equal distribution of points in the final regions. In this chapter, we give algorithms for maintaining a *RP* data structure on a multiprocessor, under the operations of addition and deletion of points.

In Section 4.1, we define a *RP* data structure, and formulate the problem of maintaining a *RP* data structure on a multiprocessor. Section 4.2 presents a solution to this problem for the special case of a slice partitioned data structure. In Section 4.3, we generalize this solution for the case of a general *RP* data structure. In Section 4.4, we analyze the solutions for a hypercube. In Section 4.5, we present some performance results. In Section 4.6, we discuss modifications to the solutions for better average case performance.

4.1. Recursively Partitioned Data Structure

An *RP* data structure involves partitioning a set of points S in a k dimensional Euclidean space E^k. An *RP* data structure is specified hierarchically in terms of levels. Let h be the number of levels. At each level l, $0 \le l \le h-1$, each region at the level is split into a specified number n_l sub regions at level $l+1$, based on a specified coordinate c_l for the level. It is assumed here, that each of the k coordinates provides a linear ordering of the points in S. In other words, no two points in S have the same value for any coordinate. An *RP* data structure is similar to the Quad tree data structure used in Computational geometry [34].

An objective of an *RP* data structure is that in each of the final regions (i.e those at level h), there are either $\lfloor N/P \rfloor$ or $\lceil N/P \rceil$ points, where $N = |S|$ and $P = \prod_{l=0}^{h-1} n_l$ (i.e the number of final regions). This is achieved as follows. Let D be a region at level l, $0 \le l < h$. We need to determine the partitioning lines to derive the sub regions D_i, $0 \le i < n_l$, ordered along the c_l coordinate. Let n be the number of points in D and let $f(x,y,z)$ denote $x\lfloor y/z \rfloor + \min(x, y \bmod z)$. Consider the linear order defined by the c_l coordinate. The region D_i is assigned all those points in D that occur in position j in the linear order, where $f(i,n,n_l) < j \le f(i+1,n,n_l)$. It can be easily proved that this data structure realizes the objective of an *RP* data structure stated earlier. The following lemma states some properties of f that will be used later. The proof of this lemma is trivial and hence omitted.

Lemma 4.1.. The following statements are true for $0 \le x \le z-1$.

(1) $f(x+1,y,z) \ge f(x,y,z) + \lfloor y/z \rfloor$.

(2) $f(x,y,z) \le f(x,y+\alpha,z) \le f(x,y,z) + \alpha$, for $\alpha \ge 0$.

(3) $\lfloor \alpha/z \rfloor \le f(x+1,y+\alpha,z) - f(x,y+\alpha,z)$
$- f(x+1,y,z) + f(x,y,z) \le \lceil \alpha/z \rceil$, for $\alpha \ge 0$.

The slice partitioned data structure in Fig. 2.2(a) is a *RP* data structure with $h = 1$, $n_0 = 4$, $c_0 = 1$, i.e the x coordinate. The rectangle partitioned data structure in Fig. 2.2(b) is a *RP* data structure with

$h = 2$, $n_0 = 4, n_1 = 4$, $c_0 = 1$, and $c_1 = 2$. Another *RP* data structure is shown in Fig. 4.1, which has $h = 3$, $n_0 = 2$, $n_1 = 4$, $n_2 = 2$, $c_0 = 1$, $c_1 = 2$, and $c_2 = 1$. This *RP* data structure has 1 region at level 0, 2 at level 1, 4 at level 2, 8 at level 3 and 16 at level 4.

The problem considered in this chapter consists of maintaining a distributed *RP* data structure on a multiprocessor, consisting of P processors. Each of the P final regions is associated with a corresponding processor. The processor associated with a final region D should contain the points that belong to D. It should be noted that the boundaries of regions at all levels change, as points are added and deleted from the distributed data structure.

4.2. Algorithms for Slice Partitioned Data Structure

In a slice partitioned data structure, the domain region D is divided into P sub regions D_i, $0 \leq i < P$, based on a linear order. We assume that the linear order for partitioning is given by the first coordinate, without loss of generality.

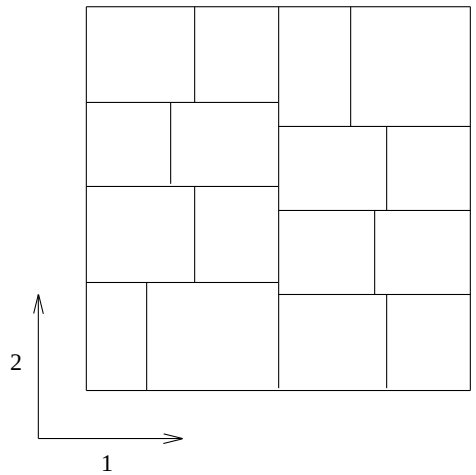


Figure 4.1. A recursively partitioned data structure

We are now ready to describe the algorithm for updating the distributed slice partitioned data structure, under the operations of addition and deletion of points. The points assigned to a processor are stored in a data structure that allows fast insert and delete operations and easy access to the points in both ascending and descending order of the first coordinate. One such data structure is a balanced binary search tree based on the first coordinate. Insertion and deletion can be done in $O(\log n)$ time, where n is the number of points in the tree. Also, the points can be accessed in both descending and ascending order of the first coordinate by just traversing the tree.

We define p_{act} to be N if $N < P$, and equal to $P-1$ if $N \geq P$. For each region D_i, we maintain two numbers lb_i, and rb_i at processor i. The value rb_i, for $0 \leq i < p_{act}$, is defined as the first coordinate of the last point belonging to region D_i. It is defined to be $+\infty$, if $i \geq p_{act}$. The value lb_i, for $i \geq 1$, is defined to be equal to rb_{i-1}. It is defined to be $-\infty$, if $i = 0$. Thus each region D_i, $0 \leq i < p$, contains all those points from S whose first coordinate x satisfies the condition $lb_i < x \leq rb_i$.

We now present the algorithm for adding (deleting) a set W of points to (from) the distributed data structure. The set W is first broadcast to all the processors. Then the following steps are done at a processor i.

procedure *slice_add(delete)*

(1) The set W is broken into set W_1, containing points with first coordinate less than or equal to lb_i, set W_3, containing points with first coordinate greater than rb_i, and set W_2 containing rest of the points. The points in W_2 are then added to (deleted from) the data structure in the processor. Let $N' = N + |W|$ $(N - |W|)$, be the number of points in D after the add (delete) operation.

(2) Let $left_i$ be the current number of points in the regions to the left of D_i. This can be obtained by a parallel prefix operation or directly as $f(i,N,p) + |W_1|$ $(f(i,N,p) - |W_1|)$. If $left_i < f(i,N',p)$ then the processor has to send points to the processors on the left side. The points to be shifted are obtained by traversing the points in the data structure in increasing order of the first coordinate. As the points are traversed their position in the linear order can be

determined. Thus the processor, to which these points have to be shifted, can be identified. All the points that need to be shifted to a processor, are collected and sent in a single message. This process is continued until either there are no more points, or the next point considered is to remain in the processor.

(3) Let $right_i$ be the sum of the current number of points in the region of processor i and $left_i$ If $right_i > f(i+1, N', p)$ then the processor sends points to processors on the right side. This process is done exactly as in Step 2 above except that the points are traversed in decreasing order of the first coordinate.

(4) The processor receives messages from the left if $left_i > f(i, N', p)$. It receives all those messages till it has received all its points from the left. These points are then added to the data structure.

(5) The processor receives messages from the right if $right_i < f(i+1, N', p)$. This is done in a manner similar to Step 4.

(6) Set N to N'. The value of p_{act} is recomputed. The processor sends its last assigned point to processor $i+1$, if $i \neq p-1$, and waits for a similar message from processor $i-1$, if $i \neq 0$. The values of lb_i and rb_i are then updated.

end *slice_add(delete)*

This completes our description of the algorithm. The correctness of the algorithm is clear from its description. We will not evaluate this algorithm separately, as we will be presenting the analysis for the general case in Section 4.4.

4.3. Algorithms for Recursively Partitioned Data Structures

In this section, we generalize the algorithm of the previous section for the case of a general *RP* data structure. We present two algorithms, the first being a simple generalization of the

slice_add(delete) algorithm, the other being a more efficient algorithm, specially for the case when $h \gg k$, i.e when the number of levels is larger than the number of coordinates. Such a situation arises, for example, in the parallel implementation of $K-d$ trees [34].

We make the following assumptions in both the algorithms. The data structure for storing the points in a region of level h, allocated to a processor, consists of set of k balanced binary search trees, one for each coordinate. For each region D at level l, $0 \leq l \leq h$, we associate a processor p_D. This could be done as follows. If $l = h$, p_D is the same as the processor assigned to region D. If $l \neq h$, p_D is recursively defined to be p_{D_0}, where D_0 is the first sub region of D at level $l+1$. We assume that for each region D, the *lb* and *rb* values, as defined in the previous section, are available at processor p_D. Also, let U denote the region at level 0.

4.3.1. General Algorithm 1

Let W be the set of points that are added or deleted. The algorithm for updating is described below.

procedure *gen1_add(delete)*

(1) The set W is sent to all processors using a tree. Using the *lb* and *rb* values, only the points that fall in a processor's region are directed to a processor. The local additions (deletions) are then performed.

(2) This is the rebalancing step. It proceeds in a top down fashion, i.e rebalancing regions of level l is followed by rebalancing regions at level $l+1$ and so on. Let D be a region at level l, and D_i, $0 \leq i < n_l$, be its sub regions at level $l+1$. The rebalancing of D is done as in the *slice_add(delete)* algorithm by the processors p_{D_i}, $0 \leq i < n_l$. The important differences are

   (a) The number N', that gives the number of points in D after the add (delete) operation, the numbers $left_i$ and $right_i$ are computed using a parallel prefix algorithm.

(b) The points to be shifted to the left or right from region D_i have to be first collected at p_{D_i}. If a points have to be shifted to the left or right, they are respectively the a minimum or maximum points in region D_i based on the c_l coordinate. These are obtained by first finding the a minimum or maximum points, based on the c_l coordinate, in all sub regions of D_i at level h. This information is then recursively combined to obtain information for regions at higher levels. The combining operation is a merge (as in sorting) operation. Let R be a sub region of D_i at level t. Referring to Fig. 4.2, processor p_R gets the a minimum or maximum points in R for the c_l coordinate, from the a minimum and maximum points for sub regions R_j, stored at processor p_{R_j}, $0 \leq j < n_t$. The combination operation is continued till level $l+1$, when the required information is obtained at p_{D_i}.

(c) The points sent from processor p_{D_i}, to the left and right, are directed to sub regions of D_i at level h, where they are deleted from the local data structure. Similarly, the points received by processor p_{D_i} are directed to sub regions of D_i at level h, where they are added to the local data structure.

end *gen1_add(delete)*

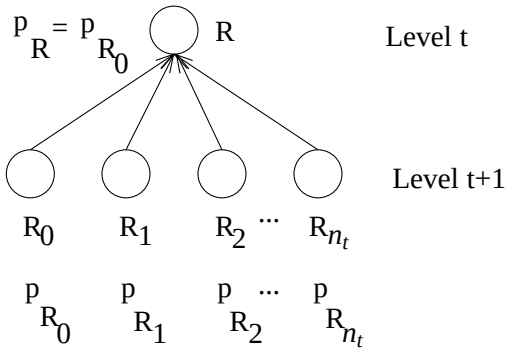


Figure 4.2. The bottom-up combining operation

This algorithm is an efficient algorithm, except for the case when $h \gg k$, i.e the number of levels in the data structure is larger than the number of coordinates. In the next subsection, we present an algorithm that is efficient for all the cases.

4.3.2. General Algorithm 2

For this algorithm, we introduce a new operation *add_delete*(A,B). This operation implies deleting the points in set B followed by adding the points in set A. The main idea of the algorithm is as follows. Referring to Fig. 4.3, suppose we have an *add_delete*(A,B) operation on a region D at level l. This operation will be converted to *add_delete*(A^i,B^i) operations on its sub regions D_i, $0 \le i < n_l$, so that the sub regions D_i are balanced.

The conversion is performed by the processors p_{D_i}, $0 \le i < n_l$. For this conversion, p_{D_i} needs to know only two sets *lset*(D_i) and *rset*(D_i) of points from D_i, for $0 \le i < n_l$. The set *lset*(D_i) consists of the first w points in D_i in the linear order based on coordinate c_l, where $w = \max(|A|,|B|)$. If there

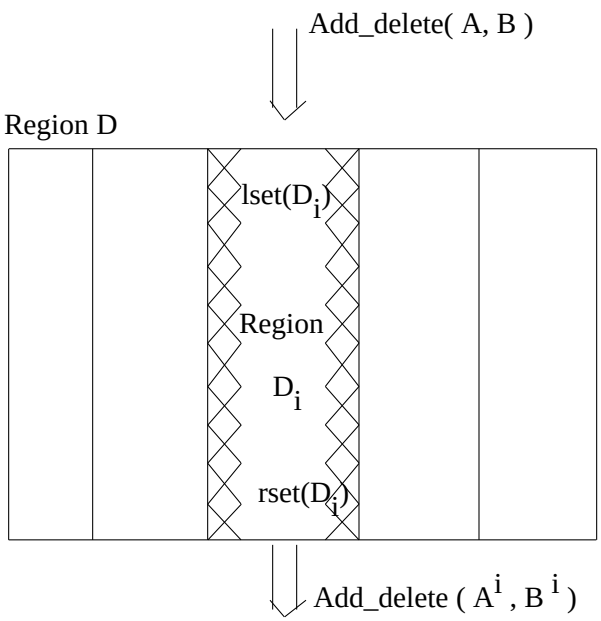


Figure 4.3. The main idea of general algorithm 2

are not so many points, we take *lset*(D_i) to be the same as the complete set of points in D_i. Similarly, *rset*(D_i) consists of the last w points in D_i in the linear order based on coordinate c_l, and the whole set of points in D_i in case there are not so many points.

Let n be the number of points in region D. The conversion of an *add_delete* (A,B) operation on region D to *add_delete* operations on the sub regions of D is done as follows. Let $\alpha = |A|$ and $\beta = |B|$. The work done by processor p_{D_i} is described below. This algorithm is called *decompose*.

procedure *decompose*

(1) The set A of points is first sorted into a list A based on coordinate c_l. The list A is then partitioned into lists A_1, A_2 and A_3 as in the algorithm for slice partitioned data structure. Similarly, the set B is partitioned into lists B_1, B_2 and B_3. The lists A^i, B^i are initialized to be null. The lists L_i and R_i are initialized to *lset*(D_i) and *rset*(D_i) respectively. Let $\alpha_j = |A_j|$ and $\beta_j = |B_j|$, for $j = 1,2,3$.

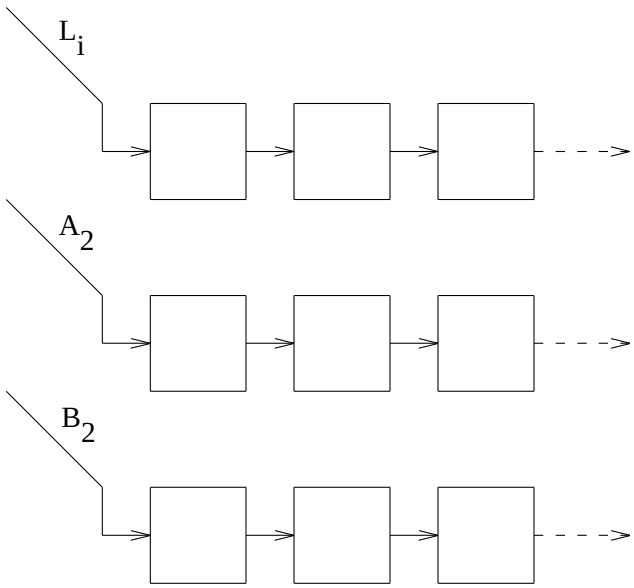
(2) If $f(i,n,n_l) + \alpha_1 - \beta_1 < f(i,n+\alpha-\beta,n_l)$ then the processor has to send some information to the left. This is done as in the algorithm for a slice partitioned data structure. The points to be shifted are obtained by looking at both the lists L_i and A_2. This process is continued until either all the points have been shifted, or the next point considered is to remain in the processor. The procedure for determining the next point is given below and is illustrated in Fig. 4.4.

  (2.1) Pick the point p with the current least c_l coordinate from the lists L_i and A_2.

  (2.2) If the point p is from A_2, go to Step (2.3), else do the following. Add p to B^i, and also delete p from L_i. If p is not present in B_2, exit from the procedure with p as the next point. In the other case delete p from B_2 and go to Step (2.1).

  (2.3) Delete p from A_2, exit from the procedure with p as the next point.

(3) If $f(i+1,n,n_l) + \alpha_1 + \alpha_2 - \beta_1 - \beta_2 > f(i+1,n+\alpha-\beta,n_l)$ then the processor has to send information to the right. This is done in a manner similar to Step (2). In this case, the next point is obtained



The lists L_i and A_2 are scanned in increasing order of c_l coordinate, ignoring elements from L_i that are also in B_2.

Figure 4.4. Step 2 of the *decompose* algorithm

by first picking the point with the highest c_l coordinate from R_i and A_2 and following a procedure similar to Steps (2.1)..(2.3).

(4) If $f(i,n,n_l)+\alpha_1-\beta_1 > f(i,n+\alpha-\beta,n_l)$ then the processor receives messages from the left. These points are added to the list A^i.

(5) If $f(i+1,n,n_l)+\alpha_1+\alpha_2-\beta_1-\beta_2 < f(i+1,n+\alpha-\beta,n_l)$ then the processor receives messages from the right. These points are added to the list A^i.

(6) Combine the lists A^i and A_2 into a single list A^i. Similarly combine the lists B^i and B_2 into a single list B^i. The sets A^i and B^i give the sets of points that need to be added and deleted from D_i.

end *decompose*

This completes our description of the algorithm. We now have the following lemmas.

Lemma 4.2. The above algorithm correctly decomposes the *add_delete* (A,B) operation on region D into *add_delete* (A^i,B^i) operations on its sub regions D_i.

Proof: The above algorithm is similar to the algorithm for slice partitioned data structures, except that we only have the set of points *lset* (D_i), *rset* (D_i) at processor p_{D_i}, instead of the whole set of points in D_i. We prove that this information is sufficient.

Case 1. $\alpha-\beta \geq 0$. In this case, the maximum number of points to be sent to the left is $f(i,n+\alpha-\beta,n_l)-(f(i,n,n_l)+\alpha_1-\beta_1)$. The worst case occurs when all these points come from *lset* (D_i). We also have to consider the β_2 deletions. Thus, we need to have at the most

$$f(i,n+\alpha-\beta,n_l)-(f(i,n,n_l)+\alpha_1-\beta_1)+\beta_2$$

points from *lset* (D_i). This, from Lemma 4.1, is

$$\leq \alpha-\beta-\alpha_1+\beta_1+\beta_2 \leq \max(\alpha,\beta),$$

proving the result.

The maximum number of points to be sent to the right is $f(i+1,n,n_l)+\alpha_1+\alpha_2-\beta_1-\beta_2-f(i+1,n+\alpha-\beta,n_l)$. We also have to consider the β_2 deletions. Thus we need in the worst case

$$f(i+1,n,n_l)+\alpha_1+\alpha_2-\beta_1-\beta_2-f(i+1,n+\alpha-\beta,n_l)+\beta_2$$

$$\leq \alpha_1+\alpha_2-\beta_1 \leq \alpha-\beta \leq \max(\alpha,\beta)$$

points from *rset* (D_i), proving the result.

Case 2. $\alpha-\beta<0$. The proof of this case is similar to Case 1.

□

Lemma 4.3. $|A^i|,|B^i| \leq \max(\alpha,\beta)$. The maximum number of points received from the left and right is $\max(\alpha,\beta)$. The maximum number of points sent to left and right is $\max(\alpha,\beta)$.

Proof: We only prove $|A^i| \leq \max(\alpha,\beta)$. The proof that $|B^i| \leq \max(\alpha,\beta)$ is similar. The maximum number of points a processor can receive from the left is

$$f(i,n,n_l)+\alpha_1-\beta_1-f(i,n+\alpha-\beta,n_l).$$

The maximum number of points it can receive from the right is

$$f(i+1,n+\alpha-\beta,n_l)-(f(i+1,n,n_l)+\alpha_1+\alpha_2-\beta_1-\beta_2).$$

The worst case size of A^i is the maximum number of points received from left and right added to the α_2 additions in list A_2. If the processor receives points only from the left, then the worst case size of A^i is given by

$$f(i,n,n_l)+\alpha_1-\beta_1-f(i,n+\alpha-\beta,n_l)+\alpha_2.$$

If $\alpha-\beta\geq 0$, from Lemma 4.1, this is at most $\alpha_1+\alpha_2-\beta_1\leq\max(\alpha,\beta)$. If $\alpha-\beta<0$, from Lemma 4.1, this is at most $\beta-\alpha+\alpha_1-\beta_1+\alpha_2\leq\beta\leq\max(\alpha,\beta)$. Similarly, if the processor receives points only from the right, it can be proved that $|A^i|\leq\max(\alpha,\beta)$. If a processor receives points from left and right, then the worst case size of $|A^i|$ is given by

$$\beta_2+f(i+1,n+\alpha-\beta,n_l)-f(i+1,n,n_l)-f(i,n+\alpha-\beta,n_l)+f(i,n,n_l).$$

If $\alpha-\beta\geq 0$, from Lemma 4.1, this is at most $\beta_2+\alpha-\beta\leq\alpha\leq\max(\alpha,\beta)$. If $\alpha-\beta<0$, from Lemma 4.1 after rearranging the terms, this is at most $\beta_2-\left\lfloor(\beta-\alpha)/n_l\right\rfloor\leq\beta\leq\max(\alpha,\beta)$.

This proof also shows that the total number of points received from the left and right is at most $\max(\alpha,\beta)$. The proof that $|B^i|\leq\max(\alpha,\beta)$ will also show that the total number of points sent to left and right is at most $\max(\alpha,\beta)$.

□

Lemma 4.4. If $|n|\geq\max(\alpha,\beta)n_l$, then all the communication steps of processor p_{D_i} will be to processors $p_{D_{i-1}}$, $p_{D_{i+1}}$.

Proof: If a processor p_{D_i} sends a message to processor $p_{D_{i+2}}$, then we necessarily must have $f(i+1,n,n_l)+\alpha>f(i+2,n+\alpha-\beta,n_l)$. In case $\alpha-\beta\geq 0$, applying Lemma 4.1 we have,

$$f(i+2,n+\alpha-\beta,n_l)\geq f(i+2,n,n_l)\geq f(i+1,n,n_l)+\left\lfloor n/n_l\right\rfloor.$$

Combining the two inequalities, we have $\alpha>\left\lfloor n/n_l\right\rfloor$, implying $n<\max(\alpha,\beta)n_l$. In case $\alpha-\beta<0$, applying Lemma 4.1 we have,

$$f(i+1,n,n_l) \le f(i+2,n,n_l) - \left\lfloor n/n_l \right\rfloor$$

$$\le f(i+2,n+\alpha-\beta,n_l) + \beta - \alpha - \left\lfloor n/n_l \right\rfloor$$

Combining the two inequalities, we have $\alpha > \alpha - \beta + \left\lfloor n/n_l \right\rfloor$. Thus we have $n < \max(\alpha,\beta)n_l$.

Similarly, we can prove that a processor p_{D_i} sends a message to processor $p_{D_{i-2}}$ only if $n < \max(\alpha,\beta)n_l$. This completes the proof of the lemma.

□

Let W be the set of points being added (deleted) from the distributed data structure. It is sent as an *add_delete* (W,ϕ) (*add_delete* (ϕ,W)) operation to processor p_U. Let $w = |W|$. The complete algorithm for updating the distributed data structure is described below.

procedure *gen2_add_delete*

(1) For all regions D at level l, $1 \le l \le h$, the sets *lset*(D) and *rset*(D) (i.e w minimum and maximum points in the c_{l-1} coordinate) are determined and stored in processor p_D. This is done in a recursive fashion as described in part 2(b) of the *gen1_add(delete)* algorithm, except that the computation for all the k coordinates are done together.

(2) The algorithm *decompose* is invoked to convert the *add_delete* problem at level 0, to *add_delete* problems at level 1. This is then repeated to recursively convert *add_delete* problems at level l to *add_delete* problems at level $l+1$. This process is continued till level h, where the actual additions and deletions are done into the data structures.

end *gen2_add_delete*

4.4. Analysis for a Hypercube

Let d be the dimension of the hypercube. The number of processors $P=2^d$. Thus each of the numbers n_l, $0 \le l \le h-1$ is a power of two. In order to analyze the above algorithms, we have to know how the processors are assigned to the regions. This will be most efficient if it is done in a binary

reflected gray code fashion [27]. The idea in this assignment is that processors associated with adjacent sub regions of a region are connected directly by a link. A gray code mapping is illustrated in Fig. 4.5. The association of final regions to processors is shown directly in the figure, and the association of the regions at higher levels is shown beside.

Let w be the number of points being added or deleted from the distributed data structure. We first analyze the *gen1_add(delete)* algorithm. Step 1 of the algorithm is accomplished in d message passing steps, with message sizes at most $O(w)$. The local additions and deletions can be done in $O(kw\log(N/P))$ time Thus the total time for Step 1 is $O(dt_\alpha + dwt_\beta + kw\log(N/P))$, where t_α is the message latency time, and t_β is the time for transmitting a byte.

Let D be a region at level l, $1 \le l \le h$. We observe that the points sent from (received by) p_D, during the rebalancing operation on its parent region, is exactly the same as the points sent from (received by) p_D in the *decompose* step of the second algorithm. Moreover, the processors communicated by p_D is the same in the two steps. Thus, applying Lemma 4.3, we see that the number of points sent from (received by) p_D is $O(w)$. Also, applying Lemma 4.4, we see that the communication will be restricted to neighbors if $N \ge w2^d$, which is assumed in this analysis.

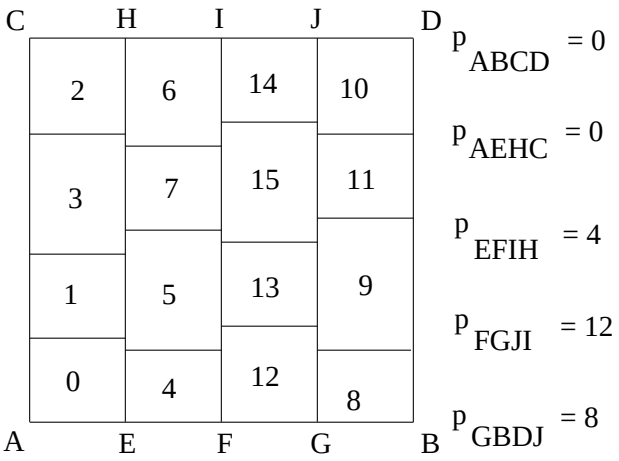


Figure 4.5. Gray code mapping

Step 2(b) can be done by an inverse broadcast tree. The information for regions at level h is obtained in time $O(w+\log(N/P))$. The message size is $O(w)$, the number of message passing stages is $O(d)$. Step 2(c) is done in a reverse manner compared to Step 2(b). There are $O(d)$ message passing stages, and the message size is $O(w)$. The time for addition and deletion from the local data structure is $O(kw\log(N/P))$. The time for rebalancing all regions at a particular level is hence $O(dt_\alpha + dwt_\beta + kw\log(N/P))$. Thus the total time for Step 2, for rebalancing regions at all levels is, $O(hdt_\alpha + hdwt_\beta + hkw\log(N/P))$ Thus we have the following theorem.

Theorem 4.1. For the *gen1_add(delete)* algorithm, if $N \geq w\,2^d$, the time taken by an add or a delete operation of w points is $O(hdt_\alpha + hdwt_\beta + hkw\log(N/P))$.

A similar analysis can be done for the *gen2_add_delete* algorithm. Step 1 can be done by an inverse broadcast tree using d steps, with message size $O(kw)$. The initial information for regions at level h is obtained in time $O(k(w+\log(N/P)))$. Thus the time taken by Step 1 is hence $O(k(w+\log(N/P)) + dt_\alpha + dkwt_\beta)$.

Step 2 of the algorithm involves h stages. By similar arguments as before, we see that if $N \geq w\,2^d$, then the time taken by the *decompose* step over all the h levels is $O(hw\log w + ht_\alpha + hwt_\beta)$. The first term is because of the sorting steps in the *decompose* algorithm. Finally, we have to consider the addition and deletion of points from the local data structure, which takes $O(kw\log(N/P))$ time. Thus we have the following theorem.

Theorem 4.2. For the *gen2_add_delete* algorithm, if $N \geq w\,2^d$, the time taken by an add or a delete operation of w elements is $O(dt_\alpha + dkwt_\beta + kw\log(N/P) + hw\log w)$.

From the above theorems, we observe that the performance of the *gen2_add_delete* algorithm is better than the performance of *gen1_add(delete)* algorithm, for the case $h \gg k$.

4.5. Implementation and Results

The *gen2_add_delete* algorithm has been implemented on the Intel iPSC2/D4-MX hypercube, using roughly 1400 lines of C code.

In this section, we present the implementation results for the *gen2_add_delete* algorithm. Specifically, we have chosen three *RP* data structures on 16 processors. For all the data structures the value of k is 3, i.e we are dealing with points in three dimensions, i.e E^3. The first data structure is called the slice partitioned data structure. It is described by $h=1$, $c_0=1$, and n_0=16. The second data structure is called the rectangle partitioned data structure. It is described by $h=2$, $c_0=1$, $n_0=4$, $c_1=2$, and $n_1=4$. The third data structure is called the cube partitioned data structure. It is described by $h=3$, $c_0=1$, $n_0=2$, $c_1=2$, $n_1=4$, $c_2=3$, and $n_2=2$.

The results of the algorithm for the three partitioned data structures are shown in Tables 4.1 through 4.3. These tables give the times (in milliseconds) for addition and deletion of w points into the distributed data structure that already contains N points. The times were obtained as the average over 50 repetitions of the operations, each time with a different set of w points to be added or deleted. The points to be added or deleted were obtained in a random fashion from the domain. From the

N in 1000	w (#add/del)							
	100		200		500		1000	
	Add	Del	Add	Del	Add	Del	Add	Del
10	144	148	184	213	303	530	451	1423
20	146	147	183	220	307	563	520	1387
50	144	149	184	214	307	525	524	1355
100	144	149	184	216	310	520	530	1342
200	145	151	184	212	311	543	529	1565
500	146	149	186	216	311	519	530	1525

Table 4.1. Time (in ms) for slice partitioned data structure

N in 1000	w (#add/del)							
	100		200		500		1000	
	Add	Del	Add	Del	Add	Del	Add	Del
10	202	203	299	311	601	709	958	1468
20	205	206	301	318	608	736	1155	1580
50	205	207	302	321	614	755	1133	1551
100	204	211	325	323	627	702	1169	1501
200	217	222	307	338	625	731	1142	1666
500	209	208	308	324	651	729	1149	1598

Table 4.2. Time in (ms) for rectangle partitioned data structure

tables, we observe that the times for addition and deletion depends primarily on the number of points being added or deleted, but not so much on the number of points already in the distributed data structure.

N in 1000	w (#add/del)							
	100		200		500		1000	
	Add	Del	Add	Del	Add	Del	Add	Del
10	269	273	449	442	928	1008	1490	1892
20	276	277	435	447	934	1016	1765	2129
50	276	278	452	461	939	1007	1800	2108
100	278	281	449	460	958	1027	1790	2110
200	279	280	444	453	944	1028	1796	2120
500	280	283	448	457	952	1019	1805	2143

Table 4.3. Time (in ms) for cube partitioned data structure

4.6. Extensions

In this section, we extend the algorithms presented in Section 4.3 for better average case performance. In other words, we investigate algorithms which take advantage of the statistical distribution of the points being added or deleted.

The second algorithm can be changed for better average case performance. The main drawback is that irrespective of the input, the algorithm computes for each region D the sets $lset(D)$, $rset(D)$, which are the w minimum and maximum points. Some of these points may not be used in Step 2 of the algorithm, and this represents a waste, especially for large values of w.

To solve this problem, we return to the context of the *decompose* algorithm. From the proof of Lemma 4.2, we know that processor p_{D_i} needs to have at most

$$f(i, n+\alpha-\beta, n_l)-(f(i, n, n_l)+\alpha_1-\beta_1)+\beta_2$$

minimum points of region D_i. A similar expression gives the number of maximum points from region D_i that is sufficient. The required number of minimum and maximum points can be obtained as in Step 2 of the *gen1_add(delete)* algorithm. However, as this information is obtained, we have also obtained the same information for regions at lower levels, which could be used later.

Now let us return to context of the *decompose* algorithm. Suppose the information of the minimum c points and the maximum d points of region D_i, based on the c_l coordinate, is available at p_{D_i}. Also, assume that the *decompose* algorithm needs the minimum a points and the maximum b points from region D_i. We need to get more points only in the case when $a > c$ or $b > d$.

Consider the case when $a > c$ as an example. Thus the information of the points in D_i with index varying from $c+1$ through a is needed at p_{D_i}. Let x be c_l coordinate of the point in D_i, whose index is c. The required information is first obtained by computing the $c-a$ minimum points in the sub regions of D_i that have c_l coordinate value greater than x. This information is then recursively combined as in Step 2 of *gen1_add(delete)* algorithm. At the end of this operation, for every sub region (at different levels) of R_i, the information of the minimum c points in the region is available at

its associated processor. Thus the above scheme can be continued at lower levels.

Except for the message latency term, which is $O(hdt_{\alpha})$ in the worst case, the worst case complexity of the algorithm described above is the same as the *gen2_add_delete* algorithm. However, its average case complexity will be better than the *gen2_add_delete* algorithm.

4.7. Summary

A recursively partitioned data structure involves partitioning a set of points in k dimensional Euclidean space E^k. This strategy is specified hierarchically in terms of levels. At each level, we specify how the region has to be split, and the number of parts it is divided into. The objective is to have a near about equal distribution of points in the final regions. In this chapter, we described how such a recursively partitioned data structure can be maintained dynamically on a multiprocessor, under the operations of additions and deletions of points. We analyze the algorithms and present results on a hypercube multiprocessor.

CHAPTER 5.

TRADEOFFS IN PARALLELISM AND HIERARCHY

The computational requirements of computer aided design problems are ever increasing. One approach to handling this complexity involves the use of parallel processing. This was outlined in Chapter 1. Another approach to this problem is to take advantage of the hierarchical nature of circuits. In Section 5.1, we look at the use of hierarchical methods for solving different problems in computer-aided design. In Section 5.2, we discuss how the orthogonal approaches of hierarchy and parallelism can be combined to speed up various computer aided design applications. Towards this goal, we formulate a general problem in scheduling. Approximate algorithms for this problem is the subject matter of Chapters 6 and 7. In Section 5.3, we report on an application of the theory for the problem of circuit extraction.

5.1. Use of Hierarchy

This approach exploits the structure available in a hierarchical representation of a circuit. VLSI circuits are typically specified and designed in a hierarchical fashion in terms of cells. The hierarchical representation of an example circuit is shown in Fig. 5.1. Each cell consists of some primitives and some sub cells. Hierarchical methods speed up applications by performing the appropriate operation on a cell only once, even though it is used many times, and appropriately modifying it wherever it is used as a sub cell. These methods also speed up CAD applications on account of reduction of the size of the problem considered at any instant, and thus represent a divide and conquer technique. Some example CAD problems, for which hierarchical methods are useful, are placement and routing, compaction, extraction and design rule checking.

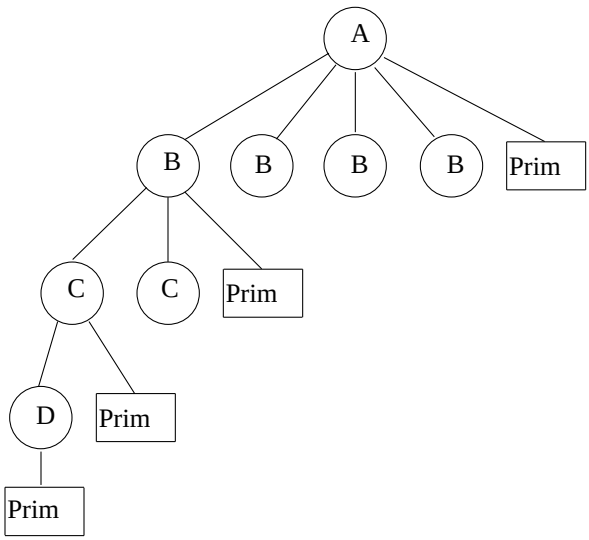


Figure 5.1. Hierarchical representation of a circuit

The placement problem consists of mapping the components in a structural description onto positions on a layout surface, so that various parameters such as area, wire length are minimized. Hierarchical analysis and partitioning can be used to speed up the placement problem. In hierarchical placement [35], the placement is done following the hierarchy in the structural specification. Placement is first carried out for lower level cells, the results of which are used in the placement of cells at the next higher level and so on. In partitioning based placement [36], the flat placement problem is subdivided into separate but interrelated problems. Using min-cut partitioning methods such as Kernighan Lin's algorithm [37], the set of primitive blocks are split recursively till a certain level. The resulting structure obtained is a slicing tree. The final partitions obtained represent "super nodes". The primitive blocks in a super node are placed using a standard placement algorithm. The placement problem in different super nodes are, however, interrelated as the placement decision in one partition effects the placement decision in other partitions. Since partitioning methods are very similar to hierarchical methods, we let the term hierarchy to also include the case of partitioning.

Once a placement is obtained, the routing problem consists of constructing the required connections between components. As in the case of placement, hierarchy and partitioning based techniques can be used to speed up routing. In hierarchical routing, the routing is done following the structural specification. In partitioning based routing [38, 39], the routing problem in a region is subdivided into routing problems on its sub regions. The solution to an integer programming problem, with constant number of variables and constraints, is used to guide the assignment of each net (connection) to a path of sub regions. This is done so that the capacity constraints at the crossing between sub regions are satisfied, and also so that nets are routed as much as possible through shortest paths.

The problem of compaction involves taking a preliminary layout of a circuit, and transforming it into a smaller layout, that is also design rule correct, by moving wires and components. There are two approaches for hierarchical compaction, depending on whether the sub cells are fixed or can be stretched. In the fixed cell approach [40], the leaf cells are first compacted. The positions of the ports, and a protection frame for the cell are then obtained. The protection frame consists of material which is within a maximum design rule distance from the boundary. It is used to determine the spacing between sub cells, during compaction at a higher level. The interconnections between sub cells are obtained by routing. In the non fixed cell approach [41], port abstractions of leaf cells are first computed. These determine how the ports of a cell can move relative to each other, maintaining a feasible solution. This information is used, with the protection frames, to carry out the compaction at the higher level. The leaf cells are then compacted, with additional constraints on the port positions.

A final example for use of hierarchy are the related problems of circuit extraction and design rule checking. A design rule checker determines if the input layout satisfies all spacing, width, enclosure and extension rules. Hierarchical circuit extractors [22], and design rule checkers [42], perform the appropriate operation on a cell only once. In the case of extraction, the information of a cell that is needed at a higher level consists of its ports, and their assigned electrical node names. In the case of design rule checking, each cell is checked ignoring errors that fall in its protection frame. This is

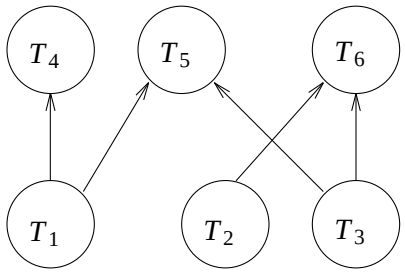
because these errors could be corrected at a higher level. The information that is needed at the higher level consists of the protection frame for the cell. In both hierarchical extraction and design rule checking, the input circuit is first transformed into one where the cells do not overlap, before applying the procedure described above.

5.2. Combining Hierarchy and Parallelism

Hierarchical analysis, as described in the previous section, subdivides a task into a set of sub tasks. These sub tasks may have dependencies between them. For example, in hierarchical placement, the placement of a higher level cell can be performed only after the placement of its sub cells are completed. In case of circuit extraction, as we shall see in Section 5.3, the subdivision can be done so that the sub tasks are independent.

The set of sub tasks with dependency relations can be represented by a directed acyclic graph, called the task graph, where the nodes correspond to the sub tasks, and the arcs correspond to the precedence constraints. For sake of convenience, we will refer to sub tasks as tasks, from now on. An example task graph is shown in Fig. 5.2(a). Referring to the figure, task T_4 can start only after T_1 is completed, T_5 can start only after T_1 and T_3 are completed, and T_6 can start only after T_2 and T_3 are completed.

To exploit the benefits of parallel processing, the set of tasks are to be scheduled onto a set of processors. The objective is to minimize the overall completion time. One approach to solve this problem is to use an algorithm called the critical path algorithm [43]. In this algorithm, for each task node, we determine the length of the longest path (i.e add the times of the task nodes in the path) in the task graph starting from the task node. This is assigned as the priority value of the task. The schedule is then constructed by iteratively choosing a schedulable task (i.e its predecessor tasks have completed) with the highest priority and assigning it to a free processor.



(a)

τ Task

5	5	10	20	5	10

σ Task

Proc						
1	1	1	1	1	1	1
20/11	20/11	20/11	20/11	20/11	20/11	20/11
5/2	5/2	5/2	5/2	5/2	5/2	5/2
3	3	40/13	40/13	3	40/13	
3	3	25/7	25/7	3	25/7	

(b)

Figure 5.2. (a) Example task graph (b) Time and speedup functions for tasks

The above approach has the limitation that the final schedule time is bounded below by the length of the longest path in the task graph, called the critical path. Thus it may be advantageous to assign more than one processor to a task, as this reduces the critical path time. For example, in placement, it may be beneficial to assign more than one processor to the placement task for a cell with a large number of blocks. One could use an available parallel placement algorithm to perform the placement of a cell in parallel. Similarly, in extraction, it may be beneficial to assign more than one processor to the extraction task for a cell with a large number of rectangles. An available parallel extractor can then be used to perform this step in parallel. Thus, we need to consider the problem of scheduling a set of tasks onto a parallel processor, where each of the tasks can be parallelized. This problem is called the *parallelizable dependent task scheduling (PDTS)* problem.

The *PDTS* problem is formulated as follows. The input to the problem consists of a task graph. Let n be the number of tasks and p the number of processors. Associated with each task T_i, $1 \le i \le n$, we assume that an estimate of the time taken by the task τ_i on a single processor, and values $\sigma_i(j)$,

$1 \leq j \leq p$, that give the speedups that can be obtained by running the task on j processors, are known. The problem then is to schedule the collection of tasks on p processors so that the maximum completion time is minimized. A single task may be assigned more than one processor. An example problem instance is shown in Fig. 5.2.

Some discussion on the validity of the assumptions in the problem formulation follow. The formulation requires that an estimate of the time be provided for every task. Such assumptions are commonly used in the task scheduling and load balancing literature [44, 45]. This is usually possible as the following examples from VLSI CAD applications show. For placement, the time taken for min-cut based placement methods can be given as $O(n^3)$, where n is the number of components to be placed. The order information is sufficient since the constant factor can be factored out from the times for all the tasks. If simulated annealing is used for placement, the time taken can be given as $O((\sum_{i=1}^{T} iter_i(n))\, d(n))$. Here, $i = 1,2, \cdots T$ represents the different temperature values, and $iter_i(n)$ gives the number of iterations performed at the i^{th} temperature, and $d(n)$ gives the time to perform a single move. For compaction, extraction and design rule checking the time taken can be estimated as $O(n \log n)$, where n is the number of rectangles. For routing, the time taken can be estimated as $O(n \log n)$, where n is the number of nets. The problem formulation also requires the availability of the speedup values. These can be estimated by running the corresponding parallel algorithm on different number of processors.

In Chapters 6 and 7, we present approximate algorithms for the *PDTS* problem. In Chapter 6, we consider the general case of the problem. In Chapter 7, we consider a special case of the problem, when the tasks are independent.

5.3. Case Study : Circuit Extraction

In this section, we report on an actual implementation of our approach of combining hierarchy and parallelism for one typical VLSI CAD application, namely circuit extraction.

5.3.1. Hierarchical Circuit Extraction

Hierarchical extractors exploit the repetitive information in a hierarchical representation of a circuit by extracting a cell that is used many times only once and appropriately modifying it wherever it is used. The difference in analysis time obtained by using hierarchical methods will vary depending on the regularity of the circuit. 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 [46] forbid any overlapping of cells. However, this is inconvenient and unnecessarily restrictive. Other extractors like [47, 23] transform the circuit with overlapping cells to a circuit in which there are no overlapping cells.

Thus, there are two steps in hierarchical circuit extraction. The first step is the transformation step. The second step is the actual extraction step. The first step involves transforming a circuit that has overlapping cells into one that does not have any overlapping cells. One of the ways of achieving this involves looking at the sub cells of each cell, and instantiating a sub cell A if it overlaps another sub cell or geometry. The instantiation process involves replacing the cell A with its sub cells and geometry. The whole process is repeated until there are no overlapping sub cells in each of the cells. At the end of this step, some cells that are not useful are discarded. A cell is useful if is the root cell, or it occurs as a sub cell of an useful cell. In the worst case, the transformation step may convert the whole circuit into a single cell.

At the end of the first step, the job of extraction consists of extracting each of the cells that remain and appropriately combining the results using common node names. For each cell, the information obtained before the extraction step consists of its ports, the geometry in terms of rectangles, and ports of its sub cells. The ports are represented as rectangles which have a span of zero along one

dimension. With this representation, the extraction of a cell is done in a similar manner as flat circuit extraction.

5.3.2. Parallel Hierarchical Extraction

The transformation step of hierarchical circuit extraction can be parallelized by assigning cells to different processors dynamically from a list. The cells are ordered so that sub cells of a cell occur before a cell. A processor analyzing a cell determines the sub cells of the cell that overlap each other or some geometry. For each such sub cell, the processor waits for the analysis of the sub cell to be completed, before instantiation. The amount of parallelism obtained in this step depends on the circuit and is generally not more than a factor of 2, implying this step is essentially a sequential step. Fortunately, the second step of hierarchical circuit extraction is the most time consuming step. There are two approaches we could take for parallelizing this step.

5.3.2.1. Algorithm LPT

One approach to parallelizing the second step involves assigning the extraction of cells to processors dynamically from a list. The list of cells could be obtained by sorting the cells in decreasing order of the number of rectangles it contains. This algorithm corresponds to the *LPT* algorithm (largest processing time first) [48] for scheduling independent tasks on multiprocessors. In this algorithm, whenever a processor is free it takes the next available task from a list of tasks. The list of tasks is initially sorted in decreasing order of execution times.

The disadvantage with the above approach is that some of cells could be very large. A large cell assigned to a single processor will be a bottleneck for the whole computation, reducing the parallelism that can be obtained. Thus this algorithm will not speedup the extraction in all cases.

5.3.2.2. Algorithm PITS

The second approach is to apply the theory of parallelizable independent task scheduling (*PITS*). For our application, the different tasks represent the different cells that need to be extracted. The time taken for a cell is estimated from the number of rectangles in the cell. The speedup function is assumed to be the same for all the cells, and is given by the speedup curve obtained by the parallel algorithm discussed in Chapter 2. The speedup curve used in the implementation is shown in Fig. 5.3.

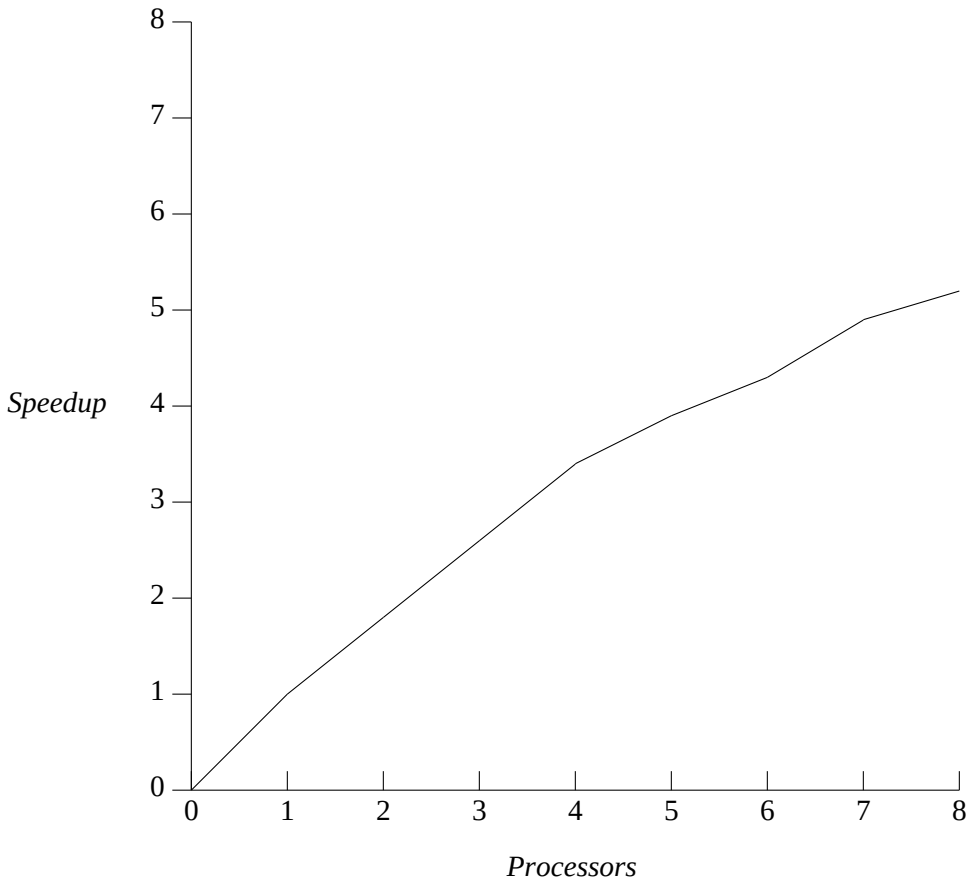


Figure 5.3. Speedup curve used in the implementation

The schedule given by the *fast_part_schedule* algorithm, to be discussed in Chapter 7, in conjunction with the parallel algorithm for circuit extraction, described in Chapter 2, is used to solve the second step of hierarchical circuit extraction. This approach is very modular, as we could interface the scheduling algorithm to any other parallel algorithm for circuit extraction.

5.3.2.3. Results

The parallel hierarchical algorithms (named *PHACE*), described in the previous section, have been implemented on an Encore 510 shared memory multiprocessor consisting of 8 processors, using approximately 7000 lines of C code. In Table 5.1, we give the time taken in seconds on some example layouts, which are the same as the ones described in Chapter 2, for five algorithms. These are the sequential flat algorithm, parallel flat algorithm, sequential hierarchical algorithm, parallel hierarchical algorithm based on *LPT* algorithm and parallel hierarchical algorithm based on *PITS* algorithm. The times reported exclude the time for input and output. The parameter model used is the *HPEX* model [23]. In Table 5.2, we give the breakup of the time taken for hierarchical extraction for both the sequential case and the parallel case with *PITS* scheduling.

Circuit	Flat Sequential	Flat Parallel (8 processors)	Hierarchical Sequential	Hierarchical Parallel(*LPT*) (8 processors)	Hierarchical Parallel(*PITS*) (8 processors)
Hypercube Router Chip	217	63	200	192	63
Multiplier Array	246	66	240	217	61
Static Ram	346	103	170	126	38
Placement Coprocessor	763	218	185	125	40
Associative Cache	1062	268	1045	1030	222

Table 5.1. Time taken in seconds by the 5 algorithms with *HPEX* model

Circuit	Hierarchical Sequential		Hierarchical Parallel(*PITS*) (8 processors)	
	Step 1	Step 2	Step 1	Step 2
Hypercube Router Chip (54800)	27	173	22	41
Multiplier Array (66000)	21	219	21	40
Static Ram (130700)	11	159	8	30
Placement Coprocessor (258600)	13	172	9	31
Associative Cache (258700)	51	994	50	172

Table 5.2. Breakup of time for sequential and parallel hierarchical extraction

As can be observed in Table 5.1, for circuits 1, 2, 5 the use of hierarchy did not result in any improvement over the flat case. However, for circuits 3, 4 the use of hierarchy resulted in a good improvement. In each of cases, the *PITS* based algorithm performed the best (or close to the best). The parallel algorithm, with *LPT* scheduling performs poorly in most of the cases.

In Fig. 5.4, we graphically illustrate the working of the *LPT* based algorithm and the *PITS* based algorithm for the fourth circuit. We show the time taken by each of the processors. We also indicate the important cells that are extracted by each of the processors.

5.4. Summary

We have described how the benefits of hierarchical analysis and parallelism can be combined for a variety of problems in computer aided design. Towards this goal, we formulated a general problem in scheduling called parallelizable dependent task scheduling problem. The theory was then

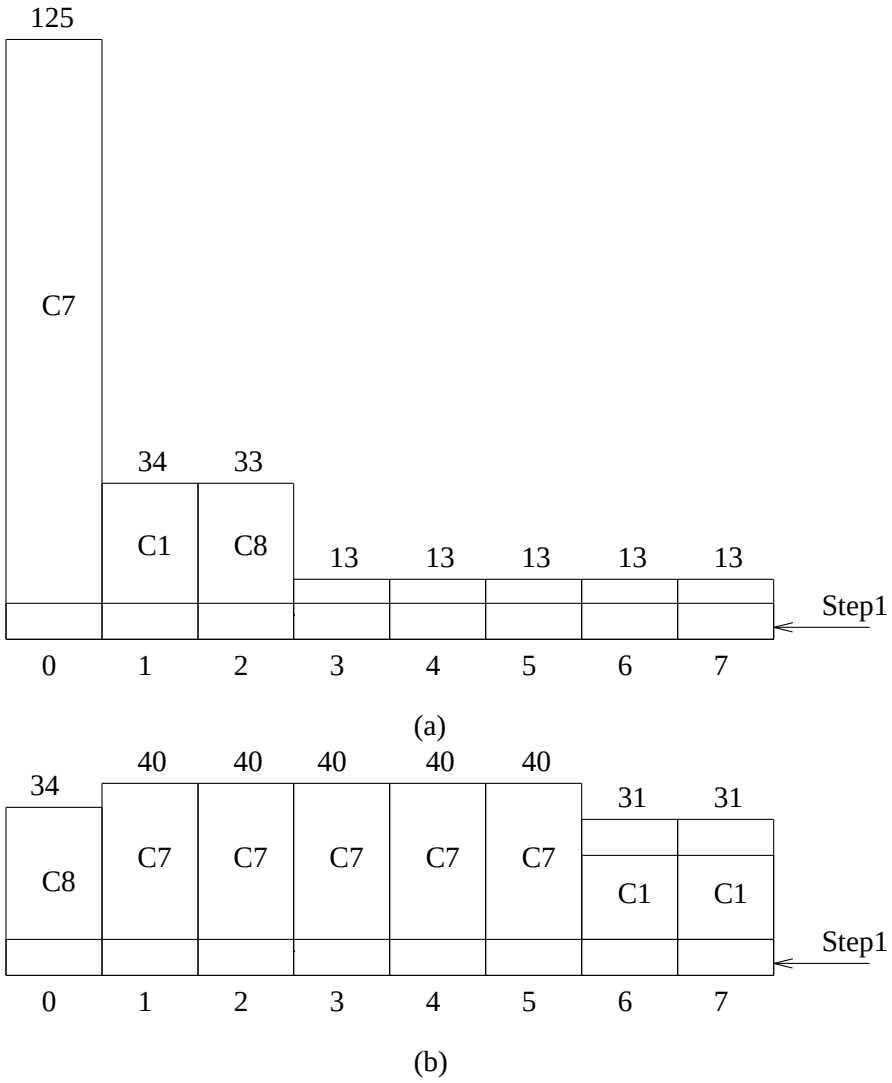


Figure 5.4. (a) *LPT* based (b) *PITS* based

applied for a case study in parallel circuit extraction. We present results for the case study on a Encore shared memory multiprocessor.

CHAPTER 6.

PARALLELIZABLE DEPENDENT TASK SCHEDULING PROBLEM

Scheduling a collection of tasks on a multiprocessor, that minimizes certain objective functions, has attracted a lot of attention in the literature. An excellent survey of a number of scheduling problems can be found in [44, 45]. Except in a few cases, most of the scheduling problems are NP-hard [49]. Thus we need to look at efficient heuristic algorithms for the problems. It would be preferable if the performance of the heuristic algorithms can be bounded with respect to the optimal performance. Such algorithms, called *approximate* algorithms, have been discovered for many problems in scheduling [44].

In this chapter, we consider the *parallelizable dependent task scheduling (PDTS)* problem, formulated in Chapter 5. We present an algorithm for the problem and analyze its performance. This problem arises in many other applications. For example, parallelizing compilers take an input program and construct the corresponding task graph [50]. Each of the task nodes could correspond to loops that could be parallelized across processors. The parallelizing compiler has to determine the number of processors to be allocated to each of the task nodes, and the schedule, to minimize the total execution time.

A schedule S is defined as a specification of tasks to processors. For each task, it specifies the starting time for the task and the processors that it should be run on. This specification should satisfy the precedence constraints of the task graph. Also, at any time instant a processor can be assigned to at most one task. A task T_i assigned to a set of processors can start only after all the processors in the set have finished the task assigned to them immediately before T_i. For a schedule S, let $f(S)$ denote the final finish time. The objective of the problem is to find the schedule S which minimizes $f(S)$.

An example schedule is shown by a "Gantt chart" in Fig. 6.1. Each processor is allocated a column containing rectangles that denote its tasks. The length of the rectangle corresponding to a task gives the time of execution of the task. The cross-hatched regions represent times during which a processor is idle.

6.1. Approximate Algorithm

Let $\eta_i(j) = \frac{\sigma_i(j)}{j}$. The condition that $\eta_i(j) \geq \eta_i(j+1)$, for all i, j, where $1 \leq i \leq n$ and $1 \leq j \leq p-1$, is called the *efficiency* condition. It is very likely that this condition holds, as it can be easily proved that if the speedup curve of a task has a concave shape then it satisfies the efficiency condition, though the converse is not true.

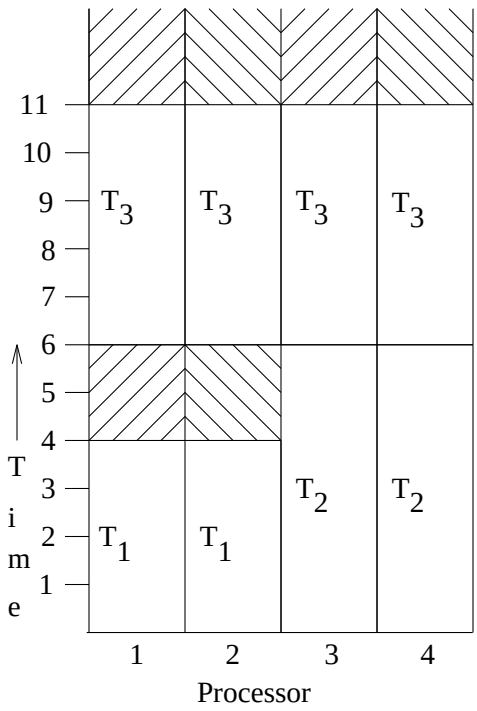


Figure 6.1. An example schedule

The *PDTS* problem is a NP-hard problem, as the simpler problem of scheduling tasks, where the tasks can run on only one processor is NP-hard. In this section, we give an algorithm for the problem. The algorithm has two steps. In the first step, called the allocation step, we determine the number of processors to be assigned to each of the tasks. The second step, called the mapping step, will be to actually construct the schedule from the output of the first stage. In the next two subsections, we describe the issues in both the steps.

6.1.1. The Allocation Step

One of the objectives in the allocation step is to minimize the critical path time of the task graph by assigning more processors to tasks, which tends to reduce the final schedule time. However, assigning more processors to tasks increases the average load on a processor, as the speedups obtained are less than linear, and hence tends to increase the final schedule time. For example, in Fig. 5.2(a), the average load and critical path time are 11 and 25 respectively. If T_4 is assigned two processors, its execution time will reduce to 11. The critical path time will reduce to 16. However, since T_4 has been replaced by two tasks taking 11 units of time each, the average load on a processor is increased to 11.4. Thus we need to assign processors so that the maximum of the critical path and the average load on a processor is minimized. This problem can be formulated by the following integer non linear program.

Minimize ϕ, *subject to the following constraints*

(1) $\phi \geq y_i, 1 \leq i \leq n.$

(2) $\phi \geq \frac{1}{p}\left(\sum_{1 \leq i \leq n} \frac{s_i \tau_i}{\sigma_i(s_i)}\right)$

(3) $y_i - y_j \geq \frac{\tau_i}{\sigma_i(s_i)}$, *for all* i, j, *where* T_i *depends on* T_j.

(4) $y_i \geq \frac{\tau_i}{\sigma_i(s_i)}$, *for all* $i.$

(5) $1 \leq s_i \leq p$ *and integral , for all* $i.$

Here, y_i gives the earliest finish time of task T_i, given an infinite number of processors. The variable s_i denotes the number of processors allocated to task T_i. The conditions in (1), (2) imply that we are minimizing the maximum of critical path time and the average. The conditions in (3) are due to precedence constraints between the tasks. It relates the earliest finish time of task T_i in terms of the earliest finish times of its predecessors.

The integer constraint of (5) could be omitted, leading to a non linear optimization problem. For this problem, we assume that the speedup functions $\sigma_i(x)$ are given as functions over reals. The solution to the non linear optimization problem can then be rounded to get a close answer to the integer non linear optimization problem described above. Thus we will concentrate on the non linear optimization problem in rest of this section.

Consider the case when the speedup functions are concave, which is usually the case. It can be easily proved that if $f(x)$ is concave and positive over a domain, then $1/f(x)$ is a convex function over the domain. Thus the functions $\frac{1}{\sigma_i(x)}$ are all convex. Thus constraints in (3) (4) are of the form $g(x) \leq 0$, where $g(x)$ is a convex function. The constraint (2) can also be cast in this form if the functions $\frac{x}{\sigma_i(x)}$ are all convex. This is a reasonable condition, as $\sigma(x)$ by Amdahl's law [51] is usually given as $\frac{1}{(1-\alpha)/x + \alpha}$, where α is the fraction of the program that is inherently serial. The function $\frac{x}{\sigma(x)} = (1-\alpha) + \alpha x$, which is convex. With this extra condition, the nonlinear optimization problem is essentially minimizing a convex function over a closed convex set. Hence, the optimum solution can be obtained by standard algorithms such as steepest descent algorithms [52]. If the two conditions described above are not valid, non linear optimization procedures can only guarantee a relative minimum.

6.1.2. Mapping Step

The second step is to construct the schedule from the processor allocation information provided by the allocation step. A fundamental issue in this step is the following. A task T_i can start only after all the tasks it depends on are completed. Also, the task T_i may be delayed further until the required s_i processors are available. Greater the value of s_i, greater will be the waiting time, which may result in an increase in the overall schedule time, especially if no other task can be scheduled during this waiting period. Thus it may be necessary to modify the processor allocation during the mapping step. Also, decreasing the value of s_i, while decreasing the waiting time, may increase the critical path time of the task graph, and hence may increase the overall schedule time.

We present a simple algorithm below for the mapping step, whose function depends on a parameter $0 < \gamma < 1$.

procedure *map*

[1] For all tasks T_i, set $a_i = s_i$, and if $a_i > \lceil \gamma p \rceil$ then reduce a_i to $\lceil \gamma p \rceil$.

[2] Build the schedule iteratively, by scheduling a new task T_i whenever all its predecessors have been completed, and a_i processors are available.

end *map*

The schedule constructed in step (2) has the property that, at any time instant, the number of idling processors is less than the number of processors required by any of the unfinished tasks that are ready to start (i.e its predecessor tasks have completed). In other words, there is no forced idleness. Also, the second step of the algorithm can be performed dynamically at run time, when the tasks are actually executed.

6.2. Analysis of the algorithm

The choice of γ is crucial for the performance of the *map* algorithm. The following theorem gives the value of γ that realizes the best worst case performance.

Theorem 6.1: Assuming that the efficiency condition is true, the value $\frac{(3-\sqrt{5})}{2} \approx .382$ is the best choice for γ. It ensures that the finish time $f(M)$ of the schedule M produced by the *map* algorithm is within $\left\lfloor \frac{p}{\frac{(\sqrt{5}-1)p}{2}} \right\rfloor A + C$, where A is the average and C is the critical path time of the task graph with the processor allocation as specified at the end of the allocation step.

Proof: Let us suppose that the cutoff used in step (1) of the algorithm is x. Consider the average load A' on a processor in schedule M, i.e $A' = (1/p) \sum_i \frac{a_i \tau_i}{\sigma_i(a_i)}$. Clearly, $s_i >= a_i$. Hence, from the efficiency condition, $A' \leq (1/p) \sum_i \frac{s_i \tau_i}{\sigma_i(s_i)} = A$.

For each task T_i, we define a value t_i based on the output of the allocation phase. This represents the earliest start time of T_i. Thus, the value of t_i is 0 for tasks which do not depend on other tasks, and $\max_j \{ t_j + \frac{\tau_j}{\sigma_j(s_j)} \mid$ there is an arc from T_j to $T_i \}$ otherwise.

Consider the schedule M. At any time t, we define a quantity $v(t)$ called the virtual time. Let U be the set of tasks that have not been started till time t, and let E be the set of tasks that are currently executing at time t. For each task T_i in E, let α_i denote the fraction of the task T_i that has been completed at time t. Then, $v(t)$ is defined as $\min\{ \min_{T_i \in U} t_i, \min_{T_i \in E} (t_i + \alpha_i \frac{\tau_i}{\sigma_i(s_i)}) \}$. Clearly, the value of $v(t)$ at the end of the schedule is C.

It should be noted that the virtual time function $v(t)$ is defined in terms of the task graph at the end of the allocation phase. The value $v(t)$ is a measure of the portion of the task graph that has been completed at time t. It can be easily verified that if $v(t) \geq t_i$, then by time t all the predecessor tasks of

task T_i have been completed in M.

We now use an amortization argument to prove the result of the theorem. Let $\mu = \frac{p}{p-x}$, and let $T = \mu A + C$. We want to find the smallest μ so that $f(M)$ is upper bounded by T. Consider any interval of time from t to $t + \Delta t$, during which the allocation of tasks to processors in M does not change. There are two cases.

Case 1: The virtual time does not increase in the interval. This implies the existence of a task T_i, with $v(t) = t_i$, that has not been executed till $t + \Delta t$. Since $v(t) = t_i$, all predecessor tasks of T_i have finished execution by time t. Thus T_i is ready for execution at time t. Since T_i requires at most x processors, it implies that at least $p - x$ (actually $p - x + 1$) processors are busy during the interval. This is illustrated in Fig. 6.2(a). Thus the average load of all the processors during the interval is at least $\Delta t \frac{p-x}{p}$. Thus the time Δt can be charged to $\frac{p}{p-x} A' = \mu A'$. However, since $A' \leq A$, Δt can be

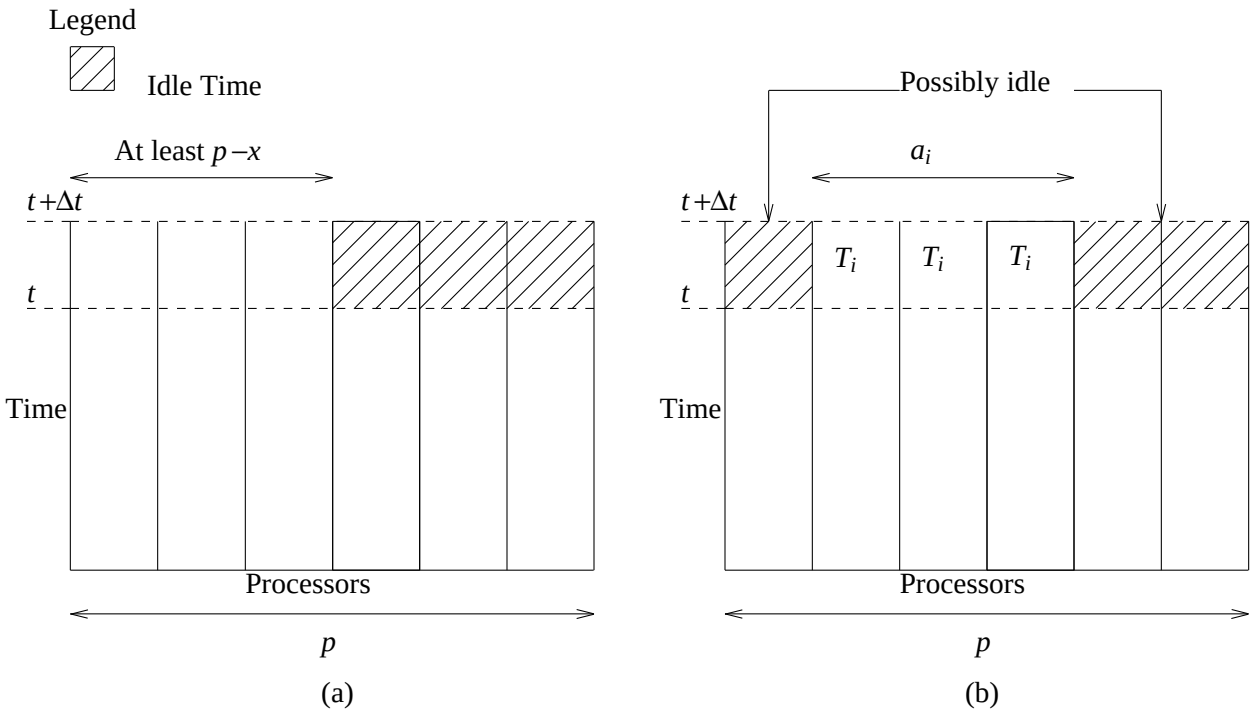


Figure 6.2. (a) Case 1 (b) Case 2 of Theorem 6.1

charged to μA.

Case 2: The virtual time increases in the interval. Without any loss of generality, we can assume that the increase is due to the execution of a single task T_i. This is illustrated in Fig. 6.2(b). The fraction of T_i that is executed in the interval is $\Delta\alpha_i = \frac{\Delta t}{\tau_i/\sigma_i(a_i)}$. Thus the increase in virtual time in the interval is $\Delta\alpha_i \frac{\tau_i}{\sigma_i(s_i)} = \Delta t \frac{\sigma_i(a_i)}{\sigma_i(s_i)}$. Since $a_i \le s_i$, applying the efficiency condition, we observe that the increase in virtual time is not less than $\Delta t \frac{a_i}{s_i}$.

If $s_i \le x$, then $a_i = s_i$. Thus the increase in virtual time is the same as Δt. Thus Δt can be charged to the increase in virtual time, and hence to the critical path C.

In the other case $s_i > x$, implying $a_i = x$. The increase in virtual time is at least $\Delta t \frac{x}{s_i}$. Thus Δt cannot be fully charged to the increase in virtual time. There is also a component due to the μA product. The contribution of task T_i to the average A, in the interval t to $t + \Delta t$, is $\Delta\alpha_i \frac{s_i \tau_i}{p\,\sigma_i(s_i)}$. Thus the virtual time increase, plus μ times the average, together provide a contribution that is at least $\Delta t(\frac{x}{s_i} + \frac{x\mu}{p})$. This has to be at least Δt. In other words, $\Delta t \le \Delta t(\frac{x}{s_i} + \frac{x\mu}{p})$. This can be guaranteed if $1 \le \frac{x}{p} + \frac{x\mu}{p}$. Substituting $\mu = \frac{p}{p-x}$, we solve for the smallest x (implying smallest μ) that satisfies the above condition. Simplifying, we get the inequality $x^2 - 3px + p^2 \le 0$. The smallest integer x satisfying the above equation is $\left\lceil p\frac{(3-\sqrt{5})}{2} \right\rceil$. This proves the theorem.

□

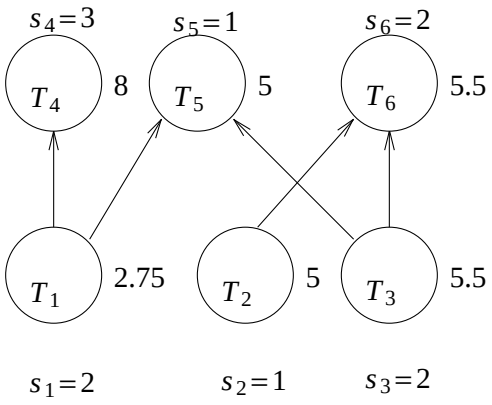
Thus, asymptotically the schedule time of the *map* algorithm is within $\frac{2}{\sqrt{5}-1}A + C$, that is within $1.62A + C$. It is very likely that we can design more sophisticated theoretical and practical algorithms for the mapping problem, which adjusts the number of processors allocated to a task at the time the schedule is being constructed, instead of having a maximum cut off. We are currently investigating algorithms of this nature. However, these algorithms may only allow for a static construction

of the schedule. The *map* algorithm described above represents a simple algorithm that shows the different tradeoffs involved in the mapping problem.

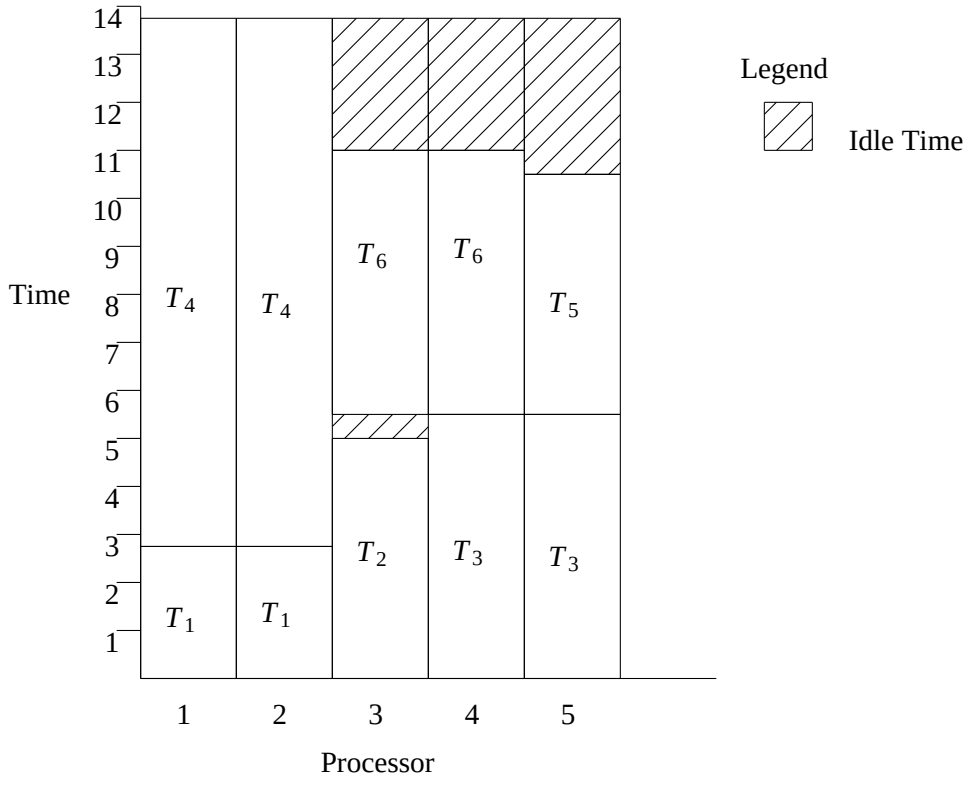
Finally, we want to analyze the complete algorithm for the *PDTS* scheduling problem. The time taken by an optimal schedule is at least $\max(A^{opt}, C^{opt})$, where A^{opt}, C^{opt} are the average and the critical path time corresponding to the optimal schedule. However, if the allocation step is carried out optimally $\max(A, C) \leq \max(A^{opt}, C^{opt})$, where A, C are the average and the critical path time of the task graph, with processor allocation as specified at the end of the allocation step. Thus combining these observations, we have the following theorem.

Theorem 6.2: If the allocation step is performed optimally and the efficiency condition is true, the schedule time of the complete algorithm is within $\left\lfloor \frac{(\sqrt{5}+1)p}{2} \right\rfloor \Big/ \left\lfloor \frac{(\sqrt{5}-1)p}{2} \right\rfloor \approx 2.62$ times the optimal schedule time.

Example 6.1: Consider the problem instance in Fig. 5.2. There are 6 tasks, and the task graph needs to be scheduled onto 5 processors. Initially, the average load and critical path time of the task graph are 11 and 25 respectively. In Fig. 6.3(a), we show a possible result of the allocation phase. The number of processors assigned to a task, and the corresponding time for the task, are shown in the figure. The average load and critical path time are respectively 12.3 and 10.75 respectively. In Fig. 6.3(b), we show the final schedule constructed by the *map* algorithm using a cutoff of $\lceil 0.382 * 5 \rceil = 2$. The final schedule length is 13.75. For comparison, if a task cannot be assigned to multiple processors, the optimal schedule time is 25.



(a)



(b)

Figure 6.3. (a) Allocation step (b) Mapping Step

6.3. Summary

In this chapter, we have described an algorithm for the *parallelizable dependent task scheduling* problem. We have analyzed its performance and derived some bounds with respect to the optimal, under some assumptions on the speedup functions. Future research on this problem should concentrate on improving the performance of the algorithms, both from theoretical and practical perspectives.

CHAPTER 7.

PARALLELIZABLE INDEPENDENT TASK SCHEDULING PROBLEM

In this chapter, we introduce a special case of the *PDTS* scheduling problem, in which the tasks are independent. We present *approximate* algorithms that guarantees a solution within $\frac{2}{(1+1/p)}$ of the optimal solution, provided the *efficiency* condition holds.

The organization of this chapter is as follows. In Section 7.1, we describe previous work on the problem, for the case when an individual task is restricted to run on a single processor. In Section 7.2, we describe *approximate* algorithms for the new scheduling problem and analyze their performance.

7.1. Previous Work

The problem described in this chapter has been considered before, for the special case when a task is restricted to run on a single processor. This problem is called the *independent task scheduling problem.* Even, for this special case, the problem is NP-hard for all $p \geq 2$. The best polynomial approximation algorithm known for this problem is the *MULTIFIT* algorithm [53]. This uses bin packing techniques to minimize the maximum completion time. The solution produced by this algorithm is provably within 1.22 of the optimal solution, and probably within 1.176 of the optimal.

In this section, we review another algorithm, called the *LPT* algorithm [48]. This algorithm generates a solution that is provably within $\frac{4}{3} - \frac{1}{3p}$ of the optimal. The algorithm also has a worst case performance that achieves this bound, i.e this bound is tight.

The *LPT* algorithm is one among a host of algorithms called the *list scheduling* algorithms. The basic idea in these algorithms is to consider tasks in order from a list, and build a feasible schedule. At any stage we have a partial schedule. Let F_j denote the current finish time of processor j. The complete algorithm is described below.

procedure *list_schedule*

[1] Set i to 1, and F_j to 0 for all $1 \leq j \leq p$.

[2] Choose a processor j with the smallest F_j among all processors (i.e the processor with the current minimum finish time).

[3] Assign T_i as the next task to processor j and update F_j to $F_j + \tau_i$.

[4] If $i \neq n$ increment i and go to 2. In the other case the algorithm terminates with overall schedule value equal to the maximum of all the F_j values.

end *list_schedule*.

The *list scheduling* algorithm is a fast algorithm and can be implemented in time $O(n \log p)$ time, given the list. The *LPT* (largest processing time) algorithm is a *list scheduling* algorithm that uses a list of tasks that is sorted in decreasing order based on execution times. Thus the *LPT* algorithm can be implemented in time $O(n \log n + n \log p)$ time. Let $f(S)$ denote the finish time of a schedule S. It was proved in [48] that for any instance of the *independent task scheduling* problem, the schedule A of the *LPT* algorithm, satisfies the condition $f(A) \leq (\frac{4}{3} - \frac{1}{3p}) f(OPT)$, where OPT is an optimal schedule. An example problem instance and the schedule constructed by the *LPT* algorithm are shown in Fig. 7.1.

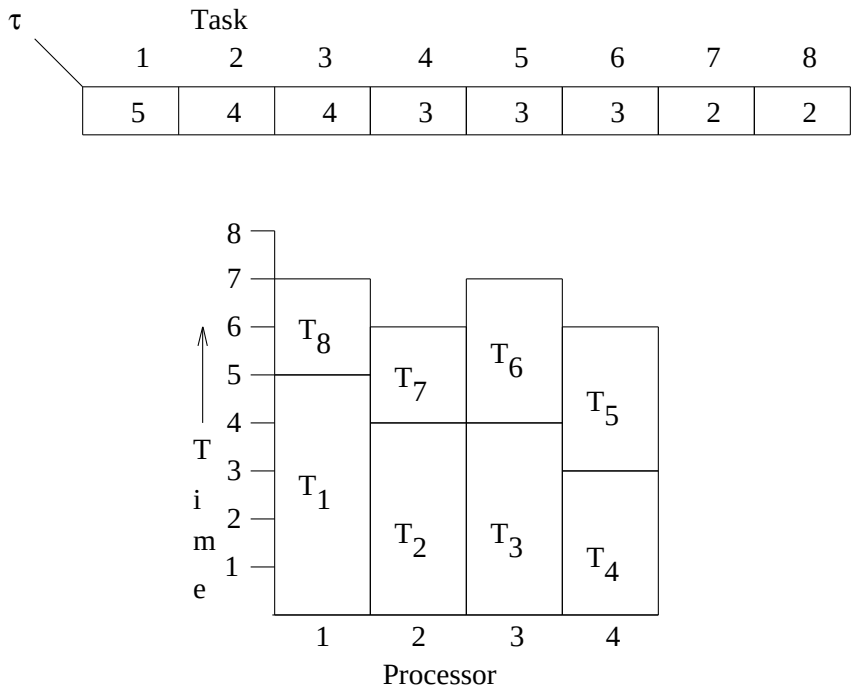


Figure 7.1. Illustration for the LPT algorithm

7.2. Approximate Algorithm

The *parallelizable independent task scheduling (PITS)* problem allows the possibility of executing a task among multiple processors. The problem continues to be NP-hard as we can reduce the *independent task scheduling* problem to the new problem by having the same set of tasks, with speedup functions $\sigma_i(j)=1$ for all i, j, where $1 \le i \le n$, and $2 \le j \le p$. In this section, we present an *approximate* algorithm for the new problem.

The main idea in the algorithm is to construct an *LPT* schedule and iteratively modify it by assigning some tasks to more processors, provided this results in a decrease in the schedule time. At any stage, for every task T_i, we have a number s_i which denotes the number of processors that has been allocated to T_i. During the course of the algorithm, the number of processors allocated to a task may be increased iteratively. Each increase is allowed only if it results in an "immediate" decrease in the schedule time. This is a conservative strategy. The intuitive reason behind this is that increasing

the number of processors allocated to a task will lead to an increase in the cumulative load of all the tasks, which could potentially increase the schedule time. The complete algorithm is described below.

procedure *part_schedule*

[1] Set s_i to 1 for all $1 \le i \le n$. Also, set a to p (the invariant maintained is $a = p - \sum_{s_i > 1} s_i$). Set the current schedule C to be an *LPT* schedule.

[2] If $a = 0$ then go to 5, else determine a task T_i such that for all j, $1 \le j \le n$, $\frac{\tau_i}{\sigma_i(s_i)} \ge \frac{\tau_j}{\sigma_j(s_j)}$. Let $h = \frac{\tau_i}{\sigma_i(s_i)}$.

[3] If $a = 1$ and $s_i = 1$ then go to 5, else do the following. If $f(C) \ne h$ then go to 5, else continue.

[4] If $s_i = 1$, set a to $a - 2$, else set a to $a - 1$. Increment s_i by 1. Construct a schedule C' as follows. The tasks T_j, with $s_j > 1$ are first assigned to the required number of processors (their total requirement is less than p). This is followed by an assignment of the remaining tasks using an *LPT* algorithm. If $f(C') > h$ then decrement s_i and go to 5, else set C to C' and go to 2.

[5] The output schedule A is set to C.

end *part_schedule*.

Some intuitive explanations follow. In Step 3, the conditional statement $f(C) \ne h$ checks if the task T_i is a bottleneck with the current number of allocated processors. If it is, Step 4 checks if allocating one more processor to T_i degrades the overall schedule time. If it does not, one more processor is allocated to T_i and the current schedule is updated.

Example 7.1: We illustrate the algorithm using the problem instance shown in Fig. 7.2(a). The initial solution is shown in Fig. 7.2(b). Going through the steps, T_1 is selected in Step 2, i.e $i = 1$, and h is set to 10. The check in Step 3 succeeds as the current solution is 10, which is equal to h. The task

T_1 is temporarily assigned 2 processors, and the check in Step 4 is performed. As can be seen in Fig. 7.2(c), all the tasks can be scheduled within time 8, which is not higher than 10. Thus T_1 is currently assigned 2 processors. Next, the task T_2 is selected in Step 2, i.e $i = 2$, and h is set to 8. The check in Step 3 succeeds as the current solution is 8, which is equal to h. Thus T_2 is temporarily assigned two processors, and the check in Step 4 is performed. As can be seen in Fig. 7.2(d), all the tasks can be scheduled within time 6, which is not higher than 8. Thus T_2 is currently assigned 2 processors. Next, the task T_1 or T_3 could be selected with $h = 5$, and we assume it is the former. However, the current solution is 6, which is not equal to h. The algorithm is hence terminated, with the schedule in Fig. 7.2(d) as the solution.

We are now ready to analyze the quality of the solution given by the *part_schedule* algorithm. For sake of analysis, we assume that the *efficiency condition*, described in Chapter 5, holds. For an instance of the problem, let A denote the the schedule given by the *part_schedule* algorithm, and let OPT denote an optimum schedule.

Let S be a schedule. We now have some definitions. We define $k_i(S)$ to denote the number of processors assigned to task T_i in schedule S. A task T_i assigned to $j = k_i(S) > 1$ processors can be looked upon as a collection of j sub tasks each having execution time $\frac{\tau_i}{\sigma_i(j)}$. These subtasks will be referred to as *split* tasks. The other tasks T_i, with $k_i(S) = 1$, will be referred to as *unsplit* tasks. In scheduling a collection of *split* and *unsplit* tasks, all *split* tasks that are from the same task are required to start together. The current solution, during the course of the *part_schedule* algorithm, is obtained by first assigning the *split* tasks followed by the assignment of the *unsplit* tasks in an *LPT* fashion. The variable a in the algorithm guarantees that there are at most p *split* tasks. Thus all the *split* tasks start at the same time, satisfying the scheduling requirement.

We define $AVG(S)$ to denote $\frac{1}{p}\sum_{i=1}^{n}\frac{\tau_i k_i(S)}{\sigma_i(k_i(S))}$. Informally $AVG(S)$ gives the average execution load on a processor using schedule S. We define $h(S) = \max\{\frac{\tau_i}{\sigma_i(k_i(S))}, 1 \le i \le n\}$. In other words,

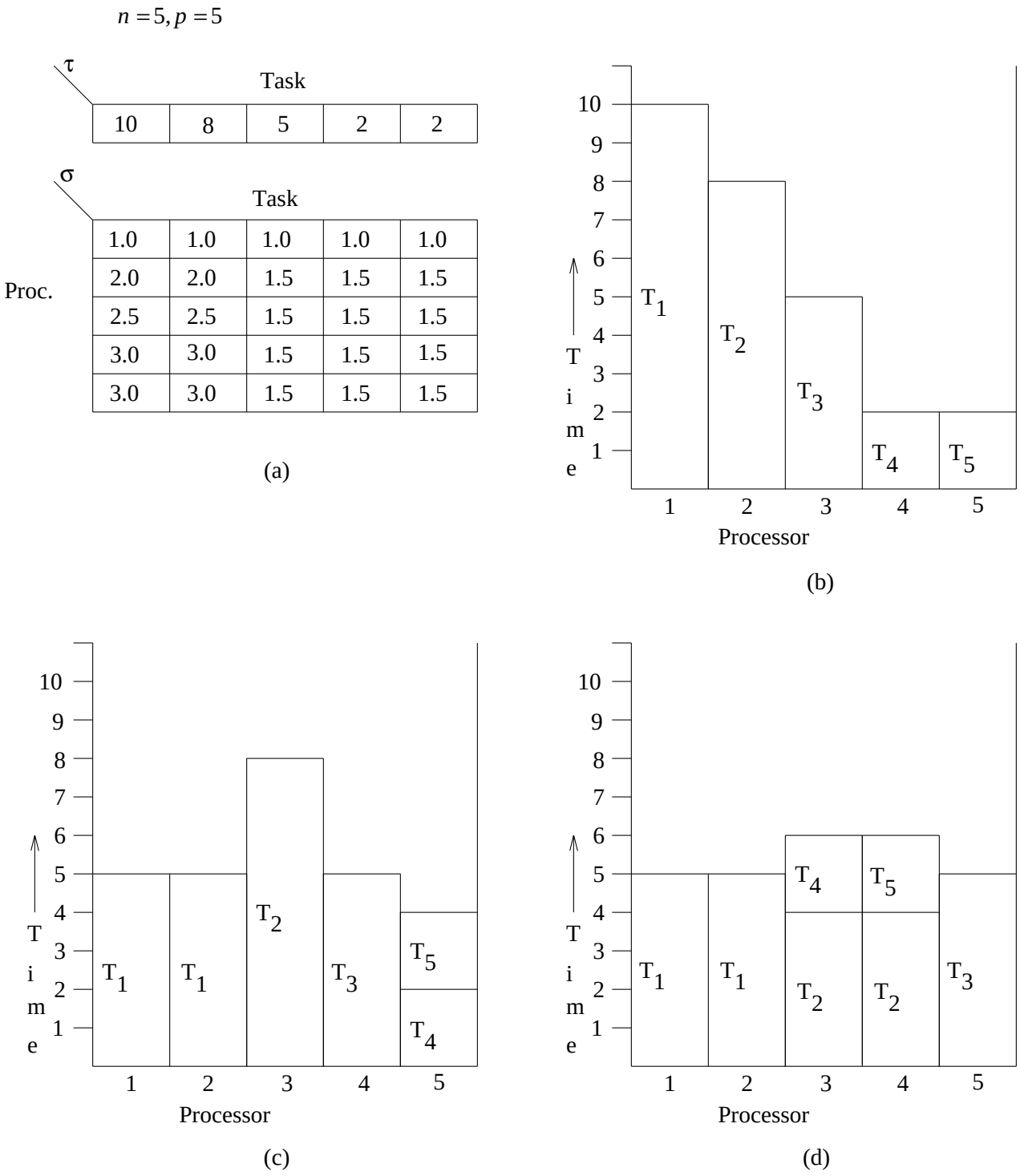


Figure 7.2. Illustration of the part_schedule algorithm

$h(S)$ is the maximum execution time of a *split* or an *unsplit* task in S. As before, we define $f(S)$ to

be the final finish time according to schedule S.

The following lemma gives a lower bound for $f(S)$.

Lemma 7.1: $f(S) \geq \max\{h(S), AVG(S)\}$.

Proof: The finish time cannot be less than the average execution load, and also cannot be less than the execution time of a task in the schedule.

□

The following lemma proves that unless the schedule produced by the *part_schedule* is optimal, it always assigns lesser number of processors to each of the tasks, compared to an optimal algorithm.

Lemma 7.2: If $f(A) \neq f(OPT)$, then $k_i(A) \leq k_i(OPT)$, for all $1 \leq i \leq n$.

Proof: Assume $k_i(A) > k_i(OPT)$ for some i, $1 \leq i \leq n$. Let $l = k_i(A)$ and $m = k_i(OPT)$ for the sake of convenience. Because the number of processors allocated to a task is increased iteratively, there is a stage in the *part_schedule* algorithm when $s_i = m$. The task T_i is assigned one more processor, at some stage in the algorithm, because the condition in Step 3 is true in that stage. This implies that the current schedule at that stage has finish time equal to $\frac{\tau_i}{m}$. Since the current schedule time only improves during the course of the algorithm, we conclude $f(A) \leq \frac{\tau_i}{m}$. From Lemma 7.1, since T_i is assigned to m processors in an optimal solution, we have $f(OPT) \geq \frac{\tau_i}{m}$. This implies $f(A) \leq f(OPT)$. But this is impossible as $f(OPT) < f(A)$, from the definition of optimality and the assumption in the lemma. Thus we have a contradiction, proving the lemma.

□

Lemma 7.3. If $f(A) \neq f(OPT)$, then $AVG(A) \leq AVG(OPT)$.

Proof: $AVG(S)$ of a schedule S can be rewritten as $\frac{1}{p}\sum_{i=1}^{n}\frac{\tau_i}{\eta_i(k_i(S))}$. Because of the *efficiency condition*, this implies that if a task T_i is assigned to more processors, then its contribution to the function *AVG* will increase. From Lemma 7.2, the optimal does not assign lesser number of processors for any task as compared to the *part_schedule* algorithm, proving the lemma.

□

The following lemmas establish an upper bound on the solution obtained during the course of the *part_schedule* algorithm.

Lemma 7.4: Let S be a schedule that is obtained as in Step 4 of the algorithm from a collection of *split* and *unsplit* tasks, where the number of *split* tasks, x, is no more than p. Let y be the execution time of the $(p-x+1)^{th}$ largest *unsplit* task (if there are not that many *unsplit* tasks y is taken as 0). Then, $f(S) \leq \max\{h(S), AVG(S) + y(1-1/p)\}$.

Proof: The finish time of the schedule S could either be the finish time of a task T_i scheduled at time 0, or of some task which was scheduled later. In the first case, T_i must be a *split* or an *unsplit* task with the maximum execution time. In other words, $f(S) = h(S)$. In the other case, the task T_i is an *unsplit* task and $\tau_i \leq y$. This is because the highest $p-x$ *unsplit* tasks are scheduled to start at time 0. The task T_i starts at time $f(S) - \tau_i$. From the property of the *LPT* algorithm, none of the processors have completed before this time. Thus $f(S) - \tau_i \leq AVG'$, where AVG' is average of all the tasks already started at this time. But $AVG' \leq AVG(S) - \frac{\tau_i}{p}$. Thus $f(S) \leq AVG(S) + \tau_i(1-1/p) \leq AVG(S) + y(1-1/p)$, proving the lemma.

□

Lemma 7.5: Let C be a schedule in the *part_schedule* algorithm at some stage. Let z be the execution time of a *split* task in C. Then, $f(C) \leq 2z$.

Proof: Let T_i be the task corresponding to z. Thus, $z = \frac{\tau_i}{\sigma_i(k_i(C))}$. Clearly, $f(C) \leq \frac{\tau_i}{\sigma_i(k_i(C)-1)}$, as one more processor was allocated to T_i at some stage. Because of the *efficiency condition* it can be easily proved that $\frac{\tau_i}{\sigma_i(k_i(C)-1)} \leq \frac{2\tau_i}{\sigma_i(k_i(C))}$. Thus combining the inequalities, we have $f(C) \leq 2z$.

□

Lemma 7.6: Let C be a schedule in the *part_schedule* algorithm at some stage. Let x be the number of *split* tasks and let y be the execution time of the $(p-x+1)^{th}$ largest *unsplit* task. Then

(1) $y \leq \frac{\tau_i}{\sigma_i(k_i(C))}$, for all i such that $k_i(C) > 1$.

(2) $f(C) \leq \max\{ h(C), \frac{2AVG(C)}{(1+1/p)} \}$.

Proof: For (1), we have to prove y is less than the execution times of all the *split* tasks. Let z be the minimum execution time among all the *split* tasks. Let T_i be the task realizing the minimum, i.e $z = \frac{\tau_i}{\sigma_i(k_i(C))}$. From Lemma 7.5, we have $f(C) \leq 2z$. Assuming $z < y$, then we see that $f(C) \geq z+y$. This is because the *unsplit* task corresponding to y will start after the completion of a *split* task with execution time not lesser than z, or an *unsplit* task with execution time not lesser than y. Thus we have $2z \geq z+y$, i.e $z \geq y$, arriving at a contradiction, proving (1).

From (1), the execution time of the *unsplit* task corresponding to y, is less than or equal to the execution time of the x *split* tasks. It also is less than or equal to the execution times of the $p-x$ largest *unsplit* tasks, from the definition of y. Thus $AVG(C) \geq y+y/p$. In other words, $y \leq \frac{AVG(C)}{(1+1/p)}$. Substituting this in the result of Lemma 7.4, we obtain the result of (2).

□

Now, we are ready to describe the main result which bounds the quality of the schedule given by the *part_schedule* algorithm.

Theorem 7.1: For an instance of the problem, let A be the schedule given by the *part_schedule* algorithm, and let OPT be an optimum schedule. Then, $f(A) \leq \frac{2f(OPT)}{(1+1/p)}$.

Proof: We assume $f(A) \neq f(OPT)$, or else the theorem is trivially true. There are two cases.

Case 1: $f(A) \neq h(A)$. From Lemma 7.1, $f(A) \geq \max\{ h(A), AVG(A) \}$. Therefore $f(A) > h(A)$. From statement (2) of Lemma 7.6, this implies $f(A) \leq \frac{2AVG(A)}{(1+1/p)}$. From Lemma 7.3, $AVG(A) \leq AVG(OPT)$, and from Lemma 7.1, $AVG(OPT) \leq f(OPT)$. Combining the inequalities, we have $f(A) \leq \frac{2f(OPT)}{(1+1/p)}$.

Case 2: $f(A) = h(A)$. The theorem is proved in this case by considering the various possible cases of termination in the *part_schedule* algorithm. Let T_b be a task with execution time $h(A)$, i.e $h(A) = \frac{\tau_b}{\sigma_b(k_b(A))}$. Let x be the number of *split* tasks in A.

Case 2.1: The algorithm terminated because $a = 0$ in Step 2 or $a = 1$ and $k_b(A) = 1$ (i.e T_b is an *unsplit* job). The number of *split* tasks is p in the first case, and $p-1$ in the second case. Let z be the minimum execution time of a *split* task, and let T_l be a task realizing this minimum, i.e $z = \frac{\tau_l}{\sigma_l(k_l(A))}$.

It can be seen that $AVG(A) \geq z(1-1/p) + \frac{h(A)}{p}$. This is because there are at least $p-1$ *split* tasks with execution time not lesser than z, and a *split* or an *unsplit* task with execution time equal to $h(A)$. Since $f(A) = h(A)$, the above inequality is rewritten as $z \leq \frac{(AVG(A) - \frac{f(A)}{p})}{(1-1/p)}$. Also, from Lemma 7.5, we have $2z \geq f(A)$. Combining the above two inequalities, we have $f(A) \leq \frac{2(AVG(A) - \frac{f(A)}{p})}{(1-1/p)}$. This can be rewritten as $f(A) \leq \frac{2AVG(A)}{(1+1/p)} \leq \frac{2f(OPT)}{(1+1/p)}$, from Lemma 7.1 and 7.3.

Case 2.2: The algorithm terminated because $f(A) \neq h(A)$ in Step 3. However, this case has been considered in Case 1.

Case 2.3: The schedule C' constructed in Step 4 has $f(C') > h(A)$. The number of *split* tasks in C' is $x' = x+1$. Let y be the $p-x'+1$ largest execution time of an *unsplit* task in C'. Consider the optimum solution. From Lemma 7.2, $k_b(OPT) \geq k_b(A)$. If $k_b(OPT) = k_b(A)$, then from Lemma 7.1, $f(OPT) \geq \frac{\tau_b}{\sigma_b(k_b(OPT))} = h(A) = f(A)$. This implies $f(OPT) = f(A)$ contradicting our assumption. Thus $k_b(OPT) \geq k_b(A)+1 = k_b(C')$. Thus as in Lemma 7.3, we conclude $AVG(C') \leq AVG(OPT) \leq f(OPT)$. We have two sub cases.

Case 2.3.1:

The value y is less than or equal to the execution times of all the *split* tasks in C'. Similar to the proof of statement (2) of Lemma 7.6, we can prove that

$f(C') \leq \max\{ h(C'), \frac{2AVG(C')}{(1+1/p)} \}$. However, $f(C') > h(A) \geq h(C')$. Thus $f(C') \leq \frac{2AVG(C')}{(1+1/p)} \leq \frac{2f(OPT)}{(1+1/p)}$. However, $f(A) = h(A) < f(C')$. The above two inequalities give the required result.

Case 2.3.2:

The value y is greater than the execution times of some *split* tasks in C'. Let z be the minimum execution time of a *split* task in C', and let T_l be the task corresponding to it, i.e $z = \frac{\tau_l}{\sigma_l(k_l(C'))}$. Clearly, in this case $y > z$. Thus z is less than or equal to the execution time of x' *split* tasks and $p - x'$ *unsplit* tasks. Also there is an extra task, corresponding to y. Thus $AVG(C') \geq z + y/p \geq z(1+1/p)$.

If $b \neq l$, T_l is assigned the same number of processors in schedules A and C'. Thus applying Lemma 7.5 to schedule A, we have $f(A) \leq 2z$. In the case $b = l$, $f(A) = \frac{\tau_b}{\sigma_b(k_b(A))} \leq \frac{2\tau_b}{\sigma_b(k_b(A)+1)} = 2z$, as $k_b(C') = k_b(A)+1$. Thus in both cases we have $f(A) \leq 2z$. The above inequalities imply $f(A) \leq \frac{2AVG(C')}{(1+1/p)} \leq \frac{2f(OPT)}{(1+1/p)}$, giving the required result.

□

The above theorem gives an upper bound on the solution given by the *part_schedule* algorithm. We show that this bound is tight by giving a worst case example. Let $n = t$ and $p = 2t-1$. Let $\sigma_i(j) = j$, for all i, j, where $1 \leq i \leq n$, $1 \leq j \leq p$. An optimal schedule and the schedule given by the *part_schedule* algorithm are shown in Fig. 7.3(a) and 7.3(b) respectively. Thus $f(OPT) = \frac{t}{2t-1}$, whereas $f(A) = 1$. Thus, $\frac{f(A)}{f(OPT)} = \frac{2t-1}{t}$. This can be rearranged as $\frac{2}{(1+1/(2t-1))}$, proving that the bound is tight.

Theorem 7.2: The *part_schedule* algorithm can be implemented in $O(n \log n + np \log p)$ time.

Proof: The *part_schedule* algorithm involves at most p calls to the *LPT* algorithm. The sorting required by the *LPT* algorithm needs to be done only once. This is because we can maintain two lists

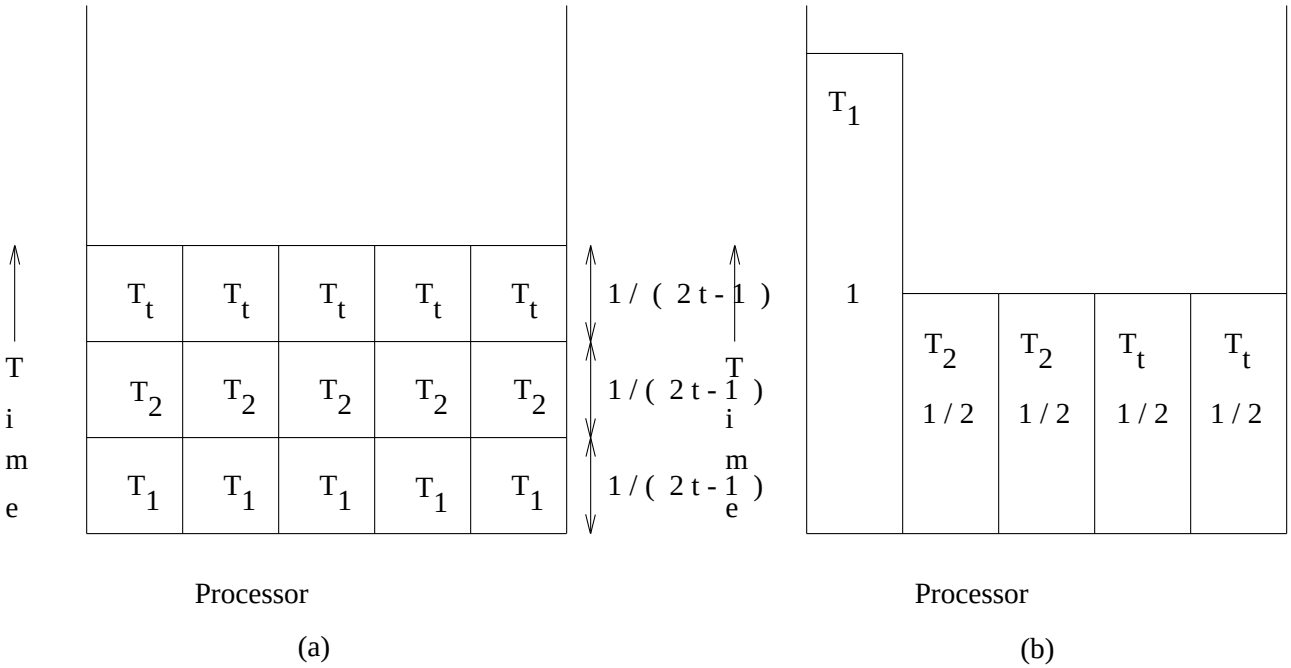


Figure 7.3. (a) Optimal schedule (b) Schedule by part_schedule algorithm

of tasks, during the course of the *part_schedule* algorithm. The first list contains the tasks T_i, with $s_i > 1$, and the second list contains the remaining tasks T_i, with $s_i = 1$. Each of the lists is in sorted order based on the execution times, i.e $\frac{\tau_i}{s_i}$. The details of this implementation are left to the reader.

□

From the proof of Theorem 7.1, it is possible to design a much faster *approximate* algorithm for the *parallelizable independent task scheduling* problem, that has the same worst case performance as the *part_schedule* algorithm. This algorithm, however, may give schedules which are possibly worse when compared to the *part_schedule* algorithm.

The main idea in the new algorithm is to use the upper bound provided by Lemma 7.4, instead of actually constructing an *LPT* schedule, during the course of the algorithm. The algorithm is given below.

procedure *fast_part_schedule*

[1] Set s_i to 1, for all $1 \le i \le n$, and set a to p ($p-a$ gives the current number of *split* tasks). Set *avg* to be $\frac{1}{p}\sum_{j=1}^{n} \tau_j$, and *est* to be $\max\{h, avg + y(1-1/p)\}$, where h, y are the highest and $(p+1)^{th}$ highest execution time of a task respectively. Let T_i be a task with the highest execution time.

[2] If $a = 0$ go to 5, else continue.

[3] If $a = 1$ and $s_i = 1$ then go to 5, else do the following. If $est \neq h$ then go to 5, else do the following.

[4] If $s_i = 1$ set a to $a-2$, else set a to $a-1$. Increment s_i. Now let $T_{i'}$ be a task such that for all j, $\frac{\tau_{i'}}{\sigma_{i'}(s_{i'})} \ge \frac{\tau_j}{\sigma_j(s_j)}$, and let $h' = \frac{\tau_{i'}}{\sigma_{i'}(s_{i'})}$. Also, let y' be the $(a+1)^{th}$ largest execution time of an *unsplit* task. Set *avg*′ to $avg + \frac{1}{p}\left(\frac{\tau_i s_i}{\sigma_i(s_i)} - \frac{\tau_i(s_i-1)}{\sigma_i(s_i-1)}\right)$ and *est*′ to $\max\{h', avg' + y'(1-1/p)\}$. If $h < est'$ then decrement s_i, restore a (i.e if $s_i = 1$, set a to $a+2$, else set a to $a+1$), and go to 5. Set *est* to *est*′, *avg* to *avg*′, h to h' and i to i', and go to 2.

[5] Assign the jobs with $s_i > 1$ to $p-a$ processors. Assign the a largest *unsplit* tasks to the remaining a processors. The other *unsplit* tasks are assigned thereafter using a *list_schedule* algorithm.

end *fast_part_schedule*.

Example 7.2: Let us consider the situation of example 7.1. Following the algorithm, the variables *avg*, h, i, y, *est*, are respectively set to 27/5, 10, 1, 0, and 10 respectively. As $est = h$, the check in Step 3 succeeds. Thus one more processor is temporarily allocated to T_1. The variables *avg*′, h', i', y', *est*′ are respectively set to 27/5, 8, 2, 2, and 8 respectively. Since $est' < h$, T_1 is currently allocated 2 processors. The variables *avg*, h, i, *est* are set 27/5, 8, 2, and 8 respectively. As $est = h$, the check in Step 3 succeeds. Thus one more processor is temporarily allocated to T_2. The variables *avg*′, h', i', y', *est*′, are respectively set to 27/5, 5, 3, 2, and 7. Since $est' < h$, T_2 is currently

allocated 2 processors. The variables avg, h, i, est are set to 27/5, 5, 3, and 7 respectively. Now the check in Step 3 fails as $est \neq h$. Thus a schedule is constructed in Step 5. This schedule is exactly the same schedule as in Fig. 7.2(d).

The following theorem bounds the performance of the *fast_part_schedule* algorithm.

Theorem 7.3. For an instance, let A be the schedule output by the *fast_part_schedule* algorithm, and let OPT be an optimal schedule. Then, $f(A) \leq \frac{2f(OPT)}{(1+1/p)}$.

Proof: This proof is along the similar lines as in Theorem 7.1.

□

The worst case example for *part_schedule* algorithm is also a worst case example for the *fast_part_schedule* algorithm.

Theorem 7.4. The *fast_part_schedule* algorithm can be implemented in time $O((n+p)\log p)$ time.

Proof: Other than the computation of avg in Step 1, and the final construction of the schedule in Step 5, the algorithm needs to look at only a subset S of all the tasks. This subset consists of the highest $p+1$ tasks of the problem. This can be obtained in $O(n \log p)$ time. During the course of the algorithm, the set of tasks in S are represented in two trees. The first tree consists of tasks T_i, with $s_i > 1$. The second tree consists of tasks T_i, with $s_i = 1$. Each of the trees is a balanced search tree, based on the current execution time, i.e $\frac{\tau_i}{s_i}$.

Step 1 can be computed in $O(n \log p)$ time. The operations in Steps 2, 3, 4 can be performed in $O(\log p)$ time per iteration. There are $O(p)$ iterations, thus these steps take a total time of $O(p \log p)$. The a largest *unsplit* tasks in Step 5 can be obtained directly from the second tree. The *list_schedule* algorithm can be implemented in $O(n \log p)$ time. Thus the total time taken by the algorithm is $O((n+p)\log p)$. The details of this implementation are left to the reader.

□

7.3. Summary

In this chapter, we have presented *approximate* algorithms for the *parallelizable independent task scheduling* problem. An open problem is to design an algorithm for the problem that has a better worst case bound. This may be difficult for the general case of the problem. It might be possible to generate better algorithms under some more assumptions on the speedup function σ. For instance, we may assume $\sigma_i(k) = \sigma_j(k)$, for any two tasks T_i and T_j.

CHAPTER 8.

CONCLUSIONS

As the sizes of VLSI circuits increases in the future, the computational requirements for performing various computer-aided design (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. Parallel processing provides an effective way for increasing the capabilities of the CAD tools. It not only reduces the time taken by various CAD tasks, but also increases the sizes of circuits that can be effectively handled by the CAD tools.

General purpose multiprocessors can offer the tremendous speed improvements that are needed to solve the complex problems of the future. Many commercial general-purpose multiprocessors have been built and are being widely used for various scientific applications. In this thesis, we have looked at the application of parallel processing to computer aided design problems, with special emphasis to the problem of VLSI circuit extraction.

In Chapter 2, we described parallel algorithms for VLSI circuit extraction on hypercube and shared memory multiprocessors. The basic idea is to divide the input circuit into smaller parts, perform various tasks in circuit extraction on each of the parts in parallel, the results of which are then combined to complete the extraction of the whole circuit. The issues in parallel circuit extraction are the design of efficient algorithms to be used in the partitioning and combining steps. In Chapter 2, we have described our implementation on Intel iPSC2 hypercube and Encore 510 Multimax, and have given performance results for different partitioning strategies. These results show that significant amount of speedups can be achieved for this problem. The results also show that point based parti-

tioning, which divides the circuit so that the sub regions get nearly the same number of rectangles, is an effective load balancing technique.

In Chapter 3, we described solutions to the *geometric connected component labeling* problem, that occurs as a sub problem in the combining phase of parallel extraction. This problem is similar to the *image component labeling* problem, that has been a subject for a great deal of research. However, the results of this research are largely of a theoretical nature. In Chapter 3, we have described some practical algorithms for the *GCCL* problem, that were implemented as a part of the parallel extractor. For a hypercube, we have described two different algorithms for the problem and compared their performance. These algorithms consist of different stages of message passing. At each stage, a processor increases its information region in the circuit. The more sophisticated algorithm uses overlap of information regions to reduce the message and computational requirements. For a shared memory multiprocessor, we have presented a simple but efficient algorithm for the problem. This algorithm uses a parallel union-find algorithm. It ensures correctness using locks.

In Chapter 4, we described solutions to problem of maintaining *recursively partitioned* data structures of points on multiprocessors, under the operations of addition and deletion of points. This problem occurs as a sub problem in the implementation of point based partitioning on distributed memory multiprocessors. The partitioning lines have to be updated as points are added or deleted from the distributed data structure. In Chapter 4, we have presented two algorithms for this problem, and have given some implementation results on the Intel iPSC2 hypercube. We have analyzed the complexities of the algorithms, and have discussed the improvements that can be made for better average case performance.

In Chapter 5, we took a general look at the use of hierarchical methods in speeding various computer-aided design problems. We have described how the benefits of hierarchical analysis and parallelism can be combined for a variety of problems. Towards this goal, we formulated the *parallelizable dependent task scheduling* problem. This problem involves scheduling a task graph,

wherein each of the tasks can be executed in parallel. This theory was then applied for a case study in parallel hierarchical circuit extraction. We have presented results for the case study on an Encore 510 multimax.

In Chapter 6, we presented an algorithm for the *parallelizable dependent task scheduling problem*. The algorithm has two steps. In the first step we decide the number of processors to be allocated to each of the tasks. The second step is to construct a schedule from this information. We have analyzed the performance of this algorithm with respect to the optimal, and have proved some bounds under some assumptions on the problem.

In Chapter 7, we looked at a special case of the problem, wherein the tasks are independent. We have presented an algorithm for the *parallelizable independent task scheduling* problem, which is basically a down hill technique. At each step of the algorithm, we consider allocating one more processor to the task with the current highest execution time. This move is accepted only if the resulting schedule improves. We have analyzed the performance of the algorithm to get a better bound with respect to the optimal, as compared to the general case of the problem.

This research has opened up some new directions for future work. On the practical side, the tradeoffs between hierarchy and parallelism must be demonstrated on other problems in computer-aided design. Towards this goal, we need to design better heuristics for the task scheduling problems, that take into account other practical considerations. On the theoretical side, we need to investigate approximate algorithms for the task scheduling problems that improve on the bounds presented in Chapter 6 and 7.

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] P. Banerjee, M. Jones, J. Sargent, R. J. Brouwer, K. P. Belkhale, A. Hagin, and S. Patil, ‘‘Parallel Algorithms for Placement, Routing, Extraction, Test Generation and Simulation on Hypercube Multiprocessors,’’ in *Advances in VLSI Computer-Aided Design*, ed., I. N. Hajj. England: JAI Press, 1988.

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

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

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] Yossi Shiloach and Uzi Vishkin, ‘‘An O(log n) Parallel Connectivity Algorithm,’’ *Journal of Algorithms*, vol. 3, pp. 57-67, 1982.

[29] D. Nassimi and S. Sahni, ‘‘Data broadcasting in SIMD computers,’’ *IEEE Trans. on Computers*, vol. C-30, No. 2, pp. 101-107, 1981.

[30] David Nassimi and Sartaj Sahni, ‘‘Finding Connected Components and Connected Ones on a Mesh-Connected Parallel Computer,’’ *SIAM Journal on Computing*, vol. 9, 4, pp. 744-757, 1980.

[31] R. Cypher, J. L. C. Sanz, and L. Snyder, ‘‘Hypercube and Shuffle-Exchange Algorithms for Image Component Labeling,’’ *Journal of Algorithms*, vol. 10, No. 1, pp. 140-150, 1989.

[32] R. E. Tarjan, ‘‘Efficiency of a good but non linear set union algorithm,’’ *Journal of the ACM*, vol. 22, pp. 215-225, 1975.

[33] T. G. Allen and G. Cybenko, ‘‘Recursive Binary Partitions,’’ in *Technical Report 87-14*, Medford, MA, 1987.

[34] F. P. Preparata and M. I. Shamos, in *Computational Geometry*. New York: Springer-Verlag, 1985.

[35] T. S. Payne and W. M. Van Cleemput, ‘‘Automated partitioning of hierarchically specified digital systems,’’ *Proc. 19th Design Automation Conf.*, pp. 182-192, 1982.

[36] K. H. Khokhani, A. M. Patel, W. Ferguson, J. Sena, and D. Hatton, ‘‘Placement of variable size circuits on LSI master slices,’’ *Proc. 18th Design Automation Conf.*, pp. 426-434, 1981.

[37] B. W. Kernighan and S. Lin, ‘‘An Efficient Heuristic for Partitioning Graphs,’’ *Bell Syst. Tech. Jour.*, Feb. 1970.

[38] M. Burstein and R. Pelavin, ‘‘Hierarchical Wire Routing,’’ *IEEE Trans. Computer Aided Design*, vol. 2, No. 4, pp. 223-234, Oct. 1983.

[39] W. K. Luk, D. T. Tang, and C. K. Wong, ‘‘Hierarchical Global Wiring for Custom Chip Design,’’ *Proc. 23rd Design Automation Conference*, pp. 481-489, June 1986.

[40] J. L. Burns and A. R. Newton, ‘‘SPARCS: a new constraint based IC symbolic layout spacer,’’ *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 534-539, May 1986.

[41] C. Y. Lo, R. Varadarajan, and W. H. Crocker, ‘‘Compaction with performance optimization,’’ *Proc. Intl. Conf. on Circuits and Systems*, pp. 514-517, 1987.

[42] S. C. Johnson, ‘‘Hierarchical design validation based on rectangles,’’ *Conf. on Advanced Research in VLSI*, pp. 97-100, 1982.

[43] N. F. Chen and C. L. Liu, ‘‘On a class of scheduling algorithms for multiprocessor computing systems,’’ in *Parallel Processing, Lecture Notes in Computer Science*, vol. 24, Berlin, pp. 1-16, 1975.

[44] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, ‘‘Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey,’’ *Annals of Discrete Mathematics*, vol. 5, pp. 287-326, 1979.

[45] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, ‘‘Sequencing and Scheduling: Algorithms and Complexity,’’ in *Report BS-R8909, Center for Mathematics and Computer Science*, P. O. Box 4079, Amsterdam, The Netherlands, 1989.

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

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

[48] R. L. Graham, ‘‘Bounds on Multiprocessing Timing Anomalies,’’ *SIAM J. Appl. Math.*, vol. 17, pp. 263-269, 1969.

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. New York, N. Y.: W. H. Freeman and Company, 1979.

[50] C.D. Polychronopoulos and U. Banerjee, ‘‘Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds,’’ *IEEE Trans. on Computers*, vol. c-36(4), pp. 410-420, April 1987.

[51] G. M. Amdahl, ‘‘Validity of the single processor approach to achieving large-scale computing capabilities,’’ in *AFIPS Conference Proceedings*, vol. 30, Montvale, N.J, p. 483, 1967.

[52] D. G. Luenberger, *Introduction to Linear and Nonlinear Programming*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1973.

[53] M. R. Garey, R. L. Graham, and D. S. Johnson, ‘‘Performance guarantees for Scheduling Algorithms,’’ *Operations Research*, vol. 26, No. 1, pp. 3-21, Jan. 1978.

VITA

Krishna Belkhale obtained the Bachelor degree in Computer Science and Engineering, from the Indian Institute of Technology, Madras, in May 1986. He obtained the Master degree in Computer Science from University of Illinois, Urbana-Champaign in Aug. 1988. At University of Illinois, he was a teaching assistant in the Department of Computer Science, from Aug. 1986 - May. 1987. Subsequently, he was employed as a research assistant in the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory, from May. 1987 - Dec. 1990.

After his graduation, he will be joining IBM, East Fishkill, N.Y. His research interests are in the areas of parallel processing, computer-aided design, design and analysis of algorithms. His publications are

[1] Prasad, B. K, Rangan, C. P., "Inverted Spanning tree Paradigm on Systolic Arrays", *Proc. Intl. Workshop on Systolic Arrays*, University of Oxford, July 1986.

[2] K. P. Belkhale, P. Banerjee, "Reconfiguration Strategies in VLSI Processor Arrays", *Proc. Int. Conf. on Computer-Design*, Oct. 1988.

[3] K.P. Belkhale, P. Banerjee, "PACE: A Parallel VLSI Circuit Extractor on the Intel Hypercube Multiprocessor", *Proc. Int. Conf. on Computer-Aided Design*, Nov. 88.

[4] P. Banerjee, M. Jones, J. Sargent, R. J. Brouwer, K. P. Belkhale and S. Patil, "Parallel Algorithms for VLSI Computer-Aided Design Tools on Hypercube Multiprocessors", a book chapter in *Advances in VLSI Computer-Aided Design*, ed., I.N. Hajj, England: JAI Press, 1989.

[5] K. P. Belkhale, P. Banerjee, "PACE2: An Improved Parallel VLSI Extractor with Parameter Extraction", *Proc. Int. Conf. on Computer-Aided Design*, Nov. 89.

[6] K. P. Belkhale, P. Banerjee, "Recursive Partitions on Multiprocessors", *Proc. 5th Distributed Memory Computing Conf*, April 1990.

[7] K. P. Belkhale, P. Banerjee, "An approximate algorithm for the partitionable independent task scheduling problem", *Proc. Int. Conf. on Parallel Processing*, Aug. 1990.

[8] K. P. Belkhale, P. Banerjee, "Parallel Algorithms for Geometric Connected Component Labeling on a Hypercube Multiprocessor", *Proc. Int. Conf. on Parallel Processing*, Aug. 1990.

[9] K. P. Belkhale, P. Banerjee, "Reconfiguration Strategies in VLSI Processor Arrays", Accepted for publication in *IEEE Transactions on Computers*.

[10] K. P. Belkhale, P. Banerjee, "A Parallel Algorithm for VLSI Circuit Extraction", Accepted for publication in *IEEE Transactions on CAD*.

[11] K. P. Belkhale, P. Banerjee, "A Parallel Algorithm for Hierarchical Circuit Extraction", To appear in *Proc. Int. Conf. on Computer Aided Design, 1990*.

PARALLEL ALGORITHMS FOR CAD WITH APPLICATIONS TO CIRCUIT EXTRACTION

Krishna P. Belkhale, Ph.D.
Department of Computer Science
University of Illinois, Urbana-Champaign, 1991

As the sizes of VLSI circuits increases in the future, the computational requirements for performing various computer-aided design (CAD) tasks will increase tremendously. In this thesis, we look at the application of parallel processing to computer aided design problems, with special emphasis to the problem of VLSI circuit extraction. We first describe parallel algorithms for VLSI circuit extraction on hypercube and shared memory multiprocessors. We give the performance results of the algorithms on an Intel iPSC2 hypercube and an Encore Multimax shared memory multiprocessor, for different partitioning strategies.

There were two sub problems in the above research that are of general importance. The first problem is called the *geometric connected component labeling* problem, that arises in the combining phase of parallel extraction. We describe some practical algorithms for this problem. The second problem involves maintaining *recursively partitioned* data structures of points on distributed memory multiprocessors, under the operations of addition and deletion of points. This problem arises when partitioning strategies are considered, that divide a circuit so that the resulting sub regions have approximately the same number of rectangles. We describe two algorithms for this general problem, and analyze their complexity.

The next step is to take a general look at the use of hierarchical methods in computer-aided design. We describe how the benefits of hierarchical analysis and parallelism can be combined. Towards this goal, we formulate the *parallelizable dependent task scheduling (PDTS)* problem, which is NP-hard. We describe a heuristic for the problem, and analyze its performance with respect to the optimal. We also consider a special case of the *PDTS* problem, wherein the tasks are indepen-

dent. We present an algorithm for this problem and analyze its performance to get a better bound with respect to the optimal, as compared to the general case of the problem. This theory is then applied for a case study in parallel hierarchical circuit extraction. We present some implementation results on an Encore Multimax multiprocessor.

DEDICATION

To Mom and Dad

ACKNOWLEDGMENTS

I would like to express my gratitude to the following people:

My thesis advisor Prof. Prithviraj Banerjee for his exceptional guidance during the course of the research. He was always available for discussion and motivation. I will always cherish the long hours of discussion that I had with him.

Prof. Daniel Reed, Prof. Resve Saleh, Prof. Franco Preparata, and Prof. David Liu for serving on my committee.

My parents and family, for their love, support and encouragement, without which I would not have been able to accomplish my goals. Their confidence in me has been a driving force during my stay in Champaign-Urbana.

My friends at Univ. of Illinois and Center for Reliable and High-Performance Computing group, for their friendship and love.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
    1.1. Parallel Processing for CAD 1
    1.2. Circuit Extraction Problem 5
        1.2.1. Netlist Extraction 5
        1.2.2. Parameter Extraction 9
    1.3. Thesis Overview 14

2. PARALLEL ALGORITHMS FOR VLSI CIRCUIT EXTRACTION 15
    2.1. Circuit Partitioning and Mapping 16
        2.1.1. Circuit Partitioning 16
        2.1.2. Implementation on a Hypercube Multiprocessor 18
        2.1.3. Implementation on a Shared Memory Multiprocessor 19
    2.2. Parallel Netlist Extraction 21
    2.3. Parallel Parameter Extraction 25
    2.4. Analysis 33
        2.4.1. Analysis for a Hypercube Multiprocessor 33
        2.4.2. Analysis for a Shared Memory Multiprocessor 35
    2.5. Results and Discussion 35
        2.5.1. Results for a Hypercube 35
        2.5.2. Results on a Shared Memory Multiprocessor 39
        2.5.3. Comparison on an Uniprocessor 39
    2.6. Summary 41

3. GEOMETRIC CONNECTED COMPONENT LABELING 42
    3.1. Geometric Connected Component Labeling Problem 42
    3.2. Previous Work 43
    3.3. Algorithms for a Hypercube Multiprocessor 47
        3.3.1. Quad Algorithm 47
            3.3.1.1. Outline of Quad Algorithm 47
            3.3.1.2. Proof of Correctness of Quad Algorithm 51
            3.3.1.3. Analysis of Quad Algorithm 53
        3.3.2. Overlap Slice Algorithm 54
            3.3.2.1. Outline of Overlap Slice Algorithm 55
            3.3.2.2. Proof of Correctness of Overlap Slice Algorithm 58

3.3.2.3. Analysis of Overlap Slice Algorithm 60
3.3.3. Overlap Quad Algorithm 61
3.3.3.1. Outline of Overlap Quad Algorithm 62
3.3.3.2. Proof of Correctness of Overlap Quad Algorithm 65
3.3.3.3. Analysis of Overlap Quad Algorithm 68
3.3.4. Implementation 69
3.4. Algorithm for a Shared Memory Multiprocessor 70
3.4.1. Parunion Algorithm 70
3.4.2. Analysis of Parunion Algorithm 72
3.5. Summary 73

4. RECURSIVELY PARTITIONED DISTRIBUTED DATA STRUCTURES 74
4.1. Recursively Partitioned Data Structure 75
4.2. Algorithms for Slice Partitioned Data Structure 76
4.3. Algorithms for Recursively Partitioned Data Structures 78
4.3.1. General Algorithm 1 79
4.3.2. General Algorithm 2 81
4.4. Analysis for a Hypercube 86
4.5. Implementation and Results 89
4.6. Extensions 91
4.7. Summary 92

5. TRADEOFFS IN PARALLELISM AND HIERARCHY 93
5.1. Use of Hierarchy 93
5.2. Combining Hierarchy and Parallelism 96
5.3. Case Study : Circuit Extraction 99
5.3.1. Hierarchical Circuit Extraction 99
5.3.2. Parallel Hierarchical Extraction 100
5.3.2.1. Algorithm LPT 100
5.3.2.2. Algorithm PITS 101
5.3.2.3. Results 102
5.4. Summary 103

6. PARALLELIZABLE DEPENDENT TASK SCHEDULING PROBLEM 105
6.1. Approximate Algorithm 106
6.1.1. The Allocation Step 107
6.1.2. Mapping Step 109
6.2. Analysis of the algorithm 110
6.3. Summary 115

7. PARALLELIZABLE INDEPENDENT TASK SCHEDULING PROBLEM 116
7.1. Previous Work 116
7.2. Approximate Algorithm 118
7.3. Summary 130

8. CONCLUSIONS 131

REFERENCES. 134

VITA. 137

LIST OF FIGURES

FIGURE PAGE

1.1. (a) SIMD and (b) MIMD general purpose multiprocessors 2
1.2. (a) Shared Memory and (b) Distributed Memory Multiprocessors 3
1.3. HIPERCAD (HIgh PERformance CAD) project 4
1.4. Identification of device layers in CMOS technology 6
1.5. Inverter Circuit 7
1.6. The scan line moving from left to right 8
1.7. Device size extraction for CMOS technology 10
1.8. Different types of capacitances 11
1.9. The first two steps in resistance extraction 12
1.10. The rectangle decomposition of A 12
1.11. Rectangle decomposition and branch creation 13
1.12. An example RC network 14
2.1. Area based (a) Slice Partitioning (b) Rectangular Partitioning 17
2.2. Point based (a) Slice Partitioning (b) Rectangular Partitioning 17
2.3. Implementing approximate point based partitioning 20
2.4. Parallel device identification for CMOS technology 21
2.5. (a) Set of rectangles (b) Maximal border segments after step P4 23
2.6. Determination of global nets, devices 26
2.7. Interfacing *MEXTRA* model for extraction 28
2.8. Interfacing *HPEX* model for extraction 30
2.9. (a) The example circuit (b) Rectangles processed by processor 0 32
2.10. *PACE* (*HPEX* model) with rectangular partitioning (area versus point) 38
2.11. *PACE* (*HPEX* model) with point based partitioning (slice versus rectangular) 39
3.1. Example grid 43
3.2. Applying divide and conquer 46
3.3. Information region of $G(i,j)$ before k^{th} stage 48
3.4. Proc $P=G(i,j)$ in an x axis communication with $Q=G(r,j)$ 49
3.5. Message structure for the Quad algorithm 50
3.6. Example illustration for Lemma 3.1 52
3.7. Information region of $G(i,0)$ before the k^{th} stage 55
3.8. $P=G(i,0)$ communicating with $O=G(r,0)$, $Q=G(s,0)$ for the k^{th} stage 56
3.9. Message structure of Overlap Slice algorithm 57
3.10. Example illustration for Lemma 3.3 59
3.11. The information region of $G(i,j)$ before the k^{th} stage 61

3.12. $P=G(i,j)$ in a x axis communication with $O=G(r,j)$ and $Q=G(s,j)$ 63
3.13. The message structure of the Overlap slice algorithm 64
3.14. Example illustration for Lemma 3.5 66
3.15. Example partitioning (b) Illustration for Parunion algorithm 71
3.16. The union algorithm 72
4.1. A recursively partitioned data structure 76
4.2. The bottom-up combining operation 80
4.3. The main idea of general algorithm 2 81
4.4. Step 2 of the *decompose* algorithm 83
4.5. Gray code mapping 87
5.1. Hierarchical representation of a circuit 94
5.2. (a) Example task graph (b) Time and speedup functions for tasks 97
5.3. Speedup curve used in the implementation 101
5.4. (a) *LPT* based (b) *PITS* based 104
6.1. An example schedule 106
6.2. (a) Case 1 (b) Case 2 of Theorem 6.1 111
6.3. (a) Allocation step (b) Mapping Step 114
7.1. Illustration for the LPT algorithm 118
7.2. Illustration of the part_schedule algorithm 121
7.3. (a) Optimal schedule (b) Schedule by part_schedule algorithm 127

LIST OF TABLES

TABLE		PAGE
2.1.	Performance of *PACE* with *HPEX* model on Intel iPSC2/D4-MX	36
2.2.	Performance of *PACE* with *MEXTRA* model on Intel iPSC2/D4-MX	37
2.3.	Performance of *PACE* with *HPEX* model on Encore 510 multimax	40
2.4.	Performance of *PACE* sequential implementation on Sun 3/280 and Sun 4/280	40
3.1.	Performance on a 4 dimensional hypercube	69
3.2.	Performance on a 7-dimensional hypercube	70
3.3.	Performance on a 8-dimensional hypercube	70
4.1.	Time (in ms) for slice partitioned data structure	89
4.2.	Time in (ms) for rectangle partitioned data structure	90
4.3.	Time (in ms) for cube partitioned data structure	90
5.1.	Time taken in seconds by the 5 algorithms with *HPEX* model	102
5.2.	Breakup of time for sequential and parallel hierarchical extraction	103